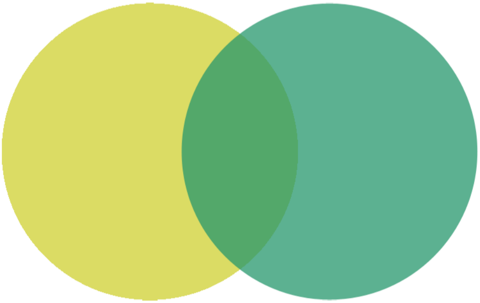
AUSTIN BINGHAM & ROBERT SMALLSHIRE

THE PYTHON



JOURNEYMAN

SixtyNORTH

The Python Journeyman

Robert Smallshire and Austin Bingham

This book is for sale at http://leanpub.com/python-journeyman

This version was published on 2018-01-02

ISBN 978-82-93483-04-5

Contents

Preface

In 2013, when we incorporated our Norway-based software consultancy and training business *Sixty North*, we were courted by *Pluralsight*, a publisher of online video training material, to produce Python training videos for the rapidly growing MOOC market. At the time, we had no experience of producing video training material, but we were sure we wanted to carefully structure our introductory Python content to respect certain constraints. For example, we wanted an absolute minimum of forward references since those would be very inconvenient for our viewers. We're both men of words who live by Turing Award winner Leslie Lamport's maxim *"If you're thinking without writing you only think you're thinking"*, so it was natural for us to attack video course production by first writing a script.

In the intervening years we worked on three courses with *Pluralsight*: Python Fundamentals[1], Python – Beyond the Basics[2], and Advanced Python[3]. These three online courses have been transformed into three books The Python Apprentice[4], this one, The Python Journeyman[5], and The Python Master[6].

You can read *The Python Journeyman* either as a standalone Python tutorial, or as the companion volume to the corresponding *Python – Beyond the Basics* video course, depending on which style of learning suits you best. In either case we assume that you're up to speed with the material covered in the preceding book or course.

Errata and Suggestions

All the material in this book has been thoroughly reviewed and tested; nevertheless, it's inevitable that some mistakes have crept in. If you do spot a mistake, we'd really appreciate it if you'd let us know via the *Leanpub* Python Journeyman Discussion[7] page so we can make amends and deploy a new version.

[1] https://www.pluralsight.com/courses/python-fundamentals
[2] https://www.pluralsight.com/courses/python-beyond-basics
[3] https://www.pluralsight.com/courses/advanced-python
[4] https://leanpub.com/python-journeyman
[5] https://leanpub.com/python-journeyman
[6] https://leanpub.com/python-master
[7] https://leanpub.com/python-journeyman/feedback

Conventions Used in This Book

Code examples in this book are shown in a fixed-width text which is colored with syntax highlighting:

```
>>> def square(x):
...     return x * x
...
```

Some of our examples show code saved in files, and others — such as the one above — are from interactive Python sessions. In such interactive cases, we include the prompts from the Python session such as the triple-arrow (>>>) and triple-dot (...) prompts. You don't need to type these arrows or dots. Similarly, for operating system shell-commands we will use a dollar prompt ($) for Linux, macOS and other Unixes, or where the particular operating system is unimportant for the task at hand:

```
$ python3 words.py
```

In this case, you don't need to type the $ character.

For Windows-specific commands we will use a leading greater-than prompt:

```
> python words.py
```

Again, there's no need to type the > character.

For code blocks which need to be placed in a file, rather than entered interactively, we show code without any leading prompts:

```
def write_sequence(filename, num):
    """Write Recaman's sequence to a text file."""
    with open(filename, mode='wt', encoding='utf-8') as f:
        f.writelines("{0}\n".format(r)
                     for r in islice(sequence(), num + 1))
```

We've worked hard to make sure that our lines of code are short enough so that each single logical line of code corresponds to a single physical line of code in your book. However, the vagaries of publishing e-books to different devices and the very genuine need for occasional long lines of code mean we can't guarantee that lines don't wrap. What we can guarantee, however, is that where a line does wrap, the publisher has inserted a backslash character \ in the final column. You need to use your judgement to determine whether this character is legitimate part of the code or has been added by the e-book platform.

```
>>> print("This is a single line of code which is very long. Too long, in fact, to fi\
t on a single physical line of code in the book.")
```

If you see a backslash at the end of the line within the above quoted string, it is *not* part of the code, and should not be entered.

Occasionally, we'll number lines of code so we can refer to them easily from the narrative next. These line numbers should not be entered as part of the code. Numbered code blocks look like this:

```
def write_grayscale(filename, pixels):
    height = len(pixels)
    width = len(pixels[0])

    with open(filename, 'wb') as bmp:
        # BMP Header
        bmp.write(b'BM')

        # The next four bytes hold the filesize as a 32-bit
        # little-endian integer. Zero placeholder for now.
        size_bookmark = bmp.tell()
        bmp.write(b'\x00\x00\x00\x00')
```

Sometimes we need to present code snippets which are incomplete. Usually this is for brevity where we are adding code to an existing block, and where we want to be clear about the block structure without repeating all existing contents of the block. In such cases we use a Python comment containing three dots `# ...` to indicate the elided code:

```
class Flight:

    # ...

    def make_boarding_cards(self, card_printer):
        for passenger, seat in sorted(self._passenger_seats()):
            card_printer(passenger, seat, self.number(), self.aircraft_model())
```

Here it is implied that some other code already exists within the `Flight` class block before the `make_boarding_cards()` function.

Finally, within the text of the book, when we are referring to an identifier which is also a function we will use the identifier with empty parentheses, just as we did with `make_boarding_cards()` in the preceding paragraph.

Welcome Journeyman!

Welcome to *The Python Journeyman*. This is a book for people who already know the essentials of the Python programming language and are ready to dig deeper, to take the steps from apprentice to journeyman. In this book, we'll cover topics to help prepare you to produce useful, high-quality Python programs in professional, commercial settings. Python is a large language, and even after this book there will be plenty left for you to learn, but we will teach you the tools, techniques, and idioms you need to be a productive member of any Python development team.

Before we really start, it's important that you are comfortable with the prerequisites to get the most of this book. We will assume that you know quite a bit already, and we'll spend very little time covering basic topics. If you find that you need to brush up on Python basics before you start this book, you should refer to the first book in our trilogy, *The Python Apprentice*.

Prerequisites

A functioning Python 3 runtime

First and foremost, you will need access to a working Python 3 system. Any version of Python 3 will suffice, and we have tried to avoid any dependencies on Python 3 minor versions. With that said, more recent Python 3 versions have lots of exciting new features and standard library functionality, so if you have a choice you should probably get the most recent stable version.

At a minimum, you need to be able to run a Python 3 REPL. You can, of course, use an IDE if you wish, but we won't require anything beyond what comes with the standard Python distribution.

Defining function, passing arguments, and returning values

You will need to know how to define functions, and you need to be comfortable with concepts like keyword arguments, default argument values, and returning values from functions.

Creating, importing and executing modules

Likewise, you will need to know how to work with basic, single-file modules in Python. We'll be covering *packages* in the next chapter, but we won't spend any time covering basic module topics like creating modules or importing them.

Built-in types: Integers, Floats, Strings, Lists, Dictionaries and Sets

We will make extensive use of Python's basic built-in types, so you need to make sure that you are fluent in their syntax and application. In particular, you need to make sure that you know the following types well:

- `int`
- `float`
- `string`
- `list`
- `dict`
- `set`

Many of our examples use these types liberally and without undue explanation, so review these before proceeding if necessary.

Fundamentals of the Python object model

Like the basic types we just mentioned, this book assumes that you are familiar with the basic Python object model. *The Python Journeyman* goes into greater depth on some advanced object-model topics, so make sure you understand concepts like single-inheritance, instance attributes, and other topics covered in *The Python Apprentice*.

Raising and handling exceptions

In Python, exceptions are fundamental to how programs are built. We'll assume that you're familiar with the basic concept of exceptions, as well as the specifics of how to work with them in Python. This includes raising exceptions, catching them, and `finally` blocks.

Iterables and iterators

In this book you will learn how to define iterable objects and iterators, but we expect you to already know how to use them. This includes syntax like the `for`-loop as well as how to use the `next()` and `iter()` functions to manually iterate over sequences.

Defining classes with methods

Like functions which we mentioned earlier, classes are a basic part of Python and we'll expect you to be very comfortable with them in this course. You will need to know how to define classes and give them methods, as well as create and work with instances of them.

Reading and writing text and binary files

In Python, as with many languages, you can treat the data in files in one of two basic ways, text and binary. In this book we'll work with both kinds of files, so you need to make sure that you understand the distinction and the ramifications of the two different modes. And of course you need to know how to work with files in general, including opening, closing, reading from, and writing to them.

Unit testing, debugging and deployment

Before you start this book, make sure you that you are familiar with unit testing, debugging, and basic deployment of Python programs. Some of these topics will be relied on directly by the material in this book. Perhaps more importantly, you may want to apply these skills to the code you write as when following this book. Some of the topics we cover can be complex and a bit tricky, so knowing how to test and debug your code as you learn might be very useful.

Well begun is half done

Python is becoming more popular every day, and it's being applied in all sort of domains and applications. One of Python's strengths is that it's approachable and easy to learn, so that almost anyone can learn to write a basic Python program. *The Python Journeyman* will take you beyond that, beyond the basics. We want to teach you some of the deeper aspects of Python, and give you the skills you need to write *great* Python programs.

Chapter 1 - Organizing Larger Programs

In this chapter we'll be covering more of Python's techniques for organizing programs. Specifically we'll be looking at Python's concept of *packages* and how these can be used to add structure to your program as it grows beyond simple modules.

Packages

As you'll recall, Python's basic tool for organizing code is the *module*.[8] A module typically corresponds to a single source file, and you load modules into programs by using the `import` keyword. When you import a module, it is represented by an object of type `module` and you can interact with it like any other object.

A *package* in Python is just a special type of module. The defining characteristic of a package is that it can contain other modules, including other packages. So packages are a way to define hierarchies of modules in Python. This allows you to group modules with similar functionality together in ways that express their cohesiveness.

An example of a package: `urllib`

Many parts of Python's standard library are implemented as packages. To see an example, open your REPL and import `urllib` and `urllib.request`:

```
>>> import urllib
>>> import urllib.request
```

Now if you check the types of both of the these modules, you'll see that they're both of type `module`:

[8] See Chapter 3 in The Python Apprentice

```
>>> type(urllib)
<class 'module'>
>>> type(urllib.request)
<class 'module'>
```

The important point here is that `urllib.request` is *nested* inside `urllib`. In this case, `urllib` is a package and `request` is a normal module.

The `__path__` attribute of packages

If you closely inspect each of these objects, you'll notice an important difference. The `urllib` package has a `__path__` member that `urllib.request` does not have:

```
>>> urllib.__path__
['./urllib']
>>> urllib.request.__path__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute '__path__'
```

This attribute is a list of filesystem paths indicating where `urllib` searches to find nested modules. This hints at the nature of the distinction between packages and modules: packages are generally represented by directories in the filesystem while modules are represented by single files.

Note that in Python 3 versions prior to 3.3, `__path__` was just a single string, not a list. In this book we're focusing on Python 3.5+, but for most purposes the difference is not important.

Locating modules

Before we get in to the details of packages it's important to understand how Python locates modules. Generally speaking, when you ask Python to `import` a module, Python looks on your filesystem for the corresponding Python source file and loads that code. But how does Python know where to look? The answer is that Python checks the `path` attribute of the standard `sys` module, commonly referred to as `sys.path`.

The `sys.path` object is a list of directories. When you ask Python to import a module, it starts with the first directory in `sys.path` and checks for an appropriate file. If no match is found in the first directory it checks subsequent entries, in order, until a match is found or Python runs out of entries in `sys.path`, in which case an `ImportError` is raised.

sys.path

Let's explore `sys.path` from the REPL. Run Python from the command line with no arguments and enter the following statements:

```
>>> import sys
>>> sys.path
['', '/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/r\
ope_py3k-0.9.4-py3.5.egg', '/Library/Frameworks/Python.framework/Versions/3.5/lib/pyt\
hon3.5/site-packages/decorator-3.4.0-py3.5.egg', '/Library/Frameworks/Python.framewor\
k/Versions/3.5/lib/python3.5/site-packages/Baker-1.3-py3.5.egg', '/Library/Frameworks\
/Python.framework/Versions/3.5/lib/python3.5/site-packages/beautifulsoup4-4.1.3-py3.5\
.egg', '/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages\
/pymongo-2.3-py3.5-macosx-10.6-intel.egg', '/Library/Frameworks/Python.framework/Vers\
ions/3.5/lib/python3.5/site-packages/eagertools-0.3-py3.5.egg', '/home/projects/emacs\
_config/traad/traad', '/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.\
5/site-packages/bottle-0.11.6-py3.5.egg', '/home/projects/see_stats', '/Library/Frame\
works/Python.framework/Versions/3.5/lib/python3.5/site-packages/waitress-0.8.5-py3.5.\
egg', '/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/\
pystache-0.5.3-py3.5.egg', '/Library/Frameworks/Python.framework/Versions/3.5/lib/pyt\
hon3.5/site-packages/pyramid_tm-0.7-py3.5.egg', '/Library/Frameworks/Python.framework\
/Versions/3.5/lib/python3.5/site-packages/pyramid_debugtoolbar-1.0.6-py3.5.egg', '/Li\
brary/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/pyramid-1.\
4.3-py3.5.egg', '/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site\
-packages/transaction-1.4.1-py3.5.egg', '/Library/Frameworks/Python.framework/Version\
s/3.5/lib/python3.5/site-packages/Pygments-1.6-py3.5.egg', '/Library/Frameworks/Pytho\
n.framework/Versions/3.5/lib/python3.5/site-packages/PasteDeploy-1.5.0-py3.5.egg', '/\
Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/translat\
ionstring-1.1-py3.5.egg', '/Library/Frameworks/Python.framework/Versions/3.5/lib/pyth\
on3.5/site-packages/venusian-1.0a8-py3.5.egg', '/Library/Frameworks/Python.framework/\
Versions/3.5/lib/python3.5/site-packages/zope.deprecation-4.0.2-py3.5.egg', '/Library\
/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/zope.interface-\
4.0.5-py3.5-macosx-10.6-intel.egg', '/Library/Frameworks/Python.framework/Versions/3.\
5/lib/python3.5/site-packages/repoze.lru-0.6-py3.5.egg', '/Library/Frameworks/Python.\
framework/Versions/3.5/lib/python3.5/site-packages/WebOb-1.2.3-py3.5.egg', '/Library/\
Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/Mako-0.8.1-py3.5\
.egg', '/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages\
/Chameleon-2.11-py3.5.egg', '/Library/Frameworks/Python.framework/Versions/3.5/lib/py\
thon3.5/site-packages/MarkupSafe-0.18-py3.5-macosx-10.6-intel.egg', '/Library/Framewo\
rks/Python.framework/Versions/3.5/lib/python3.5/site-packages/pip-1.4.1-py3.5.egg', '\
/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/ipython\
-1.0.0-py3.5.egg', '/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/s\
ite-packages/pandas-0.12.0-py3.5-macosx-10.6-intel.egg', '/Library/Frameworks/Python.\
framework/Versions/3.5/lib/python3.5/site-packages/setuptools-1.1.6-py3.5.egg', '/Lib\
rary/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/readline-6.\
2.4.1-py3.5-macosx-10.6-intel.egg', '/home/projects/see_stats/distribute-0.6.49-py3.5\
.egg', '/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages\
/nltk-2.0.4-py3.5.egg', '/Library/Frameworks/Python.framework/Versions/3.5/lib/python\
3.5/site-packages/PyYAML-3.10-py3.5-macosx-10.6-intel.egg', '/Library/Frameworks/Pyth\
on.framework/Versions/3.5/lib/python3.5/site-packages/numpy-1.8.0-py3.5-macosx-10.6-i\
ntel.egg', '/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-pack\
ages/grin-1.2.1-py3.5.egg', '/Library/Frameworks/Python.framework/Versions/3.5/lib/py\
thon3.5/site-packages/argparse-1.2.1-py3.5.egg', '/Library/Frameworks/Python.framewor\
k/Versions/3.5/lib/python33.zip', '/Library/Frameworks/Python.framework/Versions/3.5/\
lib/python3.5', '/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/plat\
-darwin', '/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/lib-dynloa\
d', '/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages']
```

As you see, your `sys.path` can be quite large. It's precise entries depend on a number of factors, including how many third-party packages you've installed and how you've installed them. For our purposes, a few of these entries are of particular importance. First, let's look at the very first entry:

```
>>> sys.path[0]
''
```

Remember that `sys.path` is just a normal list so we can examine it's contents with indexing and slicing. We see here that the first entry is the empty string. This happens when you start the Python interpreter with no arguments, and it instructs Python to search for modules first in the current directory.

Let's also look at the tail of `sys.path`:

```
>>> sys.path[-5:]
['/Library/Frameworks/Python.framework/Versions/3.5/lib/python33.zip', '/Library/Fram\
eworks/Python.framework/Versions/3.5/lib/python3.5', '/Library/Frameworks/Python.fram\
ework/Versions/3.5/lib/python3.5/plat-darwin', '/Library/Frameworks/Python.framework/\
Versions/3.5/lib/python3.5/lib-dynload', '/Library/Frameworks/Python.framework/Versio\
ns/3.5/lib/python3.5/site-packages']
```

These entries comprise Python's standard library and the `site-packages` directory where you can install third-party modules.

`sys.path` in action

To really get a feel for `sys.path`, let's create a Python source file in a directory that Python would not normally search:

```
$ mkdir not_searched
```

In that directory create a file called `path_test.py` with the following function definition:

```
# not_searched/path_test.py

def found():
    return "Python found me!"
```

Now, start your REPL from the *directory containing the `not_searched` directory* and try to import `path_test`:

```
>>> import path_test
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'path_test'
```

The `path_test` module — which, remember, is embodied in `not_searched/path_test.py` — is not found because `path_test.py` is not in a directory contained in `sys.path`. To make `path_test` importable we need to put the directory `not_searched` into `sys.path`.

Since `sys.path` is just a normal list, we can add our new entry using the `append()` method. Start a new REPL and enter this:

```
>>> import sys
>>> sys.path.append('not_searched')
```

Now when we try to import `path_test` in the same REPL session we see that it works:

```
>>> import path_test
>>> path_test.found()
Python found me!
```

Knowing how to manually manipulate `sys.path` can be useful, and sometimes it's the best way to make code available to Python. There is another way to add entries to `sys.path`, though, that doesn't require direct manipulation of the list.

The `PYTHONPATH` environment variable

The `PYTHONPATH` environment variable is a list of paths that are added to `sys.path` when Python starts.

The format of `PYTHONPATH` is the same as `PATH` on your platform. On Windows `PYTHONPATH` is a semicolon-separated list of directories:

```
c:\some\path;c:\another\path;d:\yet\another
```

On Linux and OS X it's a colon-separated list of directories:

```
/some/path:/another/path:/yet/another
```

To see how `PYTHONPATH` works, let's add `not_searched` to it before starting Python again. On Windows use the `set` command:

```
> set PYTHONPATH=not_searched
```

On Linux or OS X the syntax will depend on your shell, but for bash-like shells you can use `export`:

```
$ export PYTHONPATH=not_searched
```

Now start a new REPL and check that `not_searched` is indeed in `sys.path`:

```
>>> [path for path in sys.path if 'not_searched' in path]
['/home/python_journeyman/not_searched']
```

And of course we can now import `path_test` without manually editing `sys.path`:

```
>>> import path_test
>>> path_test.found()
Python found me!
```

There are more details to `sys.path` and `PYTHONPATH`, but this is most of what you need to know. For more information you can check these links to the Python documentation:

- `PYTHONPATH` documentation[9]
- `sys.path` documentation[10]

Implementing packages

We've seen that packages are modules that can contain other modules. But how are packages implemented? To create a normal module you simply create a Python source file in a directory contained in `sys.path`. The process for creating packages is not much different.

To create a package, first you create the package's *root directory*. This root directory needs to be in some directory on `sys.path`; remember, this is how Python finds modules and packages for importing. Then, in that root directory, you create a file called `__init__.py`. This file — which we'll often call the *package init* file — is what makes the package a module. `__init__.py` can be (and often is) empty; its presence alone suffices to establish the package.

> In Namespace packages we'll look at a somewhat more general form of packages which can span multiple directory trees. In that section we'll see that, since PEP 420[a] was introduced in Python 3.3, `__init__.py` files are not technically required for packages any more.
>
> So why do we do we talk about them as if they are required? For one thing, they are still required for earlier versions of Python. In fact, many people writing code for 3.3+ aren't aware that `__init__.py` files are optional. As a result, you'll still find them in the vast majority of packages, so it's good to be familiar with them.

[9] http://docs.python.org/3/using/cmdline.html#envvar-PYTHONPATH

[10] http://docs.python.org/3/library/sys.html#sys.path

Furthermore, they provide a powerful tool for package initialization. So it's important to understand how they can be used.

Perhaps most importantly, though, we recommend that you include `__init__.py` files when possible because "explicit is better than implicit". The existence of a package initialization file is an unambiguous signal that you intend for a directory to be a package, and it's something that many Python developers instinctively look for.

[a] https://www.python.org/dev/peps/pep-0420/

A first package: `reader`

As with many things in Python, an example is much more instructive than words. Go your command prompt and create a new directory called 'reader':

```
$ mkdir reader
```

Add an empty `__init__.py` to this directory. On Linux os OS X you can use the touch command:

```
$ touch reader/__init__.py
```

On Windows you can use `type`:

```
> type NUL > reader/__init__.py
```

Now if you start your REPL you'll see that you can import `reader`:

```
>>> import reader
>>> type(reader)
<class 'module'>
```

The role of `__init__.py`

We can now start to examine the role that `__init__.py` plays in the functioning of a package. Check the `__file__` attribute of the `reader` package:

```
>>> reader.__file__
'./reader/__init__.py'
```

We saw that `reader` is a module, even though on our filesystem the name "reader" refers to a directory. Furthermore, the source file that is executed when `reader` is imported is the package init file in the `reader` directory, a.k.a `reader/__init__.py`. In other words — and to reiterate a critical point — a package is nothing more than a directory containing a file named `__init__.py`.

To see that the `__init__.py` is actually executed like any other module when you import `reader`, let's add a small bit of code to it:

```
# reader/__init__.py
print('reader is being imported!')
```

Restart your REPL, import `reader`, and you'll see our little message printed out:

```
>>> import reader
reader is being imported!
```

Adding functionality to the package

Now that we've created a basic package, let's add some useful content to it. The goal of our package will be to create a class that can read data three different file formats: uncompressed text files, text files compressed with gzip[11], and text files compressed with bz2[12].[13] We'll call this class `MultiReader` since it can read multiple formats.

We'll start by defining `MultiReader`. The initial version the class will only know how to read uncompressed text files; we'll add support for gzip and bz2 later. Create the file `reader/multireader.py` with these contents:

[11] http://www.gzip.org/

[12] http://www.bzip.org/

[13] In the spirit of "batteries included", the Python standard library include support for reading both gzip and bz2.

```
# reader/multireader.py


class MultiReader:
    def __init__(self, filename):
        self.filename = filename
        self.f = open(filename, 'rt')

    def close(self):
        self.f.close()

    def read(self):
        return self.f.read()
```

Start a new REPL and import your new module to try out the `MultiReader` class. Let's use it to read the contents of `reader/__init__.py` itself:

```
>>> import reader.multireader
>>> r = reader.multireader.MultiReader('reader/__init__.py')
>>> r.read()
"# reader/__init__.py\n"
>>> r.close()
```

In a somewhat "meta" turn, our package is reading some of its own source!

Subpackages

To demonstrate how packages provide high levels of structure to your Python code let's add more layers of packages to the `reader` hierarchy. We're going to add a subpackage to `reader` called `compressed` which will contain the code for working with compressed files. First, let's create the new directory and its associated `__init__.py` On Linux or OS X use this:

```
$ mkdir reader/compressed
$ touch reader/compressed/__init__.py
```

On Windows use these commands:

```
> mkdir reader\compressed
> type NUL > reader\compressed\__init__.py
```

If you restart your REPL you'll see that you can import `reader.compressed`:

```
>>> import reader.compressed
>>> reader.compressed.__file__
'reader/compressed/__init__.py'
```

gzip support

Next we'll create the file `reader/compressed/gzipped.py` which will contain some code for working with the gzip compression format:

```
# reader/compressed/gzipped.py

import gzip
import sys

opener = gzip.open

if __name__ == '__main__':
    f = gzip.open(sys.argv[1], mode='wt')
    f.write(' '.join(sys.argv[2:]))
    f.close()
```

As you can see, there's not much to this code: it simply defines the name `opener` which is just an alias for `gzip.open()`. This function behaves much like the normal `open()` in that it returns a file-like-object[14] which can be read from. The main difference, of course, is that `gzip.open()` decompresses the contents of the file during reading while `open()` does not.[15]

Note the idiomatic "main block" here. It uses `gzip` to create a new compressed file and write data to it. We'll use that later to create some test files. For more information on

[14] See chapter 9 of The Python Apprentice for background on files and file-like-objects.

[15] A subpackage may seem like an unnecessary amount of ceremony for something as simple as this. In production code that would likely be true, but for teaching purposes it's helpful to have simple code.

`__name__` and `__main__`, see chapter 3 of The Python Apprentice[a].

[a] https://leanpub.com/python-apprentice/

bz2 support

Similarly, let's create another file for handling bz2 compression called `reader/compressed/bzipped.py`

```
# reader/compressed/bzipped.py

import bz2
import sys

opener = bz2.open

if __name__ == '__main__':
    f = bz2.open(sys.argv[1], mode='wt')
    f.write(' '.join(sys.argv[2:]))
    f.close()
```

At this point you should have a directory structure that looks like this:

```
reader
├── __init__.py
├── multireader.py
└── compressed
    ├── __init__.py
    ├── bzipped.py
    └── gzipped.py
```

If you start a new REPL, you'll see that we can import all of our modules just as you might expect:

```
>>> import reader
>>> import reader.multireader
>>> import reader.compressed
>>> import reader.compressed.gzipped
>>> import reader.compressed.bzipped
```

A full program

Let's glue all of this together into a more useful sort of program. We'll update `MultiReader` so that it can read from gzip files, bz2 files, and normal text files. It will determine which format to use based on file extensions. Change your `MultiReader` class in `reader/multireader.py` to use the compression handlers when necessary:

```
 1  # reader/multireader.py
 2
 3  import os
 4
 5  from reader.compressed import bzipped, gzipped
 6
 7  # This maps file extensions to corresponding open methods.
 8  extension_map = {
 9      '.bz2': bzipped.opener,
10      '.gz': gzipped.opener,
11  }
12
13
14  class MultiReader:
15      def __init__(self, filename):
16          extension = os.path.splitext(filename)[1]
17          opener = extension_map.get(extension, open)
18          self.f = opener(filename, 'rt')
19
20      def close(self):
21          self.f.close()
22
23      def read(self):
24          return self.f.read()
```

The most interesting part of this change is line 5 where we import the subpackages `bzipped` and `gzipped`. This demonstrates the fundamental organizing power of packages: related functionality can be grouped under a common name for easier identification.

With these changes, MultiReader will now check the extension of the filename it's handed. If that extension is a key in extension_map then a specialized file-opener will be used — in this case, either reader.compressed.gzipped.opener or reader.compressed.gzipped.opener. Otherwise, the standard open() will be used.

To test this out, let's first create some compressed files using the utility code we built into our compression modules. Execute the modules directly from the command line:

```
$ python3 -m reader.compressed.bzipped test.bz2 data compressed with bz2
$ python3 -m reader.compressed.gzipped test.gz data compressed with gzip
$ ls
reader                test.bz2        test.gz
```

Feel free to verify for yourself that the contents of test.bz2 and test.gz are actually compressed (or at least that they're not plain text!)

Start a new REPL and let's take our code for a spin:

```
>>> from reader.multireader import MultiReader
>>> r = MultiReader('test.bz2')
>>> r.read()
'data compressed with bz2'
>>> r.close()
>>> r = MultiReader('test.gz')
>>> r.read()
'data compressed with gzip'
>>> r.close()
>>> r = MultiReader('reader/__init__.py')
>>> r.read()
. . . the contents of reader/__init__.py . . .
>>> r.close()
```

If you've put all the right code in all the right places, you should see that your MultiReader can indeed decompress gzip and bz2 files when it sees their associated file extensions.

Package review

We've covered a lot of information in this chapter already, so let's review.

1. Packages are modules which can contain other modules.

2. Packages are generally implemented as directories containing a special `__init__.py` file.
3. The `__init__.py` file is executed when the package is imported.
4. Packages can contain subpackages which are themselves implemented as directories containing `__init__.py` files.
5. The `module` objects for packages have a `__path__` attribute.

Relative imports

In this book we've seen a number of uses of the `import` keyword, and if you've done any amount of Python programming then you should be familiar with it. All of the uses we've seen so far are what are called *absolute imports* wherein you specify all of the ancestor modules of any module you want to import. For example, in order to import `reader.compressed.bzipped` in the previous section you had to mention both `reader` and `compressed` in the `import` statement:

```
# Both of these absolute imports mention both `reader` and `compressed`
import reader.compressed.bzipped
from reader.compressed import bzipped
```

There is an alternative form of imports called *relative imports* that lets you use shortened paths to modules and packages. Relative imports look like this:

```
from ..module_name import name
```

The obvious difference between this form of `import` and the absolute imports we've seen so far is the `.`s before `module_name`. In short, each `.` stands for an ancestor package of the module that is doing the import, starting with the package containing the module and moving towards the package root. Instead of specifying imports with absolute paths from the root of the package tree, you can specify them *relative* to the importing module, hence *relative imports*.

> Note that you can only use relative imports with the "`from <module> import <names>`" form of `import`. Trying to do something like "`import .module`" is a syntax error.

Critically, relative imports can only be used within the current top-level package, never for importing modules outside of that package. So in our previous example the `reader` module could use a relative import for `gzipped`, but it needs to use absolute imports for anything outside of the top-level `reader` package.

Let's illustrate this with some simple examples. Suppose we have this package structure:

```
graphics/
  __init__.py
  primitives/
    __init__.py
    line.py
  shapes/
    __init__.py
    triangle.py
  scenes/
    __init__.py
```

Further suppose that the module `line.py` module includes the following definition:

```
# graphics/primitives/line.py

def render(x, y):
   "draw a line from x to y"
   # . . .
```

The `graphics.shapes.triangle` module is responsible for rendering — you guessed it! – triangles, and to do this it needs to use `graphics.primitives.lines.render()` to draw lines. One way `triangle.py` could import this function is with an absolute import:

```
# graphics/shapes/triangle.py

from graphics.primitives.line import render
```

Alternatively you could use a relative import:

```
# graphics/shapes/triangle.py

from ..primitives.line import render
```

The leading `..` in the relative form means “the parent of the package containing this module”, or, in other words, the `graphics` package.

This table summarizes how the `.`s are interpreted when used to make relative imports from `graphics.shapes.triangle`:

triangle.py	module
`.`	`graphics.shapes`
`..`	`graphics`
`..primitives`	`graphics.shapes.primitives`

Bare dots in the `from` clause

In relative imports it’s also legal for the `from` section to consist purely of dots. In this case the dots are still interpreted in exactly the same way as before.

Going back to our example, suppose that the `graphics.scenes` package is used to build complex, multi-shape scenes. To do this it needs to use the contents of `graphics.shapes`, so `graphics/scenes/__init__.py` needs to import the `graphics.shapes` package. It could do this in a number of ways using absolute imports:

```
# graphics/scenes/__init__.py

# All of these import the same module in different ways
import graphics.shapes
import graphics.shapes as shapes
from graphics import shapes
```

Alternatively, `graphics.scenes` could use relative imports to get at the same module:

```
# graphics/scenes/__init__.py

from .. import shapes
```

Here the `..` means "the parent of the current package", just as with the first form of relative imports.

It's easy to see how relative imports can be useful for reducing typing in deeply nested package structures. They also promote certain forms of modifiability since they allow you, in principle, to rename top-level and sub-packages in some cases. On the whole, though, the general consensus seems to be that relative imports are best avoided in most cases.

`__all__`

Another topic we want to look at is the optional `__all__` attribute of modules. `__all__` lets you control which attributes are imported when someone uses the `from module import *` syntax. If `__all__` is not specified then `from x import *` imports all public[16] names from the imported module.

The `__all__` module attribute must be a list of strings, and each string indicates a name which will be imported when the `*` syntax is used. For example, we can see what `from reader.compressed import *` does. First, let's add some code to `reader/compressed/__init__.py`:

```
# reader/compressed/__init__.py

from reader.compressed.bzipped import opener as bz2_opener
from reader.compressed.gzipped import opener as gzip_opener
```

Next we'll start a REPL and display all the names currently in scope:

```
>>> locals()
{'__loader__': <class '_frozen_importlib.BuiltinImporter'>,
 '__name__': '__main__',
 '__builtins__': <module 'builtins' (built-in)>,
 '__package__': None,
 '__doc__': None}
```

Now we'll import all public names from `compressed`:

[16] It turns out that leading underscores in Python are a bit more than just a convention to indicate implementation details for the benefit of humans. Only module scope (*i.e.* global) attributes without any leading underscores will be imported, unless they are explicitly named.

```
>>> from reader.compressed import *
>>> locals()
{'bz2_opener': <function open at 0x1018300e0>,
 'gzip_opener': <function open at 0x101a36320>,
 'gzipped': <module 'reader.compressed.gzipped' from './reader/compressed/gzipped.py'\
>, 'bzipped': <module 'reader.compressed.bzipped' from './reader/compressed/bzipped.p\
y'>, '__package__': None,
 '__name__': '__main__',
 '__builtins__': <module 'builtins' (built-in)>,
 '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
 '__doc__': None}
>>> bzipped
<module 'reader.compressed.bzipped' from './reader/compressed/bzipped.py'>
>>> gzipped
<module 'reader.compressed.gzipped' from './reader/compressed/gzipped.py'>
```

What we see is that `from reader.compressed import *` imported the `bzipped` and `gzipped` submodules of the `compressed` package directly into our local namespace. We prefer that `import *` only imports the different "opener" functions from each of these modules, so let's update `compressed/__init__.py` to do that:

```
# reader/compressed/__init__.python

from reader.compressed.bzipped import opener as bz2_opener
from reader.compressed.gzipped import opener as gzip_opener

__all__ = ['bz2_opener', 'gzip_opener']
```

Now if we use `import *` on `reader.compressed` we only import the "opener" functions, not their modules as well:

```
>>> locals()
{'__package__': None, '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__d\
oc__': None, '__builtins__': <module 'builtins' (built-in)>, '__name__': '__main__'}
>>> from reader.compressed import *
>>> locals()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '\
_frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {}, '__buil\
tins__': <module 'builtins' (built-in)>, 'bz2_opener': <function open at 0x10efb9378>\
, 'gzip_opener': <function open at 0x10f06b620>}
>>> bz2_opener
<function open at 0x1022300e0>
>>> gzip_opener
<function open at 0x10230a320>
```

The `__all__` module attribute can be a useful tool for limiting which names are exposed by your modules. We still don't recommend that you use the `import *` syntax outside of convenience in the REPL, but it's good to know about `__all__` since you're likely to see it in the wild.

Namespace packages

Earlier we said that packages are implemented as directories containing a `__init__.py` file. This is true for most cases, but there are situations where you want to be able to split a package across multiple directories. This is useful, for example, when a logical package needs to be delivered in multiple parts, as happens in some of the larger Python projects.

Several approaches to addressing this need have been implemented, but it was in PEP420[17] in 2012 that an official solution was built into the Python language. This solution is known as *namespace packages*. A namespace package is a package which is spread over several directories, with each directory tree contributing to a single logical package from the programmer's point of view.

Namespace packages are different from normal packages in that they don't have `__init__.py` files. This is important because it means that namespace packages can't have package-level initialization code; nothing will be executed by the package when it's imported. The reason for this limitation is primarily that it avoids complex questions of initialization order when multiple directories contribute to a package.

[17] https://www.python.org/dev/peps/pep-0420/

But if namespace packages don't have `__init__.py` files, how does Python find them during import? The answer is that Python follows a relatively simple algorithm[18] to detect namespace packages. When asked to import the name "foo", Python scans each of the entries in `sys.path` in order. If in any of these directories it finds a directory named "foo" containing `__init__.py`, then a normal package is imported. If it doesn't find any normal packages but it *does* find `foo.py` or any other file that can act as a module[19], then this module is imported instead. Otherwise, the import mechanism keeps track of any directories it finds which are named "foo". If no normal packages or modules are found which satisfy the import, then all of the matching directory names act as parts of a namespace package.

An example of a namespace package

As a simple example, let's see how we might turn the `graphics` package into a namespace package. Instead of putting all of the code under a single directory, we would have two independent parts rooted at `path1` and `path2` like this:

```
path1
└── graphics
    ├── primitives
    │   ├── __init__.py
    │   └── line.py
    └── shapes
        ├── __init__.py
        └── triangle.py
path2
└── graphics
    └── scenes
        └── __init__.py
```

This separates the `scenes` package from the rest of the package.

Now to import `graphics` you need to make sure that both `path1` and `path2` are in your `sys.path`. We can do that in a REPL like this:

[18] https://www.python.org/dev/peps/pep-0420/#specification

[19] Python has a flexible import system, and you are not limited to just importing from files. For example, PEP 273 describes how to Python allows you to import modules from zip-files. We cover this in detail in Chapter 13 of "The Python Master".

```
>>> import sys
>>> sys.path.extend(['./path1', './path2'])
>>> import graphics
>>> graphics.__path__
_NamespacePath(['./path1/graphics', './path2/graphics'])
>>> import graphics.primitives
>>> import graphics.scenes
>>> graphics.primitives.__path__
['./path1/graphics/primitives']
>>> graphics.scenes.__path__
['./path2/graphics/scenes']
```

We put `path1` and `path2` at the end of `sys.path`. When we import `graphics` we see that its `__path__` includes portions from both `path1` and `path2`. And when we import the `primitives` and `scenes` we see that they are indeed coming from their respective directories.

There are more details to *namespace packages*, but this addresses most of the important details that you'll need to know. In fact, for the most part it's not likely that you'll need to develop your own namespace packages at all. If you do want to learn more about them, though, you can start by reading PEP 420[20].

Executable directories

Packages are often developed because they implement some program that you want to execute. There are a number of ways to construct such programs, but one of the simplest is through the use of *executable directories*. Executable directories let you specify a main entry point which is run when the directory is executed by Python.

What do we mean when we say that Python "executes a directory"? We mean passing a directory name to Python on the command line like this:

```
$ mkdir text_stats
$ python3 text_stats
```

Normally this doesn't work and Python will complain saying that it can't find a `__main__` module:

[20] http://www.python.org/dev/peps/pep-0420/

```
$ python3 text_stats
/usr/local/bin/python3: can't find '__main__' module in 'text_stats'
```

However, as that error message suggests, you can put a special module named `__main__.py` in the directory and Python will execute it. This module can execute whatever code it wants, meaning that it can call into modules you've created to provide, for example, a user interface to your modules.

To illustrate this, let's add a `__main__.py` to our `text_stats` directory. This program will count (crudely!) the number of words and characters passed in on the command line:

```
# text_stats/__main__.py

import sys

segments = sys.argv[1:]

full_text = ' '.join(segments)

output = '# words: {}, # chars: {}'.format(
    len(full_text.split()),
    sum(len(w) for w in full_text.split()))

print(output)
```

Now if we pass this `text_stats` directory to Python we will see our `__main__.py` executed:

```
$ python text_stats I'm seated in an office, surrounded by heads and bodies.
# words: 10, # chars: 47
```

This is interesting, but used in this way `__main__.py` is not much more than a curiosity. As we'll soon see, however, this idea of an "executable directory" can be used to better organize code that might otherwise sprawl inside a single file.

`__main__.py` and `sys.path`

When Python executes a `__main__.py`, it first adds the directory containing `__main__.py` to `sys.path`. This way `__main__.py` can easily import any other modules with which it shares a directory.

If you think of the directory containing `__main__.py` as a program, then you can see how this change to `sys.path` allows you to organize your code in better ways. You can use separate modules for the logically distinct parts of your program. In the case of our `text_stats` example it makes sense to move the actual counting logic into a separate module, so let's put that code into a module called `counter`:

```
# text_stats/counter.py

def count(text):
    words = text.split()
    return (len(words), sum(len(w) for w in words))
```

We'll then update our `__main__.py` to use `counter`:

```
# text_stats/__main__.py

import sys

import counter

segments = sys.argv[1:]
full_text = ' '.join(segments)
output = '# words: {}, # chars: {}'.format(
    *counter.count(full_text))
print(output)
```

Now the logic for counting words and letters is cleanly separated from the UI logic in our little program. If we run our directory again we see that it still works:

```
$ python3 text_stats It is possible I already had some presentiment of my future.
# words: 11, # chars: 50
```

Zipping up executable directories

We can take the executable directory idea one step further by zipping the directory. Python knows how to read zip-files and treat them like directories, meaning that we can create executable zip-files just like we created executable directories.

Create a zip-file from your `text_stats` directory:

```
$ cd text_stats
$ zip -r ../text_stats.zip *
$ cd ..
```

> The zip-file should contain the *contents* of your executable directory but not the executable directory itself. The zip-file takes the place of the directory.

Now we can tell Python to execute the zip-file rather than the directory:

```
$ python3 text_stats.zip Sing, goddess, the anger of Peleus’ son Achilleus
# words: 8, # chars: 42
```

Combining Python’s support for `__main__.py` with its ability to execute zip-files gives us a convenient way to distribute code in some cases. If you develop a program consisting of a directory containing some modules and a `__main__.py`, you can zip up the contents of the directory, share it with others, and they’ll be able to run it with no need for installing any packages to their Python installation.

Of course, sometimes you really do need to distribute proper packages rather than more *ad hoc* collections of modules, so we’ll look at the role of `__main__.py` in packages next.

Executable packages

In the previous section we saw how to use `__main__.py` to make a directory directly executable. You can use a similar technique to create *executable packages*. If you put `__main__.py` in a package directory, then Python will execute it when you run the package with `python3 -m package_name`.

To demonstrate this, let’s convert our `text_stats` program into a package. Create an empty `__init__.py` in the `text_stats` directory. Then edit `text_stats/__main__.py` to import `counter` as a relative import:

```
# text_stats/__main__.py

import sys

from .counter import count

segments = sys.argv[1:]
full_text = ' '.join(segments)
output = '# words: {}, # chars: {}'.format(
    *count(full_text))
print(output)
```

After these changes we see that we can no longer execute the directory with `python3 text_stats` like before:

```
$ python3 text_stats horror vacui
Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.6/runpy.py", li\
ne 193, in _run_module_as_main
    "__main__", mod_spec)
  File "/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.6/runpy.py", li\
ne 85, in _run_code
    exec(code, run_globals)
  File "text_stats/__main__.py", line 5, in <module>
    from .counter import count
ImportError: attempted relative import with no known parent package
```

Python complains that `__main__.py` can't import `counter` with a relative import. This seems to be at odds with our design: `__main__.py` and `counter.py` are clearly in the same package!

The reason this fails is because of what we learned earlier about executable directories. When we ask Python to execute the `text_stats` directory it first adds the `text_stats` directory to `sys.path`. It then executes `__main__.py`. The crucial detail is that `sys.path` contains `text_stats` itself, *not the directory containing `text_stats`*. As a result, Python doesn't recognize `text_stats` as a package at all; we haven't told it the right place to look.

In order to execute our package we need to tell Python to treat `text_stats` as a module[21] with the `-m` command line flag:

[21] Remember that packages are just a particular kind of module.

```
$ python3 -m text_stats fiat lux
# words: 2, # chars: 7
```

Now Python looks for the module `text_stats.__main__` — that is, our `text_stats/__main__.py` — and executes it while treating `text_stats` as a package. As a result, `__main__.py` is able to use a relative import to pull in `counter`.

The difference between `__init__.py` and `__main__.py`

As you'll recall, Python executes `__init__.py` the first time a package is imported. So you may be wondering why we need `__main__.py` at all. After all, can't we just execute the same code in `__init__.py` as we do in `__main__.py`?

The short answer is "no". You can, of course, put whatever code you want in `__init__.py`, but Python will not execute a package unless it contains `__main__.py`. To see this, first move `text_stats/__main__.py` out of `text_stats` and try running the package again:

```
$ python3 -m text_stats sleeping giant
/usr/local/bin/python3.6: No module named text_stats.__main__; 'text_stats' is a pack\
age and cannot be directly executed
```

Now move `__main__.py` back into place and edit `text_stats/__init__.py` to print a little message:

```
# text_stats/__init__.py

print("executing {}.__init__".format(__name__))
```

Execute the package again:

```
$ python3 -m text_stats sleeping giant
executing text_stats.__init__
# words: 2, # chars: 13
```

We can see that our package's `__init__.py` is indeed executed when it's imported, but, again, Python will not let us execute a package unless it contains `__main__.py`.

Recommended layout

As we close out this chapter, let's look at how you can best structure your projects. There are no hard-and-fast rules about how you lay out your code, but some options are generally better than others. What we'll present here is a good, general-purpose structure that will work for almost any project you might work on.

Here's the basic project layout:

```
project_name/
  setup.py
  project_name/
    __init__.py
    more_source.py
    subpackage1/
      __init__.py
  test/
    test_code.py
```

At the very top level you have a directory with the project's name. This directory is *not* a package but is a directory containing both your package as well as supporting files like your `setup.py`, license details, and your test directory.

The next directory down is your actual package directory. This has the same name as your top-level directory. Again, there's no rule that says that this must be so, but this is a common pattern and makes it easy to recognize where you are when navigating your project. Your package contains all of the production code including any subpackages.

Separation between package and test

The `test` directory contains all of your tests. This may be as simple as a few Python files or as complex as multiple suites of unit, integration, and end-to-end tests. We recommend keeping your tests outside of your package for a number of reasons.

Generally speaking, test and production code serve very different purposes and shouldn't be coupled unnecessarily. Since you usually don't need your tests to be installed along with your package, this keeps packaging tools from bundling them together. Also, more exotically, this arrangement ensures that certain tools won't accidentally try to treat your tests as production code.[22]

As with all things, this test directory arrangement may not suit your needs. Certainly you will find examples of packages that include their tests as subpackages.[23] If you find that you need to include all or some of your tests in a subpackage, you absolutely should.

A pragmatic starting point

There's not much more to it than that. This is a very simple structure, but it works well for most needs. It serves as a fine starting point for more complex project structures, and this is the structure we typically use when starting new projects.

Modules are singletons

The singleton pattern is one of the most widely-known patterns in software development, in large part because it's very simple and, in some ways, provides a superior option to the dreaded global variable. The intention of the singleton pattern is to limit the number of instances of an type to only one. For example, a single registry of items easily accessible from throughout the application.[24]

If you find that you need a singleton in Python, one simple and effective way to implement it is as a module-level attribute. Since modules are only ever executed once, this guarantees that your singleton will only be initialized once. And since modules are initialized in a well-defined user-controlled order, you can have strong guarantees about *when* your singleton will be initialized. This forms a strong basis for implementing practical singletons.

Consider a simple singleton registry, implemented in `registry.py`, where callers can leave their names:

[22] For example, the mutation testing tool Cosmic Ray is designed to apply mutations to Python modules and then run tests against them. If your test code is part of your package, Cosmic Ray will try to mutate your tests, resulting in very strange behavior indeed!

[23] In fact, in the past we've recommended including tests as subpackages rather than outside the package. Our experience since then has convinced us otherwise.

[24] The singleton pattern has its detractors, and for good reason. Singletons are often misused in a way that introduces unnecessary restrictions in situation where in multiple instances of a class could be useful. Be aware that singletons introduce shared global state into your program.

```
# registry.py

_registry = []

def register(name):
    _registry.append(name)

def registered_names():
    return iter(_registry)
```

Callers would use it like this:

```
import registry

registry.register('my name')

for name in registry.registered_names():
    print(name)
```

The first time `registry` is imported, the `_registry` list is initialized. Then, every call to `register` and `registered_names` will access that list with complete assurance that it has been properly initialized.

> You will recall that the leading underscore in `_registry` is a Python idiom indicating that `_registry` is an implementation detail that should not be accessed directly.

This simple pattern leverages Python's robust module semantics and is a useful way to implement singletons in a safe, reliable way.

Summary

Packages are an important concept in Python, and in this chapter we've covered most of the major topics related to implementing and working with them. Let's review the topics we looked at:

- Packages:

  - Packages are a special type of module
  - Unlike normal module, packages can contain other modules, including other packages.
  - Packages hierarchies are a powerful way to organize related code.
  - Packages have a `__path__` member which is a sequence specifying the directories from which a package is loaded.
  - A simple standard project structure includes a location for non-python files, the project's package, and a dedicated test subpackage.
- `sys.path`:
  - `sys.path` is a list of directories where Python searches for modules.
  - `sys.path` is a normal list and can be modified and queried like any other list.
  - If you start Python with no arguments, an empty string is put at the front of `sys.path`. This instructs Python to import modules from the current directory.
  - Appending directories to `sys.path` at runtime allows modules to be imported from those directories.
- `PYTHONPATH`:
  - `PYTHONPATH` is an environment variable containing a list of directories.
  - The format of `PYTHONPATH` is the same as for `PATH` on your system. It's a semicolon-separated list on Windows and a colon-separated list on Linux or Mac OS X.
  - The contents of `PYTHONPATH` are added as entries to `sys.path`.
- `__init__.py`:
  - Normal packages are implemented by putting a file named `__init__.py` into a directory.
  - The `__init__.py` file for a package is executed when the package is imported.
  - `__init__.py` files can hoist attributes from submodule into higher namespaces for convenience.
- Relative imports:
  - Relative import allow you to import modules within a package without specifying the full module path.
  - Relative imports must use the `from module import name` form of import.
  - The "from" portion of a relative import starts with at least one dot.
  - Each dot in a relative import represents a containing package.
  - The first dot in a relative import means "the package containing this module."
  - Relative imports can be useful for reducing typing.
  - Relative imports can improve modifiability in some cases.
  - In general, it's best to avoid relative imports because they can make code harder to understand.

- Namespace packages:
  - A namespace package is a package split across several directories.
  - Namespace packages are described in PEP420.
  - Namespace packages don't use `__init__.py` files.
  - Namespace packages are created when one or more directories in the Python path match an import request and no normal packages or modules match the request.
  - Each directory that contributes to a namespace package is listed in the package's `__path__` attribute.
- Executable directories:
  - Executable directories are created by putting a `__main__.py` file in a directory.
  - You execute a directory with Python by passing it to the Python executable on the command line.
  - When `__main__.py` is executed its `__name__` attribute is set to `__main__`.
  - When `__main__.py` is executed, it's parent directory is automatically added to `sys.path`.
  - The `if __name__ == '__main__':` construct is redundant in a `__main__.py` file.
  - Executable directories can be compressed into zip-files which can be executed as well.
  - Executable directories and zip-files are convenient ways to distribute Python programs.
- Modules:
  - Modules can be executed by passing them to Python with the `-m` argument.
  - The `__all__` attribute of a module is a list of string specifying the names to export when `from module import *` is used.
  - Module-level attributes provide a good mechanism for implementing singletons.
  - Modules have well-defined initialization semantics.
- Miscellaneous:
  - The standard `gzip` module allows you to work with files compressed using GZIP.
  - The standard `bz2` module allows you to work with files compressed using BZ2.

Chapter 2 - Beyond Basic Functions

By this point you should have plenty of experience with functions in Python. They are a fundamental unit of reuse in the language, and calling functions is one of the most common things you do in most Python code. But when it comes to calling things in Python, functions are just the tip of the iceberg! In this chapter we'll cover a generalisation of functions known as *callable objects*, and we'll explore some other types of callable objects including *callable instances* and *lambdas*.

Review of Functions

Up to now we've encountered free functions which are defined at module (*i.e.* global) scope and methods, which are functions enclosed within a class definition. The first parameter to an instance method is the object on which the method is invoked. Methods can also be overridden in subclasses.

We've seen that function arguments come in two flavours: positional and keyword. Positional arguments used in a call are associated with the parameters used in the definition, in order. Keyword arguments use the name of the actual argument at the call site to associate with the name of the parameter in the definition, and can be provided in any order so long as they follow any positional arguments. The choice of whether a particular argument is a positional or keyword argument is made at the call site, not in the definition; a particular argument may be passed as a positional argument in one call, but as a keyword argument in another call.

> If you struggle to remember the difference between arguments and parameters try the following alliterative mnemonic:
>
> Actual — Argument Placeholder — Parameter
>
> Some other programming languages, such as C, refer to *actual* arguments at the call-site and *formal* arguments in the definition.

Furthermore, in the function definition each formal parameter may be given a default value. It's important to remember that the right-hand-side of these default value assignments is only evaluated *once*, at the time the enclosing `def` statement is executed — which is typically when a module is first imported. As such, care must be taken when using mutable default values which will retain modifications between calls, leading to hard-to-find errors.

Lastly, we've seen that, just like almost everything else in Python, functions are first-class, which is to say they are objects which can be passed around just like any other object.

Functions as callable objects

As we have seen, the `def` keyword is responsible for binding a function object, which contains a function definition, to a function name. Here, we create a simple function `resolve()` which is just a thin wrapper around a function from the Python Standard Library `socket` module:

```
>>> import socket
>>> def resolve(host):
...     return socket.gethostbyname(host)
...
>>>
```

Inspecting the `resolve` binding shows that it refers to a function object:

```
>>> resolve
<function resolve at 0x1006f50e0>
```

and to invoke the function we must use the postfix parentheses which are the function call operator:

```
>>> resolve('sixty-north.com')
'93.93.131.30'
```

In a very real sense then, function objects are *callable* objects, insofar as we can call them.

Callable instances and the `__call__()` method

Occasionally, we would like to have a function which maintains some state, usually behind the scenes, between calls. In general, implementing functions for which the return value depends on the arguments to previous calls, rather than just the current call, is frowned upon — and rightly so — because of the difficulty of reasoning about, and therefore testing and debugging, such functions. That said, there are legitimate cases for retaining information within a function between calls, such as to implement a caching policy to improve performance. In such cases the values returned by the function given particular arguments are not changed, but the time within which a result is produced may be reduced.

There are several ways of implementing such stateful functions in Python, some of which we'll look at in the next chapter, but here we'll introduce the Python special method `__call__()` which allows objects of our own design to become *callable*, just like functions.

To demonstrate, we'll make a caching version of our DNS resolver in a file `resolver.py`:

```
# resolver.py

import socket


class Resolver:

    def __init__(self):
        self._cache = {}

    def __call__(self, host):
        if host not in self._cache:
            self._cache[host] = socket.gethostbyname(host)
        return self._cache[host]
```

Notice that just like any other method, `__call__()` accepts `self` as it's first parameter, although we won't need to provide this when we call it.

And test this from the Python REPL:

```
>>> from resolver import Resolver
```

We must call the constructor of our class to create an instance object:

```
>>> resolve = Resolver()
```

Because we implemented the `__call__()` method, we can call this instance just like a function:

```
>>> resolve('sixty-north.com')
'93.93.131.30'
```

In reality this is just syntactic sugar for:

```
>>> resolve.__call__('sixty-north.com')
'93.93.131.30'
```

But we would never use the `__call__()` form in practice.

Since `resolve` is an object of type `Resolver` we can retrieve its attributes to inspect the state of the cache which currently contains just a single entry:

```
>>> resolve._cache
{'sixty-north.com': '93.93.131.30'}
```

Let's make another call to our instance:

```
>>> resolve('pluralsight.com')
'206.71.66.43'
```

We can see that the cache has grown:

```
>>> resolve._cache
{'sixty-north.com': '93.93.131.30', 'pluralsight.com': '206.71.66.43'}
```

In order to convince ourselves that the caching is working as designed, we can run some simple timing experiments, using the Python Standard Library `timeit` module[25] which contains a handy timing function, also called `timeit`:

[25] https://docs.python.org/3/library/timeit.html

```
>>> from timeit import timeit
```

For reasons we won't go into now, the `timeit()` function accepts two code snippets as *strings*, one of which is used to perform any setup operations and the other of which is the code for which the elapsed time will be measured and reported. The function also accepts a `number` argument which, in the case of testing a cache, it's important that we set to one. Since our code needs to refer to names in the current namespace — that of the REPL — we must specifically import them from the REPL namespace (which is called `__main__`) into the namespace used by `timeit()`:

```
>>> timeit(setup="from __main__ import resolve",
...        stmt="resolve('google.com')",
...        number=1)
0.010439517005579546
```

You can see here that the DNS lookup took around one-hundredth of a second. Now execute the same line of code again:

```
>>> timeit(setup="from __main__ import resolve",
...        stmt="resolve('google.com')",
...        number=1)
4.690984496846795e-06
```

This time the time taken is short enough that Python reports it in scientific notation. Let's ask Python to report it without scientific notation using the `str.format()` method, relying on the fact that the special underscore variable stores the previous REPL result:

```
>>> print("{:f}".format(_))
0.000005
```

Now it's perhaps easier to see that the result is returned from the cache in around 5 millionths of a second, a factor of 2000 times faster.

Our callable object has been instantiated from a regular class, so it's perfectly possible to define other methods within that class, giving us the ability to create *functions* which also have methods. For example, let's add a method called `clear()` to empty the cache, and another method called `has_host()` to query the cache for the presence of a a particular host:

```
# resolver.py

import socket


class Resolver:

    def __init__(self):
        self._cache = {}

    def __call__(self, host):
        if host not in self._cache:
            self._cache[host] = socket.gethostbyname(host)
        return self._cache[host]

    def clear(self):
        self._cache.clear()

    def has_host(self, host):
        return host in self._cache
```

Let's exercise our modified resolver in a fresh REPL session. First we import and instantiate our resolver callable:

```
>>> from resolver import Resolver
>>> resolve = Resolver()
```

Then we can check whether a particular destintation has been cached:

```
>>> resolve.has_host("pluralsight.com")
False
```

We see that it has not. Of course, resolving that host by invoking our callable changes that result:

```
>>> resolve("pluralsight.com")
'206.71.66.43'
>>> resolve.has_host("pluralsight.com")
True
```

Now we can test that cache clearing works as expected:

```
>>> resolve.clear()
```

which, of course, it does:

```
>>> resolve.has_host("pluralsight.com")
False
```

So we see how the special `__call__()` method can be used to define classes which when instantiated can be called using regular function call syntax. This is useful when we want a function which maintains state between calls, or if we need one that has attributes or methods to query and modify that state.

Classes are Callable

It may not be immediately obvious, but the previous exercise also demonstrated a further type of callable object: a *class* object. Remember that everything is Python is an object, and that includes classes. We must take great care when discussing Python programs because *class objects* and *instance objects* produced from those classes are quite different things. In fact, just as the `def` keyword binds a function definition to a named reference, so the `class` keyword binds a class definition to a named reference.

Let's start a new REPL session and give a practical demonstration using our `Resolver` class:

```
>>> from resolver import Resolver
```

When we ask the REPL to evaluate the imported name `Resolver`, the REPL displays a representation of the class object:

```
>>> Resolver
<class 'resolver.Resolver'>
```

This class object is itself *callable* — and of course calling class objects is precisely what we have been doing all along whenever we have called a constructor to create new instances:

```
>>> resolve = Resolver()
```

So we see that in Python, constructor calls are made by calling the *class object*. Any arguments passed when the class object is called in this way will, in due course, be forwarded to the `__init__()` method of the class, if one has been defined.

In essence, the class object callable is a factory function which when invoked produces new instances of that class. The internal Python machinery for producing class instances is beyond the scope of this book, though.[26]

Leveraging callable classes

Knowing that classes are simply objects, and that constructor calls are nothing more than using class objects as callables, we can build interesting functions which exploit this fact. Let's write a function which returns a Python *sequence* type which will either be a `tuple` if we request an immutable sequence or a `list` if we request a mutable sequence.

```
>>> def sequence_class(immutable):
...     if immutable:
...         cls = tuple
...     else:
...         cls = list
...     return cls
```

In the function we just test a boolean flag and bind either `tuple` or `list` — both of which are *class objects* — to our `cls` reference using the assignment operator. We then return `cls`.

[26] See Chapter 5 in The Python Master

Notice that we must take care when choosing variable names which refer to class objects not to use the `class` keyword as a variable name. Popular alternatives for variables referring to class objects are the abbreviation `cls` and the deliberate misspelling `klass` with a ‘k’.

Now we can use this function to produce a sequence class with the required characteristics:

```
>>> seq = sequence_class(immutable=True)
```

We can then create an instance of the class by using the class object as a *callable* — in effect calling the constructor:

```
>>> t = seq("Timbuktu")
>>> t
('T', 'i', 'm', 'b', 'u', 'k', 't', 'u')
>>> type(t)
<class 'tuple'>
>>>
```

As an aside, we’d like to introduce you to a new language feature called *conditional expressions* which can evaluate one of two sub-expressions depending on a boolean value, and they take the form:

```
result = true_value if condition else false_value
```

This syntax is perhaps surprising, with the condition being placed between the two possible result values, but it reads nicely as English. It also emphasises the true-value, which is usually the most-common result.

We can use it to simplify our `sequence_class()` function down to a single expression, obviating the need for the intermediate variable binding `cls` and retaining the single point of return:

```
>>> def sequence_class(immutable):
...     return tuple if immutable else list
...
>>> seq = sequence_class(immutable=False)
```

```
>>>
>>> s = seq("Nairobi")
>>> s
['N', 'a', 'i', 'r', 'o', 'b', 'i']
>>> type(s)
<class 'list'>
```

Conditional expressions can be used any place a Python expression is expected. Their full syntax and somewhat tortured history is covered in PEP 308.

Lambdas

Sometimes we want to be able to create a simple callable object — usually to pass directly to a function — without the bureaucratic overhead of the `def` statement and the code block it introduces. Indeed, it many cases it is not even necessary for the callable object to be bound to a name if we are passing it directly to a function — an anonymous function object would suffice. This is where the `lambda` construct comes into play, and used with care it can make for some expressive and concise code. However, as with comprehensions, excessive use of lambdas can serve to obfuscate rather than clarify code, running counter to Pythonic principles which value readability so highly, so take care to deploy them wisely!

If you're wondering why a technique for making callable objects from Python expressions is named after the eleventh letter of the Greek alphabet, the origins go back to a foundational work in computing science in 1936 by Alonzo Church, predating electronic computers! He developed the *lambda calculus* which forms the basis of the functional programming techniques used in languages such as Lisp.

A good example of a Python function that expects a callable is the `sorted()` built-in for sorting iterable series which accepts an optional `key` argument which must be a callable. For example, if we have a list of names in strings and we wish to sort them by last name we need to pass a callable as the `key` argument of `sorted()` which will extract the second name. To do this, we can use a lambda to produce such an function, without the bother of needing to think up a name for it:

```
>>> scientists = ['Marie Curie', 'Albert Einstein', 'Niels Bohr',
...               'Isaac Newton', 'Dmitri Mendeleev', 'Antoine Lavoisier',
...               'Carl Linnaeus', 'Alfred Wegener', 'Charles Darwin']
>>> sorted(scientists, key=lambda name: name.split()[-1])
['Neils Bohr', 'Marie Curie', 'Charles Darwin', 'Albert Einstein',
'Antoine Lavoisier', 'Carl Linnaeus', 'Dmitri Mendeleev',
'Isaac Newton', 'Alfred Wegener']
```

Here our lambda accepts a single argument called `name` and the body of the lambda follows the colon. It calls `str.split()` and returns the last element of the resulting sequence using negative indexing. In isolation the lambda is just this part:

```
lambda name: name.split()[-1]
```

Lambda is itself an expression which results in a callable object. We can see this by binding the result of the lambda expression to a named reference using assignment:

```
>>> last_name = lambda name: name.split()[-1]
```

We can see that the resulting object is a function:

```
>>> last_name
<function <lambda> at 0x1006fa830>
```

And it is indeed callable like a function:

```
>>> last_name("Nikola Tesla")
'Tesla'
```

Creating callable functions this way — using a lambda and binding to a name through assignment — is equivalent to defining a regular function using `def`, like this:

```
def first_name(name):
    return name.split()[0]
```

This a good time to point out the differences between lambdas and regular functions or methods:

- `def` is a *statement* which defines a function and has the effect of binding it to a name. `lambda` is an *expression* which returns a function object.
- Regular functions must be given a name, whereas lambdas are anonymous.
- The argument list for functions is delimited by parentheses and separated by commas. The argument list for lambdas is terminated by a colon and separated by commas. Lambda arguments are *without* enclosing parentheses. Versions of Python predating Python 3 have special handling of tuples using tuple unpacking in the argument list. This confusing feature has been removed from Python 3, so in Python 3 code there are never any parentheses between the `lambda` keyword and the colon after the argument list.
- Both regular functions and `lambda` support zero or more arguments. A zero argument function uses empty parentheses for the argument list, whereas a zero argument lambda places the colon immediately following the `lambda` keyword.
- The body of a regular function is a block containing statements, whereas the body of a lambda is an *expression* to be evaluated. The lambda body can contain only a single expression, no statements.
- Any return value from a regular function must be explicitly returned using the `return` statement. No `return` statement is needed, or indeed allowed, in the lambda body. The return value will be the value of the supplied expression.
- Unlike regular functions, there is no simple way to document a lambda with a docstring.
- Regular functions can easily be tested using external testing tools, because they can be fetched by name. Most lambdas can't be tested in this way, simply because they're anonymous and can't be retrieved. This points the way to a guideline: keep your lambdas simple enough that they're obviously correct by inspection.

Detecting Callable Objects

To determine whether an object is callable, you can simply pass it to the built-in function `callable()` which returns `True` or `False`.

So, as we have seen, regular functions are callable:

```
>>> def is_even(x):
...     return x % 2 == 0
...
>>> callable(is_even)
True
```

Lambda expressions are also callable:

```
>>> is_odd = lambda x: x % 2 == 1
>>> callable(is_odd)
True
```

Class objects are callable because calling a class invokes the constructor:

```
>>> callable(list)
True
```

Methods are callable:

```
>>> callable(list.append)
True
```

Instance objects can be made callable by defining the `__call__()` method:

```
>>> class CallMe:
...     def __call__(self):
...         print("Called!")
...
>>> my_call_me = CallMe()
>>> callable(my_call_me)
True
```

On the other hand, many objects, such as string instances, are not callable:

```
>>> callable("This is not callable")
False
```

Extended Formal Parameter Syntax

We've already been using functions which support extended argument syntax, although you may not have realised it.

For example, you may have wondered how it is possible for `print()` to accept zero, one, two, or in fact any number of arguments:

```
>>> print()

>>> print("one")
one
>>> print("one", "two")
one two
>>> print("one", "two", "three")
one two three
```

Another example we've seen is the use of the `str.format()` method which can accept arbitrary named arguments corresponding to the format placeholders in the string:

```
>>> "{a}<===>{b}".format(a="Oslo", b="Stavanger")
'Oslo<===>Stavanger'
```

In this section we'll learn how to define functions — or more generally callables — which can accept arbitrary positional or keyword arguments.

Let's start with positional arguments.

Positional Arguments

Drawing an example from geometry, let's write a function which can return the area of a two-dimensional rectangle, the volume of a three-dimensional cuboid, or indeed the hyper-volume of an *n*-dimensional hyper-cuboid. Such a function needs to accept an arbitrary number of numeric arguments and multiply them together.

To do this we use a special argument syntax where the argument name is prefixed with a single asterisk:

```
>>> def hypervolume(*args):
...     print(args)
...     print(type(args))
```

Before we actually implement the computation, we'll print out the value of `args` and its type using the built-in `type()` function. Notice that the asterisk does not form part of the argument name, and the argument name we have chosen here, `args`, is conventionally (though not compulsorily) used in this situation. Colloquially this form is called "star-args". Now let's call our function a few times:

```
>>> hypervolume(3, 4)
(3, 4)
<class 'tuple'>
>>> hypervolume(3, 4, 5)
(3, 4, 5)
<class 'tuple'>
```

We can see that `args` is passed as a tuple which contains the function arguments. Knowing this, it's a simple matter to write code to multiply all the values in the tuple together to get the result. Redefining `hypervolume()` and using more a documentary name for the argument gives us:

```
>>> def hypervolume(*lengths):
...     i = iter(lengths)
...     v = next(i)
...     for length in i:
...         v *= length
...     return v
```

The function works by obtaining an iterator `i` over the tuple and using `next()` to retrieve the first value which is used to initialise an variable `v` in which the final volume will be accumulated. We then use a for-loop to continue iteration with the same iterator to deal with remainder of the values.

We can use the function to compute the areas of rectangles:

```
>>> hypervolume(2, 4)
8
```

We can also calculate the volumes of cuboids:

```
>>> hypervolume(2, 4, 6)
48
```

Because the function accepts any number of arguments, it can even calculate the hypervolumes of hyper-cuboids:

```
>>> hypervolume(2, 4, 6, 8)
384
```

It also generalises nicely down to lower dimensions to give us the length of lines:

```
>>> hypervolume(1)
1
```

However, if called with no arguments the function raises `StopIteration`. This exposes an implementation detail about which clients of our function should be unaware:

```
>>> hypervolume()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 3, in hypervolume
StopIteration
```

There are a couple of approaches to fixing this. One change could be to wrap the call to `next()` in a `try..except` construct and translate the exception into something more meaningful for the caller, such as the `TypeError` that is usually raised when an insufficient number of arguments is passed to a function. We'll take a different approach of using a regular positional argument for the first length, and the star-args to soak up any further length arguments:

```
>>> def hypervolume(length, *lengths):
...     v = length
...     for item in lengths:
...         v *= item
...     return v
...
```

Using this design the function continues to work as expected when arguments are supplied:

```
>>> hypervolume(3, 5, 7, 9)
945
>>> hypervolume(3, 5, 7)
105
>>> hypervolume(3, 5)
15
>>> hypervolume(3)
3
```

and raises a predictable `TypeError` exception when insufficient arguments are given:

```
>>> hypervolume()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: hypervolume() missing 1 required positional argument: 'length'
```

At the same time the revised implementation is even simpler and easier to understand than the previous version which used an iterator. Later, in chapter 7, we'll show you how to use `functools.reduce()`, which also could also have been used to implement our `hypervolume()` function.

When you need to accept a variable number of arguments with a positive lower-bound, you should consider this practice of using regular positional arguments for the required parameters and star-args to deal with any extra arguments.

> Note that star-args must come after any normal positional parameters, and that there can only be one occurrence of star-args within the parameter list.

The star-args syntax only collects positional arguments, and a similar syntax is provided for handling keyword arguments. Let's look at that now.

Keyword arguments

Arbitrary keyword arguments can be accepted by callables that use a parameter prefixed by `**`. Conventionally this parameter is called `kwargs` although depending on your situation you may care to choose a more meaningful name.

Let's make a function which returns a single HTML tag as a string. The first argument to the function will be a regular positional argument which will accept the tag name. This will be followed by a the arbitrary keyword-args construct to which tag attributes can be passed.

As before, we'll perform a simple experiment to determine how the keyword arguments are delivered:

```
>>> def tag(name, **kwargs):
...     print(name)
...     print(kwargs)
...     print(type(kwargs))
```

Then we call the function with some suitable attributes to create an HTML image tag, like this:

```
>>> tag('img', src="monet.jpg", alt="Sunrise by Claude Monet", border=1)
img
{'alt': 'Sunrise by Claude Monet', 'src': 'monet.jpg', 'border': 1}
<class 'dict'>
```

We can see that the arguments are transferred to our keyword-arguments formal parameter as a regular Python dictionary, where each key is a string bearing the actual argument name. Note that as with any other Python dictionary the ordering of the arguments is not preserved.[27]

Now we'll go ahead and implement our `tag()` function properly, using a more descriptive name than `kwargs`:

[27] As of Python 3.6, which incorporates PEP 468, `**kwargs` in a function signature is guaranteed to be an insertion-order-preserving mapping. In other words, the arguments in the received dictionary will be in the same order as the arguments at the call-site.

```
>>> def tag(name, **attributes):
...     result = '<' + name
...     for key, value in attributes.items():
...         result += ' {k}="{v}"'.format(k=key, v=str(value))
...     result += '>'
...     return result
...
>>> tag('img', src="monet.jpg", alt="Sunrise by Claude Monet", border=1)
'<img alt="Sunrise by Claude Monet" src="monet.jpg" border="1">'
```

Here we iterate over the items in the `attributes` dictionary, building up the result string as we go. It's worth pointing out at this stage that `str.format()` also uses the arbitrary keyword-args technique to allow us to pass arbitrary named arguments corresponding to our replacement fields in the format string.

This example also shows that it's quite possible to combine positional arguments and keyword arguments. In fact, the overall syntax is very powerful, so long as we respect the order of the arguments we define.

First, `*args`, if present, must always precede `**kwargs`. So this isn't allowed:

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

Second, any arguments preceding `*args` are taken to be regular positional arguments, as we saw in the `hypervolume()` example earlier:

```
>>> def print_args(arg1, arg2, *args):
...     print(arg1)
...     print(arg2)
...     print(args)
...
>>> print_args(1, 2, 3, 4, 5)
1
2
(3, 4, 5)
```

Thirdly, any regular arguments *after* `*args` must be passed as mandatory keyword arguments:

```
>>> def print_args(arg1, arg2, *args, kwarg1, kwarg2):
...     print(arg1)
...     print(arg2)
...     print(args)
...     print(kwarg1)
...     print(kwarg2)
...
>>> print_args(1, 2, 3, 4, 5, kwarg1=6, kwarg2=7)
1
2
(3, 4, 5)
6
7
```

Failure to do so results in a `TypeError`:

```
>>> print_args(1, 2, 3, 4, 5, 6, 7)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: print_args() missing 2 required keyword-only arguments: 'kwarg1' and 'kwar\
g2'
```

Fourthly, we have the `**kwargs` arbitrary keyword-arguments, which if present must be last in the argument list:

```
>>> def print_args(arg1, arg2, *args, kwarg1, kwarg2, **kwargs):
...     print(arg1)
...     print(arg2)
...     print(args)
...     print(kwarg1)
...     print(kwarg2)
...     print(kwargs)
...
>>> print_args(1, 2, 3, 4, 5, kwarg1=6, kwarg2=7, kwarg3=8, kwarg4=9)
1
2
(3, 4, 5)
6
7
{'kwarg4': 9, 'kwarg3': 8}
```

Any attempt to define an additional formal parameter after `**kwargs` results in a syntax error:

```
>>> def print_args(arg1, arg2, *args, kwarg1, kwarg2, **kwargs, kwargs99):
File "<stdin>", line 1
def print_args(arg1, arg2, *args, kwarg1, kwarg2, **kwargs, kwargs99):
^
SyntaxError: invalid syntax
```

Notice that the third rule above gives us a way to specify keyword-only arguments, but only if we have an occurrence of `*args` preceding the keyword-only arguments in the parameter list. This isn't always convenient. Sometimes we want keyword-only arguments without any arbitrary positional arguments as facilitated by star-args.

To accommodate this, Python allows for a special unnamed star-args argument which is just an asterisk in the parameter list. Doing so can be used to mark the end of the positional arguments, and any subsequent arguments must be supplied as keywords:

```
>>> def print_args(arg1, arg2, *, kwarg1, kwarg2):
...     print(arg1)
...     print(arg2)
...     print(kwarg1)
...     print(kwarg2)
...
>>> print_args(1, 2, kwarg1=6, kwarg2=7)
1
2
6
7
>>> print_args(1, 2, 3, kwarg1=6, kwarg2=7)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: print_args() takes 2 positional arguments but 3 positional arguments
(and 2 keyword-only arguments) were given
```

So in summary the syntax for argument lists is:

```
[[[mandatory-positional-args], *[args]], mandatory-keyword-args], **kwargs]]
```

You should take particular care when combining these language features with default arguments. These have their own ordering rule specifying that mandatory arguments must be specified before optional arguments at the call site.

Before moving on we should point at that all the features of the extended argument syntax apply equally to regular functions, lambdas, and other callables, although it's fair to say they are rarely seen in combination with `lambda` in the wild.

Extended Call Syntax

The complement to extended formal parameter syntax is extended call syntax. This allows us to use any iterable series, such as a tuple, to populate positional arguments, and any mapping type with string keys, such as a dictionary, to populate keyword arguments.

Let's go back to a simple version of `print_args()` which only deals with mandatory positional and `*args` arguments:

```
>>> def print_args(arg1, arg2, *args):
...     print(arg1)
...     print(arg2)
...     print(args)
```

We'll now create an iterable series — in this case a `tuple`, although it could be any other conforming type — and apply it at the call-site for `print_args()` using the asterisk prefix to instruct Python to unpack the series into the positional arguments:

```
>>> t = (11, 12, 13, 14)
>>> print_args(*t)
11
12
(13, 14)
```

Notice that the use of the * syntax in the actual arguments does not necessarily need to correspond to the use of * in the formal parameter list. In this example, the first two elements of our tuple have been unpacked into the mandatory positional parameters and the last two have been transferred into the `args` tuple.

Similarly, we can use the double-asterisk prefix at the call site to unpack a mapping type, such as a dictionary, into the keyword parameters, mandatory or optional. First we'll define a function `color()` which accepts three parameters `red`, `green` and `blue`. Each of these three parameters could be used as either positional *or* keyword arguments at the call site. At the end of the parameter list we add `**kwargs` to soak up any additional keyword arguments that are passed:

```
>>> def color(red, green, blue, **kwargs):
...     print("r =", red)
...     print("g =", green)
...     print("b =", blue)
...     print(kwargs)
...
```

Now we'll create a dictionary to serve as our mapping type of keyword arguments and apply it at the function call-site using the ** prefix:

```
>>>
>>> k = {'red':21, 'green':68, 'blue':120, 'alpha':52 }
>>> color(**k)
r = 21
g = 68
b = 120
{'alpha': 52}
```

Notice again how there's no necessary correspondence between the use of ** in the actual arguments versus the use of ** in the formal parameter list. Items in the dictionary are matched up with the parameters and any remaining entries are bundled into the `kwargs` parameter.

Before moving on, we'll remind you that the `dict()` constructor uses the `**kwargs` technique to permit the creation of dictionaries directly from keyword arguments. We could have used that technique to construct the dictionary `k` in the previous example, but we but we didn't want to make the example more complex than necessary. Otherwise, we could have constructed `k` as:

```
k = dict(red=21, green=68, blue=120, alpha=52)
```

Forwarding arguments

One of the most common uses of `*args` and `**kwargs` is to use them in combination to forward all arguments of one function to another.

For example, suppose we define a function for tracing the arguments and return values of other functions. We pass the function whose execution is to be traced as one argument, but that function could itself take any arguments whatsoever. We can use extended parameter syntax to accept any arguments to our tracing function and extended call syntax to pass those arguments to the traced function:

```
>>> def trace(f, *args, **kwargs):
...     print("args =", args)
...     print("kwargs =", kwargs)
...     result = f(*args, **kwargs)
...     print("result =", result)
...     return result
...
```

We'll trace a call to `int("ff", base=16)` to demonstrate that `trace()` can work with any function without advance knowledge of the signature of that function.:

```
>>> trace(int, "ff", base=16)
args = ('ff',)
kwargs = {'base': 16}
result = 255
255
```

Transposing Tables

Recall that the `zip()` built-in function can be used to combine two iterable series elementwise into one series. This new series contains tuples whose elements are corresponding elements from the two series passed to `zip()`.

Consider these tables of daytime temperatures:

```
>>> sunday = [12, 14, 15, 15, 17, 21, 22, 22, 23, 22, 20, 18]
>>> monday = [13, 14, 14, 14, 16, 20, 21, 22, 22, 21, 19, 17]
>>> for item in zip(sunday, monday):
...     print(item)
...
(12, 13)
(14, 14)
(15, 14)
(15, 14)
(17, 16)
(21, 20)
(22, 21)
(22, 22)
(23, 22)
(22, 21)
(20, 19)
(18, 17)
```

Recall further that `zip()` in fact accepts any number of iterable series. It achieves this by accepting an argument of `*iterables`, that is any number of iterables as positional arguments:

```
>>> tuesday = [2, 2, 3, 7, 9, 10, 11, 12, 10, 9, 8, 8]
>>> for item in zip(sunday, monday, tuesday):
...     print(item)
...
(12, 13, 2)
(14, 14, 2)
(15, 14, 3)
(15, 14, 7)
(17, 16, 9)
(21, 20, 10)
(22, 21, 11)
(22, 22, 12)
(23, 22, 10)
(22, 21, 9)
(20, 19, 8)
(18, 17, 8)
```

Now consider what we would need to do if, instead of three separate lists for `sunday`, `monday`, and `tuesday`, we had a single data structure in the form of a list of lists. We can make such a data structure like this:

```
>>> daily = [sunday, monday, tuesday]
```

We can pretty-print it using the Python Standard Library `pprint` function[28] from the module of the same name:

```
>>> from pprint import pprint as pp
>>> pp(daily)
[[12, 14, 15, 15, 17, 21, 22, 22, 23, 22, 20, 18],
[13, 14, 14, 14, 16, 20, 21, 22, 22, 21, 19, 17],
[2, 2, 3, 7, 9, 10, 11, 12, 10, 9, 8, 8]]
```

Now our loop over the output of `zip()` could be rendered as:

```
>>> for item in zip(daily[0], daily[1], daily[2]):
...     print(item)
...
(12, 13, 2)
(14, 14, 2)
(15, 14, 3)
(15, 14, 7)
(17, 16, 9)
(21, 20, 10)
(22, 21, 11)
(22, 22, 12)
(23, 22, 10)
(22, 21, 9)
(20, 19, 8)
(18, 17, 8)
```

Now we return to one of the main topics of this chapter, extended call syntax, which allows us to apply any iterable series to function call arguments using the * prefix — so-called star-args. Our list of lists is perfectly acceptable as an iterable series of iterable series, so we can use extended call syntax like so:

[28] https://docs.python.org/3/library/pprint.html#pprint.pprint

```
>>> for item in zip(*daily):
...     print(item)
...
(12, 13, 2)
(14, 14, 2)
(15, 14, 3)
(15, 14, 7)
(17, 16, 9)
(21, 20, 10)
(22, 21, 11)
(22, 22, 12)
(23, 22, 10)
(22, 21, 9)
(20, 19, 8)
(18, 17, 8)
```

Or, to produce the result as a single data structure, we can wrap the result in a call to `list`:

```
>>> t = list(zip(*daily))
```

Notice what is happening here. We have transformed this structure:

```
>>> pp(t)
[[12, 14, 15, 15, 17, 21, 22, 22, 23, 22, 20, 18],
[13, 14, 14, 14, 16, 20, 21, 22, 22, 21, 19, 17],
[2,   2,   3,   7,   9, 10, 11, 12, 10,   9,   8,   8]]
```

into this structure:

```
>>> pp(t)
[(12, 13, 2),
(14, 14, 2),
(15, 14, 3),
(15, 14, 7),
(17, 16, 9),
(21, 20, 10),
(22, 21, 11),
(22, 22, 12),
(23, 22, 10),
(22, 21, 9),
(20, 19, 8),
(18, 17, 8)]
```

Converting columns into rows and rows in columns like this is an operation known as *transposition*. This "zip-star" idiom is an important technique to learn, not least because if you're not familiar with the idea it may not be immediately obvious what the code is doing. It's fairly widely used in Python code and definitely one worth learning to recognise on sight.

Summary

Understanding callable objects and Python's calling syntax is a critical step in effective use of Python, and in this chapter we've covered the important details of each. The topics we've looked at include:

- Callable objects:
    - The idea of functions can be generalised to the the notion of *callables*
    - We can make callable objects from instances by implementing the special `__call__()` method on our classes and then invoking the object as if it were a function
    - Callable instances allow us to define "functions" which maintain state between calls
    - Callable instances also allow us to give "functions" attributes and methods which can be used to query and modify that state
    - Whenever we create an object by invoking a constructor we're actually *calling* a class object. Class objects are themselves callable
    - Class objects can be used just like any other callable object, including being passed to, and returned from, functions, and bound to names through assignment
    - Callable objects can be detected using the built-in `callable()` predicate function
- Lambda expressions:
    - A single expression can be used as a callable by creating a `lambda` which is an anonymous callable
    - Lambdas are most frequently used inline and passed directly as arguments to other functions
    - Unlike regular functions, the `lambda` argument list isn't enclosed in parentheses
    - The `lambda` body is restricted to being a single expression, the value of which will be returned
- Extended parameter and call syntax:
    - Extended parameter syntax allows arbitrary positional arguments to be accepted using the star-args syntax in the callable definition, which results in arguments being packaged into a tuple

  - Similarly, arbitrary keyword arguments can be accepted using the double-star-kwargs syntax which results in the keyword arguments being packaged into a dictionary
  - Extended call syntax allows us to unpack iterable series and mappings into positional and keyword function parameters respectively
  - There is no requirement for use of * and ** at the call-site to correspond to the use of * and ** in the definition. Arguments will be unpacked and repacked into the parameters as necessary
  - `*args` and `**kwargs` can be combined with mandatory positional and keyword arguments in a well-defined order
- Miscellaneous:
  - The `timeit` module can be used to measure the performance of small code snippets
  - Python supports a syntax for conditional expressions of the form `result = true_value if condition else false_value`
  - `zip()` uses extended argument syntax to accept an arbitrary number of iterable series as arguments. By combining `zip()` with the extended call syntax using * to unpack an iterable series of iterable series we can transpose two-dimensional tables of data, converting rows into columns and vice-versa
  - The `list(zip(*table))` idiom is widespread enough that you need to be able to recognise it on sight

Chapter 3 - Closures and Decorators

The functions we've looked at so far have been defined either at module scope, as a member of a class, or as anonymous "lambda" functions. Python provides us with more flexibility than that though. In this chapter we'll look at *local functions*, functions which we define within the scopes of other functions. We'll also look at the related concept of *closures* which are key to really understanding local functions. We'll close off the chapter with a look at Python's function decorators, a powerful and elegant way to augment existing functions without changing their implementation.

Local functions

As you'll recall, in Python the `def` keyword is used to define new functions. `def` essentially binds the body of the function to a name in such a way that functions are simply objects like everything else in Python. It's important to remember that `def` is executed at runtime, meaning that functions are defined at runtime.

Up to now, almost all of the functions we've looked at have been defined at module scope or inside classes (in which case we refer to them as methods.) However, Python doesn't restrict you to just defining functions in those two contexts. In fact, Python allows you to define functions inside other functions. Such functions are often referred to as *local functions* since they're defined local to a specific function's scope.

Let's see a quick example:

```
def sort_by_last_letter(strings):
    def last_letter(s):
        return s[-1]
    return sorted(strings, key=last_letter)
```

Here we define a function `sort_by_last_letter` which sorts a list of strings by their last letter. We do this by using the `sorted` function and passing `last_letter()` as the key function. `last_letter()` is defined *inside* `sort_by_last_letter()`; it is a local function.

Let's test it out:

```
>>> sort_by_last_letter(['hello', 'from', 'a', 'local', 'function'])
>>> ['a', 'local', 'from', 'function', 'hello']
```

Local functions are defined on each call

Just like module-level function definitions, the definition of a local function happens at run time when the `def` keyword is executed. Interestingly, this means that each call to `sort_by_last_letter` results in a new definition of the function `last_letter`. That is, just like any other name bound in a function body, `last_letter` is bound separately to a new function each time `sort_by_last_letter` is called.

We can see this for ourselves by making a small modification to `sort_by_last_letter` to print the `last_letter` object:

```
def sort_by_last_letter(strings):
    def last_letter(s):
        return s[-1]
    print(last_letter)
    return sorted(strings, key=last_letter)
```

If we run this a few times we see that each execution of `sort_by_last_letter` results in a new `last_letter` instance:

```
>>> sort_by_last_letter(['ghi', 'def', 'abc'])
<function sort_by_last_letter.<locals>.last_letter at 0x10cdff048>
['abc', 'def', 'ghi']

>>> sort_by_last_letter(['ghi', 'def', 'abc'])
<function sort_by_last_letter.<locals>.last_letter at 0x10cdff158>
['abc', 'def', 'ghi']

>>> sort_by_last_letter(['ghi', 'def', 'abc'])
<function sort_by_last_letter.<locals>.last_letter at 0x10cdff158>
'abc', 'def', 'ghi']
```

The main point here is that the `def` call in `sort_by_last_letter` is no different from any other name binding in the function, and a new function is created each time `def` is executed.

LEGB and local functions

Local functions are subject to the same scoping rules as other functions. Remember the LEGB rule[29] for name lookup: first the **L**ocal scope is checked, then any **E**nclosing scope, next the **G**lobal scope, and finally the **B**uiltin scope. ****

This means that name lookup in local functions starts with names defined within the local function itself. It proceeds to the enclosing scope, which in this case is the containing function; this enclosing scope includes both the local names of the containing function as well as its parameters. Finally, the global scope includes any module-level name bindings.

We can see this in a small example:

```
>>> g = 'global'
>>> def outer(p='param'):
...     l = 'local'
...     def inner():
...         print(g, p, l)
...     inner()
...
>>> outer()
global param local
```

Here we define the function `inner` local to `outer`. `inner` simply prints a global variable and a few bindings from `outer`.

This example shows the essence of how the LEGB rule applies to local functions. If you don't fully understand what's going on, you should play with this code on your own until it's clear.

Local functions are not "members"

It's important to note that local functions are not "members" of their containing function in any way. As we've mentioned, local functions are simply local name bindings in the function body. To see this, you can try to call a local function via member access syntax:

[29] See Chapter 4 in The Python Apprentice

```
>>> outer.inner()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'function' object has no attribute 'inner'
```

The function object `outer` has no attribute named `inner`. `inner` is only defined when `outer` is executed, and even then it's just a normal variable in the execution of the function's body.

When are local functions useful?

So what are local functions useful for? As we've seen they are useful for things like creating sorting-key functions. It makes sense to define these close to the call-site if they're one-off, specialized functions. So local functions are a code organization and readability aid. In this way they're similar to lambdas which, as you'll recall, are simple, unnamed function objects. Local functions are more general than lambdas, though, since they may contain multiple expressions and may contain statements such as `import`.

Local functions are also useful for other, more interesting purposes, but before we can look at those we'll need to investigate two more concepts: returning functions from functions, and closures.

Returning functions from functions

As we've just seen, local functions are no different from any other object created inside a function's body: New instances are created for each execution of the enclosing function, they're not somehow specially bound to the enclosing function, and so forth. Like other bindings in a function, local functions can also be *returned* from functions.

Returning a local function does not look any different from returning any other object. Let's see an example:

```
>>> def enclosing():
...     def local_func():
...         print('local func')
...     return local_func
...
>>> lf = enclosing()
>>> lf()
local func
```

Here `enclosing` defines `local_func` and returns it. Callers of `enclosing()` can bind its return value to a name — in this case `lf` — and then call it like any other function. In fact, `enclosing` can be considered a *function-factory*.

This ability to return functions is part of the broader notion of "first class functions" where functions can be passed to and returned from other functions or, more generally, treated like any other piece of data. This concept can be very powerful, particularly when combined with *closures* which we'll explore in the next section.

Closures and nested scopes

So far the local functions we've looked at have all been fairly boring. They are defined within another function's scope, but they don't really interact with the enclosing scope. However, we did see that local functions can reference bindings in their enclosing scope via the LEGB rule. Furthermore, we saw that local functions be *returned* from their defining scope and executed in another scope.

This raises an interesting question: How does a local function use bindings to objects defined in a scope that no longer exists? That is, once a local function is returned from its enclosing scope, that enclosing scope is gone, along with any local objects it defined. How can the local function operate without that enclosing scope?

The answer is that the local function forms what is known as a *closure*. A closure essentially remembers the objects from the enclosing scope that the local function needs. It then keeps them alive so that when the local function is executed they can still be used. One way to think of this is that the local function "closes over" the objects it needs, preventing them from being garbage collected.

Python implements closures with a special attribute named `__closure__`. If a function closes over any objects, then that function has a `__closure__` attribute which maintains the necessary references to those objects. We can see that in a simple example:

```
>>> def enclosing():
...     x = 'closed over'
...     def local_func():
...             print(x)
...     return local_func
...
>>> lf = enclosing()
>>> lf()
closed over
>>> lf.__closure__
(<cell at 0x10220f2f0: str object at 0x1021f8bb0>,)
```

The `__closure__` attribute of `lf` indicates that `lf` is a closure, and we can see that the closure is referring to a single object. In this case, that object is the `x` variable defined in the function that defined `lf`.

Function factories

We can see that local functions can safely use objects from their enclosing scope. But how is this really useful?

A very common use for closures is in *function factories*. These factories are functions that return other functions, where the returned functions are specialized in some way based on arguments to the factory. In other words, the factory function takes some arguments. It then creates a local function which takes its own arguments but also uses the arguments passed to the factory. The combination of runtime function definition and closures makes this possible.

A typical example of this kind of factory creates a function which raises numbers to a particular power. Here's how the factory looks:

```
def raise_to(exp):
    def raise_to_exp(x):
        return pow(x, exp)
    return raise_to_exp
```

`raise_to` takes a single argument, `exp`, which is an exponent. It returns a function that raises its arguments to that exponent. You can see that the local function `raise_to_exp` refers to `exp` in its implementation, and this means that Python will create a closure to refer to that object.

If we call `raise_to` we can verify that it creates this closure:

```
>>> square = raise_to(2)
>>> square.__closure__
(<cell at 0x103484be8: int object at 0x1002330c0>,)
```

And we can also see that `square` does indeed behave as we expect:

```
>>> square(5)
25
>>> square(9)
81
>>> square(1234)
1522756
```

And we can create other functions the same way:

```
>>> cube = raise_to(3)
>>> cube(3)
27
>>> cube(10)
1000
>>> cube(23)
12167
```

The `nonlocal` keyword

The use of local functions raises some interesting questions regarding name lookup. We've looked in some detail at the LEGB rule which determines how names are resolved in Python when we want the values to which those names refer. However, LEGB doesn't apply when we're making new name bindings.

Consider this simple example:

```
message = 'global'

def enclosing():
    message = 'enclosing'

    def local():
        message = 'local'
```

When we assign to `message` in the function `local`, what precisely is happening? In this case, we're creating a new name binding in that function's scope from the name `message` to the string "local". Critically, we are *not* rebinding either of the other two `message` variables in the code.

We can see this by instrumenting the code a bit:

```
message = 'global'

def enclosing():
    message = 'enclosing'

    def local():
        message = 'local'

    print('enclosing message:', message)
    local()
    print('enclosing message:', message)

print('global message:', message)
enclosing()
print('global message:', message)
```

Now we're actually calling the functions `enclosing()` and `local()`, and we can see that neither the enclosing nor the global bindings for the name `message` is affected when `local()` assigns to the name `message`. Again, `local()` is creating an entirely new name binding which only applies in the context of that function. If we run this code, we'll see that neither the global nor enclosing bindings for `message` are affected by calling `local()`:

```
global message: global
enclosing message: enclosing
enclosing message: enclosing
global message: global
```

In *The Python Apprentice*[30] we discussed Python's `global` keyword which can be used to introduce a binding from the global scope into another scope. So in our example, if we wanted the function `local()` to modify the global binding for `message` rather than creating a new one, we could use the `global` keyword to introduce the global `message` binding into `local()`. Let's do that and see the effects.

First, let's use the `global` keyword to introduce the module-level binding of `message` into the function `local()`:

```
message = 'global'

def enclosing():
    message = 'enclosing'

    def local():
        global message
        message = 'local'

    print('enclosing message:', message)
    local()
    print('enclosing message:', message)

print('global message:', message)
enclosing()
print('global message:', message)
```

If we run this, we can see that the module-level binding of `message` is indeed changed when `local` is called:

[30] See Chapter 4 in The Python Apprentice

```
global message: global
enclosing message: enclosing
enclosing message: enclosing
global message: local
```

Again, the `global` keyword should be familiar to you already. If it's not, you can always review Chapter 4 of *The Python Apprentice*.

Accessing enclosing scopes with `nonlocal`

If `global` allows you to insert module-level name bindings into a function in Python, how can you do the same for name bindings in enclosing scopes? Or, in terms of our example, how can we make the function `local()` modify the binding for `message` defined in the function `enclosing()`?

The answer to that is that Python also provides the keyword `nonlocal` which inserts a name binding from an enclosing namespace into the local namespace. More precisely, `nonlocal` searches the enclosing namespaces from innermost to outermost for the name you give it. As soon as it finds a match, that binding is introduced into the scope where `nonlocal` was invoked.

Let's modify our example again to show how the function `local()` can be made to modify the binding of `message` created in the function `enclosing()` by using `nonlocal`:

```
message = 'global'

def enclosing():
    message = 'enclosing'

    def local():
        nonlocal message
        message = 'local'

    print('enclosing message:', message)
    local()
    print('enclosing message:', message)

print('global message:', message)
enclosing()
print('global message:', message)
```

Now when we run this code, we see that `local()` is indeed changing the binding in `enclosing()`:

```
global message: global
enclosing message: enclosing
enclosing message: local
global message: global
```

`nonlocal` references to nonexistent names

It's important to remember that it's an error to use `nonlocal` when no matching enclosing binding exists. If you do this, Python will raise a `SyntaxError`.

You can see this if you add a call to `nonlocal` in `local()` which refers to a non-existent name:

```
message = 'global'

def enclosing():
    message = 'enclosing'

    def local():
        nonlocal no_such_name
        message = 'local'

    print('enclosing message:', message)
    local()
    print('enclosing message:', message)

print('global message:', message)
enclosing()
print('global message:', message)
```

When you try to execute this code, Python will complain that `no_such_name` does not exist:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
SyntaxError: no binding for nonlocal 'no_such_name' found
```

Like `global`, `nonlocal` is not something you're likely to need to use a lot, but it's important to understand how to use it for those times when it really is necessary, or for when you see it used in other peoples' code. To really drive it home, let's create a more practical example that uses `nonlocal`.

A more practical use of `nonlocal`

In this example the `make_timer()` function returns a new function. Each time you call this new function, it returns the elapsed time since the *last* time you called it. Here's how it looks:

```python
import time

def make_timer():
    last_called = None  # Never

    def elapsed():
        nonlocal last_called
        now = time.time()
        if last_called is None:
            last_called = now
            return None
        result = now - last_called
        last_called = now
        return result

    return elapsed
```

And here's how you can use it:

```
>>> t = make_timer()
>>> t()
>>> t()
1.6067969799041748
>>> t()
2.151050090789795
>>> t()
2.5112199783325195
```

As you can see, the first time you invoke `t` it returns nothing. After that, it returns the amount of time since the last invocation.

How does this work? Every time you call `make_timer()`, it creates a new local variable named `last_called`. It then defines a local function called `elapsed()` which uses the nonlocal keyword to insert make_timers()'s binding of last_called into its local scope. The inner elapsed() elaped function then uses the last_called binding to keep track of the last time it was called. In other words, elapsed() uses nonlocal to refer to a name binding which will exist across multiple calls to elapsed(). In this way, elapsed() is using nonlocal to create a form of persistent storage.

It's worth noting that each call to make_timer() creates a new, independent binding of last_called() as well as a new definition of elapsed(). This means that each call to make_timer() creates a new, independent timer object, which you can verify by creating multiple timers:

```
>>> t1 = make_timer()
>>> t2 = make_timer()
>>> t1()
>>> t1()
1.2153239250183105
>>> t2()
>>> t2()
1.1208369731903076
>>> t2()
1.9121758937835693
>>> t2()
1.4715540409088135
>>> t2()
1.4720590114593506
>>> t1()
8.223593950271606
>>> t1()
1.487989902496338
```

As you can see, calls to t1() have no effect on t2(), and they are both keeping independent times.

Function decorators

Now that we've looked at the concepts of local functions and closures, we finally have the tools we need to understand an interesting and useful Python feature called *decorators*. At a high level, decorators are a way to modify or enhance existing functions in a non-intrusive and maintainable way.

In Python, a decorator is a callable object that takes in a callable and returns a callable. If that sounds a bit abstract, it might be simpler for now to think of decorators as functions that take a function as an argument and return another function, but the concept is more general than that.

The `@` syntax for decorators

Coupled with this definition is a special syntax that lets you "decorate" functions with decorators. The syntax looks like this:

```
@my_decorator
def my_function():
    # . . .
```

This example applies the decorator `my_decorator` to the function `my_function()`. The `@` symbol is the special syntax for applying decorators to functions.

What is "decorating"?

So what does this actually do? When Python sees decorator application like this, it first compiles the base function, which in this case is `my_function`. As always, this produces a new function object. Python then passes this function object to the function `my_decorator`. Remember that decorators, by definition, take callable objects as their only argument, and they are required to return a callable object as well.

After calling the decorator with the original function object, Python takes the return value from the decorator and binds it to the name of the original function. The end result is that the name `my_function` is bound to the result of calling `my_decorator` with the function created by the `def my_function` line.

In other words, decorators allow you to replace, enhance, or modify existing functions without changing those functions. Callers of the original function don't have to change their code, because the decorator mechanism ensures that the same name is used for both the decorated and undecorated function.

A first example

As with so many things in Python, a simple example is much more instructive than words. Suppose that we had some functions which returned strings and we wanted to ensure that these strings only contained ASCII characters.

We can use the built-in ascii() function to convert all non-ASCII characters to escape sequences, so one option would be to simply modify every function to use the ascii() function. This would work, but it isn't particularly scalable or maintainable; any change to the system would have to be made in many places, including if we decided to remove it completely.

A simpler solution is to create a decorator which does the work for us. This puts all of the logic in a single place. Here's the decorator:

```
def escape_unicode(f):
    def wrap(*args, **kwargs):
        x = f(*args, **kwargs)
        return ascii(x)

    return wrap
```

As you can see, the decorator, escape_unicode, is just a normal function. It's only argument, f, is the function to be decorated. The important part, really, is the local function wrap. This wrap function uses the *star-args* and *kw-args* idiom to accept any number of arguments. It then calls f — the argument to escape_unicode — with these arguments. wrap takes f's return value, converts non-ASCII characters to escape sequences, and returns the resulting string.

In other words, wrap behaves just like f except that it escapes non-ASCII characters, which is precisely what we want.

It's important to notice that escape_unicode returns wrap. Remember that a decorator takes a callable as its argument and returns a new callable. In this case, the new callable is wrap. By using closures, wrap is able to use the parameter f even after escape_unicode has returned.

Now that we have a decorator, let's create a function that might benefit from it. Our extremely simple function returns the name of a particular northern city:

```
def northern_city():
    return 'Tromsø'
```

And of course we can see that it works:

```
>>> print(northern_city())
'Tromsø'
```

To add unicode escaping to our function, we simply decorate `northern_city` with our `escape_unicode` decorator:

```
@escape_unicode
def northern_city():
    return 'Tromsø'
```

Now when we call `northern_city` we get see that, indeed, non-ASCII characters are converted to escape sequences:

```
>>> print(northern_city())
'Troms\\xf8'
```

This is a very simple example, but it demonstrates the most important elements of decorators. If you understand what's going on in this example, then you understand 90% of what there is to know about decorators.

What can be a decorator?

Classes as decorators

Now that we've seen how decorators work, let's look at how other kinds of callables can be used as decorators. We have just used a function as a decorator, and that's probably the most common form of decorator in general use. However, two other kinds of callable are also used fairly commonly.

The first of these is `class` objects. You'll recall that class objects are callable, and calling them produces new instances of that class. So by using a class object as a decorator you replace the decorated function with a new instance of the class. The decorated function will be passed to the constructor and thereby to `__init__()`. Recall, however, that the object returned by the decorator must itself be callable, so the decorator class must implement the `__call__()` method and thereby be callable.

In other words, we can use *class objects* as decorators so long as `__init__()` accepts a single argument (besides `self`) and the class implements `__call__()`.

In this example, we'll create a decorator class `CallCount` which keeps track of how many times it's called:

```
class CallCount:
    def __init__(self, f):
        self.f = f
        self.count = 0

    def __call__(self, *args, **kwargs):
        self.count += 1
        return self.f(*args, **kwargs)
```

`CallCount`'s initializer takes a single function `f` and keeps it as a member attribute. It also initializes a `count` attribute to zero. `CallCount`'s `__call__()` method then increments that count each time it's called and then calls `f`, returning whatever value `f` produces.

You use this decorator much as you might expect, by using `@CallCount` to decorate a function:

```
@CallCount
def hello(name):
    print('Hello, {}!'.format(name))
```

Now if we call `hello` a few times we can check its call count:

```
>>> hello('Fred')
Hello, Fred!
>>> hello('Wilma')
Hello, Wilma!
>>> hello('Betty')
Hello, Betty!
>>> hello('Barney')
Hello, Barney!
>>> hello.count
4
```

Great! Class decorators can be useful for attaching extra state to functions.

Instances as decorators

We've seen how to use class objects as decorators. Another common kind of decorator is a class *instance*. As you might have guessed, when you use a class instance as a decorator, Python calls that instance's `__call__()` method with the original function and uses `__call__()`s return value as the new function. These kinds of decorators are useful for creating collections of decorated functions which you can dynamically control in some way.

For example, let's define a decorator which prints some information each time the decorated function is called. But let's also make it possible to toggle this tracing feature by manipulating the decorator itself, which we'll implement as a class instance. First, here's a class:

```
class Trace:
    def __init__(self):
        self.enabled = True

    def __call__(self, f):
        def wrap(*args, **kwargs):
            if self.enabled:
                print('Calling {}'.format(f))
            return f(*args, **kwargs)
        return wrap
```

Remember that, unlike in our previous example, the class object itself is not the decorator. Rather, instances of `Trace` can be used as decorators. So let's create an instance of `Trace` and decorate a function with it:

```
tracer = Trace()

@tracer
def rotate_list(l):
    return l[1:] + [l[0]]
```

Now if we call `rotate_list` a few times, we can see that `tracer` is doing its job:

```
>>> l = [1, 2, 3]
>>> l = rotate_list(l)
Calling <function rotate_list at 0x101a349e0>
>>> l
[2, 3, 1]
>>> l = rotate_list(l)
Calling <function rotate_list at 0x101a349e0>
>>> l
[3, 1, 2]
>>> l = rotate_list(l)
Calling <function rotate_list at 0x101a349e0>
>>> l
[1, 2, 3]
```

We can now disable tracing simply by setting `tracer.enabled` to `False`:

```
>>> tracer.enabled = False
>>> l = rotate_list(l)
>>> l
[2, 3, 1]
>>> l = rotate_list(l)
>>> l
[3, 1, 2]
>>> l = rotate_list(l)
>>> l
[1, 2, 3]
```

The decorated function no longer prints out tracing information.

The ability to use functions, class objects, and class instances to create decorators gives you a lot of power and flexibility. Deciding which to use will depend a great deal upon what exactly you're trying to do. Experimentation and small examples are a great way to develop a better sense of how to design decorators.

Multiple decorators

In all of the examples we've seen so far we've used a single decorator to decorate each function. However, it's entirely possible to use more than one decorator at a time. All you need to do is list each decorator on a separate line above the function, each with its own `@` like this:

```
@decorator1
@decorator2
@decorator3
def some_function():
    ...
```

When you use multiple decorators, they are processed in reverse order. So in this example, `some_function` is first passed to `decorator3`. The callable returned by `decorator3` is then passed to `decorator2`; that is, `decorator2` is applied to the result of `decorator3` in precisely the same way that it would be applied to a "normal" function. Finally, `decorator1` is called with the result of `decorator2`. The callable returned by `decorator1` is ultimately bound to the name `some_function`. There's no extra magic going on, and the decorators involved don't need to know that they're being used with other decorators; this is part of the beauty of the decorator abstraction.

As an example, let's see how we can combine two decorators we've already seen, our tracer and our unicode escaper. First let's see the decorators again:

```
def escape_unicode(f):
    def wrap(*args, **kwargs):
        x = f(*args, **kwargs)
        return x.encode('unicode-escape').decode('ascii')

    return wrap

class Trace:
    def __init__(self):
        self.enabled = True

    def __call__(self, f):
        def wrap(*args, **kwargs):
            if self.enabled:
                print('Calling {}'.format(f))
            return f(*args, **kwargs)
        return wrap

tracer = Trace()
```

And now let's decorate a single function with both of these:

```
@tracer
@escape_unicode
def norwegian_island_maker(name):
    return name + 'øy'
```

Now when we use this to invent names for Norwegian islands, our non-ASCII characters will be properly escape and the tracer will record the call:

```
>>> from island_maker import norwegian_island_maker
>>> norwegian_island_maker('Llama')
Calling <function escape_unicode.<locals>.wrap at 0x101a3d050>
'Llama\\xf8y'
>>> norwegian_island_maker('Python')
Calling <function escape_unicode.<locals>.wrap at 0x101a3d050>
'Python\\xf8y'
>>> norwegian_island_maker('Troll')
Calling <function escape_unicode.<locals>.wrap at 0x101a3d050>
'Troll\\xf8y'
```

and, of course, we can disable the tracing without affecting the escaping:

```
>>> from island_maker import tracer
>>> tracer.enabled = False
>>> norwegian_island_maker('Llama')
'Llama\\xf8y'
>>> norwegian_island_maker('Python')
'Python\\xf8y'
>>> norwegian_island_maker('Troll')
'Troll\\xf8y'
```

Decorating methods

So far we've only seen decorators applied to functions, but it's entirely possible to decorate methods on classes as well. In general, there's absolutely no difference in how you use decorators for methods.

To see this, let's create a class version of our island maker function and use the `tracer` decorator on it:

```
class IslandMaker:
    def __init__(self, suffix):
        self.suffix = suffix

    @tracer
    def make_island(self, name):
        return name + self.suffix
```

We can use this to cross the North Sea and make a more British version of our island maker:

```
>>> im = IslandMaker(' Island')
>>> im.make_island('Python')
Calling <function IslandMaker.make_island at 0x101a34e60>
'Python Island'
>>> im.make_island('Llama')
Calling <function IslandMaker.make_island at 0x101a34e60>
'Llama Island'
```

As you can see, `tracer` works perfectly well with methods.

Decorators and function metadata

Decorators replace a function with another callable object, and we've seen a how this can be a powerful technique for adding functionality in a modular, maintainable way. There's a subtle problem, however, with how we've used decorators so far. By naively replacing a function with another callable, we lose important metadata about the original function, and this can lead to confusing results in some cases.

To see this, let's define an extremely simple function:

```
def hello():
    "Print a well-known message."
    print('Hello, world!')
```

Let's look at some attributes of this function in the REPL. First, it has an attribute called `__name__` which is simply the name of the function as the user defined it:

```
>>> hello.__name__
'hello'
```

Similarly, it has an attribute `__doc__` which is the docstring defined by the user:

```
>>> hello.__doc__
'Print a well-known message.'
```

You may not interact with these attributes a lot directly, but they are used by tools like debuggers and IDEs to display useful information about your objects. In fact, Python's built-in `help()` function uses these attributes:

```
>>> help(hello)
Help on function hello in module __main__:

hello()
    Print a well-known message.
```

So far so good. But let's see what happens when we use a decorator on our function. First let's define a simple no-op decorator:

```
def noop(f):
    def noop_wrapper():
        return f()

    return noop_wrapper
```

and decorate our `hello` function:

```
@noop
def hello():
    "Print a well-known message."
    print('hello world!')
```

All of the sudden, `help()` is a whole lot less helpful!:

```
>>> help(hello)
Help on function noop_wrapper in module __main__:

noop_wrapper()
```

Instead of telling us that `hello` is named “hello” and reporting the expected docstring, we’re seeing information about the wrapper function used by the `noop` decorator. If we look at `hello`’s `__name__` and `__doc__` attributes, we can see why:

```
>>> hello.__name__
'noop_wrapper'
>>> hello.__doc__
>>>
```

Since we’ve replaced the original `hello()` function with a new function, the `__name__` and `__doc__` attributes we get when we inspect `hello()` are those of the replacement function. This is an obvious result in retrospect, but it’s generally now what we want. Instead, we’d like the decorated function to have its original name and docstring.

Manually updating decorator metadata

Fortunately it’s very easy to get the behavior we want. We simply need to replace both the `__name__` and `__doc__` attributes of our `noop_wrapper()` function with the same attributes from the wrapped function. Let’s update our decorator to do this:

```
def noop(f):
    def noop_wrapper():
        return f()

    noop_wrapper.__name__ = f.__name__
    noop_wrapper.__doc__ = f.__doc__
    return noop_wrapper
```

Now when we examine our decorated function, we get the results we want:

```
>>> help(hello)
Help on function hello in module __main__:

hello()
    Print a well-known message.
```

This works, but it's a bit ugly. It would nice if there were a more concise way of creating "wrapper" function which properly inherited the appropriate attributes from the functions they wrap.

Updating decorator metadata with `functools.wraps`

We're in luck! The function `wraps()` in the `functools` package does precisely that. `functools.wraps()` is itself a decorator-factory (more on these later) which you apply to your wrapper functions. The `wraps()` function takes the function to be decorated as its argument, and it returns a decorator that does the hard work of updating the wrapper function with the wrapped function's attributes.

Here's how that looks:

```
import functools


def noop(f):
    @functools.wraps(f)
    def noop_wrapper():
        return f()

    return noop_wrapper
```

If we now look at our `hello()` function in the REPL one more time, we can see that, indeed, everything is as we want:

```
>>> help(hello)
Help on function hello in module __main__:

hello()
    Print a well-known message.

>>> hello.__name__
'hello'
>>> hello.__doc__
'Print a well-known message.'
```

As you start to develop your own decorators, it's probably best to use `functools.wraps()` to ensure that your decorated functions continue to behave as your users expect.

Closing thoughts on decorators

We've seen how to use and create decorators, and hopefully it's clear that decorators are a powerful tool for Python programming. They are being used widely in many popular Python packages, so it's very useful to be familiar with them. One word of warning, though: like many powerful features in many programming languages, it's possible to overuse decorators. Use decorators when they are the right tool: when the improve maintainability, add clarity, and simplify your code. If you find that you're using decorators just for the sake of using decorators, take a step back and think about whether they're really the right solution.

Validating arguments

One interesting and practical use of decorators is for validating function arguments. In many situations you want to ensure that function arguments are within a certain range or meet some other constraints.

Let's create a decorator which verifies that a given argument to a function is a non-negative number. This decorator is interesting in that it takes an argument. In fact, as hinted at earlier, we're actually creating a *decorator factory* here, not just a decorator. A decorator factory is a function that returns a decorator; the actual decorator is customized based on the arguments to the factory.

This might appear confusing at first, but you'll see how it works if you closely follow the description of decorators in the previous section:

```
# A decorator factory: it returns decorators
def check_non_negative(index):
    #  This is the actual decorator
    def validator(f):
        #  This is the wrapper function
        def wrap(*args):
            if args[index] < 0:
                raise ValueError(
                    'Argument {} must be non-negative.'.format(index))
            return f(*args)
        return wrap
    return validator
```

Here's how you can use this decorator to ensure that the second argument to a function is non-negative:

```
@check_non_negative(1)
def create_list(value, size):
    return [value] * size
```

We can see that it works as expected:

```
>>> create_list('a', 3)
['a', 'a', 'a']
>>> create_list(123, -6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "validator.py", line 6, in wrap
    'Argument {} must be non-negative.'.format(index))
ValueError: Argument 1 must be non-negative.
```

So how does this decorator work? Again, we need to recognize that `check_non_negative` is not, in fact, a decorator at all. A decorator is a callable object that takes a callable object as an argument and returns a callable object. `check_non_negative` takes an integer as an argument and returns a function, the nested `validator` function. What's going on here?

The key to understanding this is how we use `check_non_negative`. You'll see that, at the point where we decorate `create_list`, we actually *call* `check_non_negative`. In other words, the return value of `check_non_negative` is really the decorator! Python takes `check_non_negative`'s return value and passes our function `create_list` to it.

Indeed, if you look at the `validate` function by itself, you'll see that it looks exactly like the other decorators we've defined in this module. Interestingly, the `wrap` function returned by `validator` forms a closure over not just `f` — the decorated function — but also over `index` — the argument passed to `check_non_negative`.

This can be a bit of a mind-bender, and it's well worth spending a little extra time to make sure you really understand how this works. If you understand this example, you're well on your way to mastering Python decorators.

Summary

To close out this chapter, let's review what we've covered:

- Local functions:
  - `def` is executed at runtime
  - `def` defines functions in the scope in which it is called, and this can be inside other functions
  - Functions defined inside other functions are commonly called *local functions*
  - A new local function is created each time the containing function is executed
  - Local functions are no different from other local name bindings and can be treated like any other object
  - Local functions can access names in other scopes via the LEGB rule
  - The enclosing scope for a local function includes the parameters of its enclosing function
  - Local functions can be useful for code organization
  - Local functions are similar to lambdas, but are more general and powerful
  - Functions can return other functions, including local function defined in their body
- Closures:
  - Closures allow local functions to access objects from scopes which have terminated
  - Closures ensure that objects from terminates scopes are not garbage collected
  - Functions with closures have a special `__closure__` attribute
  - Local functions and closures are the keys to implementing *function factories* which are functions that create other functions
- Function decorators:
  - Function decorators are used to modify the behavior of existing functions without having to change them directly

- Decorators are callable objects which accept a single callable object as an argument and return a new callable object
- You use the `@` symbol to apply decorators to functions
- Decorators can enhance the maintainability, readability, and scalability of designs
- Decorators can be any kind of callable object. We looked specifically at functions, class objects, and class instances
- When class objects are used as decorators, the resulting callable is a new instance of that class
- When class instances are used as decorators, the result of their `__call__` method becomes the new callable
- Multiple decorators can be applied to a function
- When are multiple decorators are used, they are applied in reverse order
- Decorators are composable: they don't have to be specially designed to work with other decorators
- Class-methods can be decorated just like functions
- Decorators are a powerful tool, but make sure that you don't overuse them or use them unnecessarily
- Technically decorators never take any arguments except the callable that they decorate
  - To parameterize decorators, you need a decorator factory that creates decorators
- Local functions can create closures over objects in any number of enclosing scopes
- The `__name__` and `__doc__` attributes of decorated functions are actually those of their replacement function, which is not always what you want
- You can manually update the `__name__` and `__doc__` attributes of your wrapper functions
- The `functools.wraps()` function can be used to create well-behaved wrappers in a simple and clear manner

Chapter 4 - Properties and Class Methods

In this chapter we're going to look at a number of topics we've already covered — including decorators from the previous chapter — and see how we can use them to improve the design of our classes. First, though, we need to look at *class scope attributes* since they are the foundation for some of the techniques we'll cover.

Class attributes

You should already be familiar with instance attributes. These are attributes which are assigned on a per-object basis, usually in the `__init__()` method of a class.

To illustrate, we'll start with an object that defines a simple shipping container with a two *instance* attributes called `owner_code` and `contents`. We'll put the code in a Python module called `shipping.py`:

```
# shipping.py

class ShippingContainer:

    def __init__(self, owner_code, contents):
        self.owner_code = owner_code
        self.contents = contents
```

This is simple enough to use from the REPL:

```
>>> from shipping import *
>>> c1 = ShippingContainer("YML", "books")
>>> c1.owner_code
'YML'
>>> c1.contents
'books'
```

If we create a second shipping container instance, it has its own independent `owner_code` and `contents` attributes, as we would expect:

```
>>> c2 = ShippingContainer("MAE", "clothes")
>>> c2.contents
'clothes'
>>> c1.contents
'books'
```

Sometimes, however, we would like to have an attribute that is associated with the *class* and not with each *instance* of the class. In other words, we would like an attribute whose value is shared between all instances of that class. Such attributes are known as *class attributes*, and they can be created by assigning to their names within the scope of the class.

Adding class attributes

Let's say we'd like to give a new serial number to each `ShippingContainer` instance we create. We first add `next_serial` at class-scope, starting at the arbitrary value of 1337 we've chosen to make our example more interesting:

```
class ShippingContainer:

    next_serial = 1337

    # . . .
```

We also modify the initializer method to assign the current value of the `next_serial` *class* attribute to a new *instance* attribute, `self.serial`. We then increment the `next_serial` class attribute:

```
    def __init__(self, owner_code, contents):
        self.owner_code = owner_code
        self.contents = contents
        self.serial = next_serial
        next_serial += 1
```

Let’s try it in a new REPL session:

```
>>> from shipping import *
>>> c3 = ShippingContainer("MAE", "tools")
Traceback (most recent call last):
File "<console>", line 1, in <module>
File "shipping.py", line 8, in __init__
  self.serial = next_serial
UnboundLocalError: local variable 'next_serial' referenced before assignment
```

Referencing class attributes

As we can see, this didn’t work as planned. Python can’t resolve the `next_serial` name when we refer to it in the `__init__()` method. To understand why, we need to recall the Python rules for searching scopes – Local, Enclosing function, Global and Built-in – or LEGB.[31] Since `next_serial` doesn’t exist at *any* of these scopes[32], we need to locate an object that *is* in one of these scopes and drill down to `next_serial` from there. In this case the `ShippingContainer` class-object is at global (module) scope so we must start from there, by qualifying the `next_serial` class attribute name as `ShippingContainer.next_serial`.

Let’s go ahead and fix our class to fully qualify references to the class attribute:

[31] For more details on the LEGB rule, see Chapter 4 of The Python Apprentice.

[32] Remember that classes don’t introduce scopes.

```
class ShippingContainer:

    next_serial = 1337

    def __init__(self, owner_code, contents):
        self.owner_code = owner_code
        self.contents = contents
        self.serial = ShippingContainer.next_serial
        ShippingContainer.next_serial += 1
```

At first it might look odd to have to refer to the class by name from within the class definition, but it's really not much different to qualifying instance attributes with `self`. As with the `self` prefix, using the class name prefix for class attributes confers the same understandability advantage, reducing the amount of detective work required to figure out which objects are being referred to. Remember the Zen of Python – "explicit is better than implicit", and "readability counts".

With these changes in place, our example works as expected:

```
>>> from shipping import *
>>> c4 = ShippingContainer("ESC", electronics")
>>> c4.serial
1337
>>> c5 = ShippingContainer("ESC", pharmaceuticals")
>>> c5.serial
1338
>>> c6 = ShippingContainer("ESC", "noodles")
>>> c6.serial
1339
```

We can also retrieve the class attribute from outside the class by qualifying it with the class name:

```
>>> ShippingContainer.next_serial
1340
```

We can also access the same attribute through any of the instances:

```
>>> c5.next_serial
>>> 1340
>>> c6.next_serial
>>> 1340
```

Returning to our code, we could have written our `__init__()` function like this:

```
class ShippingContainer:

    next_serial = 1337

    def __init__(self, owner_code, contents):
        self.owner_code = owner_code
        self.contents = contents
        self.serial = self.next_serial
        self.next_serial += 1
```

Although it works, this style is best avoided since it makes it much less clear within the function body which attributes are instance attributes and which are class attributes.

Hiding class attributes with instance attributes

There's another pitfall here of which you must aware. Although you can *read* a class attribute through the `self` reference, attempting to *assign* to a class attribute through the `self` instance reference won't have the desired affect. Look at the other instance attributes we assign to in the initializer: `owner_code`, `contents`, and `serial`. Assigning to an instance attribute is exactly how we bring those attributes into being. If we attempt to assign to an existing class attribute through the `self` reference, we actually create a *new* instance attribute which hides the class attribute, and the class attribute would remain unmodified.

You might think that use of the augmented assignment operators, such as the plus-equals we use here, would also be verboten, but they are not. The augmented assignment operators work by calling a special method on the referred-to object and don't rebind the reference on their left-hand side.

All said, it's much better and safer to access class attributes as, well, attributes of the class object, rather than via the instance.

Static methods

Let’s perform a small refactoring by extracting the logic for obtaining the next serial number into the method `_get_next_serial()` which, as you can see from the leading underscore, is an implementation detail of this class:

```python
class ShippingContainer:

    next_serial = 1337

    def _get_next_serial(self):
        result = ShippingContainer.next_serial
        ShippingContainer.next_serial += 1
        return result

    def __init__(self, owner_code, contents):
        self.owner_code = owner_code
        self.contents = contents
        self.serial = self._get_next_serial()
```

Notice that, like all of the other methods we have encountered so far, the first argument to `_get_next_serial()` is `self` which is the instance on which the method will operate. Notice, however, that although we must accept the `self` instance argument, nowhere in the method do we actually refer to `self`, so it seems completely redundant. What we would like to do is associate `_get_next_serial()` with the *class* rather than with *instances* of the class. Python gives us two mechanisms to achieve this, the first of which is the `@staticmethod` decorator.

The `@staticmethod` decorator

To convert our method to a static method, we decorate it with `@staticmethod` and remove the unused `self` argument:

```
class ShippingContainer:

    next_serial = 1337

    @staticmethod
    def _get_next_serial():
        result = ShippingContainer.next_serial
        ShippingContainer.next_serial += 1
        return result

    def __init__(self, owner_code, contents):
        self.owner_code = owner_code
        self.contents = contents
        self.serial = ShippingContainer._get_next_serial()
```

Although not strictly necessary, we can also modify the call site to call through the *class* rather than through the *instance*, by replacing `self._get_next_serial()` with `ShippingContainer._get_next_serial()`.

This code has exactly the same behaviour as before:

```
>>> from shipping import *
>>> c6 = ShippingContainer("YML", "coffee")
>>> c6.serial
1337
>>> ShippingContainer.next_serial
1338
```

Static methods in Python have no direct knowledge of the class within which they are defined. They allow us to group a function within the class because the function is conceptually related to the class.

> The name `@staticmethod` is something of an anachronism in Python. The "static" refers to a the keyword used to indicate the equivalent concept in the C++ programming language, which itself was a reuse of a keyword from the C programming language!

Class methods

As an alternative to `@staticmethod`, we can use a different decorator called `@classmethod` which passes the *class-object* as a the first formal parameter to the function. By convention we call this parameter `cls` since we can't use the fully spelled out keyword `class` as a parameter name. The `cls` parameter for class-methods plays an analagous role to the `self` parameter for instance-methods: it refers to the class object to which the function is bound.

Let's further modify our function to use `@classmethod` decorator instead of the `@staticmethod` decorator:

```
class ShippingContainer:

    next_serial = 1337

    @classmethod
    def _get_next_serial(cls):
        result = cls.next_serial
        cls.next_serial += 1
        return result

    def __init__(self, owner_code, contents):
        self.owner_code = owner_code
        self.contents = contents
        self.serial = ShippingContainer._get_next_serial()
```

Now when we call `ShippingContainer._get_next_serial()` the `ShippingContainer` class object is passed as the `cls` argument of the class-method. We then refer to `cls` within the body of the method to locate the `next_serial` class attribute.

Choosing between `@staticmethod` and `@classmethod`

The `@staticmethod` and `@classmethod` decorators are quite similar, and you may find it difficult to choose between them. This may be even more confusing if you have a heritage in another object-oriented language such as C++, C#, or Java which has a similar static-method concept. The rule is simple though. If you need to refer to the class object within the method (for example to access a class attribute) prefer to use `@classmethod`. If you don't need to access the class object use `@staticmethod`.

In practice, most static methods will be internal implementation details of the class, and they will be marked as such with a leading underscore. Having no access to either the class object or the instance object, they rarely form a useful part of the class interface.

In principle, it would also be possible to implement any `@staticmethod` completely outside of the class at module scope without any loss of functionality — so you may want to consider carefully whether a particular function should be a module scope function or a static method. The `@staticmethod` decorator merely facilitates a particular organisation of the code allowing us to place what could otherwise be free functions within classes.

Named constructors

Sometimes we would like a class to support 'named constructors' — also known as factory functions — which construct objects with certain configurations. We can do this with class-methods.

For example, we could use a factory function to implement a method which creates an empty shipping container:

```python
class ShippingContainer:

    next_serial = 1337

    @classmethod
    def _get_next_serial(cls):
        result = cls.next_serial
        cls.next_serial += 1
        return result

    @classmethod
    def create_empty(cls, owner_code):
        return cls(owner_code, contents=None)

    def __init__(self, owner_code, contents):
        self.owner_code = owner_code
        self.contents = contents
        self.serial = ShippingContainer._get_next_serial()
```

We invoke this class-method on the `ShippingContainer` class object like this:

```
>>> from shipping import *
>>> c7 = ShippingContainer.create_empty("YML")
>>> c7
<shipping.ShippingContainer object at 0x10276ea10>
>>> c7.contents
>>>
```

This technique allows us to support multiple “constructors” with different behaviours without having to resort to contortions in the `__init__()` method to interpret different forms of argument lists. Here we add a constructor for placing an iterable series of items in the container:

```
class ShippingContainer:

    next_serial = 1337

    @classmethod
    def _get_next_serial(cls):
        result = cls.next_serial
        cls.next_serial += 1
        return result

    @classmethod
    def create_empty(cls, owner_code):
        return cls(owner_code, contents=None)

    @classmethod
    def create_with_items(cls, owner_code, items):
        return cls(owner_code, contents=list(items))

    def __init__(self, owner_code, contents):
        self.owner_code = owner_code
        self.contents = contents
        self.serial = ShippingContainer._get_next_serial()
```

We can use this new constructor like so:

```
>>> from shipping import *
>>> c8 = ShippingContainer.create_with_items("MAE", ['food', 'textiles', 'minerals'])
>>> c8
<shipping.ShippingContainer object at 0x102752850>
>>> c8.contents
['food', 'textiles', 'minerals']
```

Moving to BIC codes

Let's modify our example to make it slightly more realistic. We'll adjust ShippingContainer to use a string code rather than an integer serial number. In fact, we'll modify our class to use fully-fledged BIC codes.[33] Each container has a unique BIC code which follows a standard format defined in the ISO 6346 standard. We won't go into the details of the coding system here, but we have included a simple Python module called iso6346.py in Appendix A. All we need to know for now is that the module can create a conforming BIC code given a three-letter owner code and a six digit serial number, together with an optional *equipment category identifier*.

We'll retain the integer serial number generator and introduce a static method called _make_bic_code() to combine the owner code and integer serial number into a a single string BIC code. This new method will delegate much of its work to to the iso6346 module. We'll also rework the initializer function to create and store the BIC code instead of the the separate owner code and serial numbers:

```
class ShippingContainer:

  next_serial = 1337

  @staticmethod
  def _make_bic_code(owner_code, serial):
      return iso6346.create(owner_code=owner_code,
                            serial=str(serial).zfill(6))

  @classmethod
  def _get_next_serial(cls):
      result = cls.next_serial
      cls.next_serial += 1
      return result
```

[33] BIC is the *Bureau International des Conteneurs* (International Container Bureau)

```
    @classmethod
    def create_empty(cls, owner_code):
        return cls(owner_code, contents=None)

    @classmethod
    def create_with_items(cls, owner_code, items):
        return cls(owner_code, contents=list(items))

    def __init__(self, owner_code, contents):
        self.contents = contents
        self.bic = ShippingContainer._make_bic_code(
            owner_code=owner_code,
            serial=ShippingContainer._get_next_serial())
```

Let's try the modified code:

```
>>> from shipping import *
>>> c = ShippingContainer.create_empty('YML')
>>> c.bic
'YMLU0013374'
```

Overriding static- and class-methods

We'll return to class inheritance in more depth later in chapter 8, but for now we'll look at how class and static methods behave in the presence of inheritance.

Static methods with inheritance

Unlike static methods in many other languages, static methods in Python can be overridden in subclasses. Let's introduce a subclass of `ShippingContainer` called `RefrigeratedShippingContainer`:[34]

[34] In BIC codes the fourth character specifies the "equipment category". The default category is 'U', but refrigerated shipping containers use an equipment category of 'R'. We must specify this when creating the BIC code by passing an additional `category` argument to the `iso6346.create()` function. We do this in the overriden `_make_bic_code()` static method in the derived class.

```
class RefrigeratedShippingContainer(ShippingContainer):

  @staticmethod
  def _make_bic_code(owner_code, serial):
      return iso6346.create(owner_code=owner_code,
                            serial=str(serial).zfill(6),
                            category='R')
```

Let's try instantiating our new class and checking its BIC code:

```
>>> from shipping import *
>>> r1 = RefrigeratedShippingContainer("MAE", 'fish')
>>> r1.bic
'MAEU0013374'
```

This hasn't worked as we had hoped. The fourth character in the BIC code is still 'U'. This is because in `ShippingContainer.__init__()` we have called `_make_bic_code()` through a specific *class*. To get polymorphic override behaviour we need to call the static method on an *instance*. Let's experiment a little at the REPL so we understand what's going on. First we'll test the static method by calling directly on the base class:

```
>>> ShippingContainer._make_bic_code('MAE', 1234)
'MAEU0012349'
```

And then directly on the derived class:

```
>>> RefrigeratedShippingContainer._make_bic_code('MAE', 1234)
'MAER0012347'
```

Now we have an 'R' for refrigeration for the fourth character. If you're wondering why the last digit also changes from a nine to a four, it's because the last digit is a check-digit computed by the ISO 6346 implementation.

In both cases we get exactly what we have asked for. The class-specific versions of the static methods are called. Now we'll create some instances. First off, the base class:

```
>>> c = ShippingContainer('ESC', 'textiles')
>>> c._make_bic_code('MAE', 1234)
'MAEU0012349'
```

Here the the fourth character of the result is the default 'U', so we know the base version was called. Notice that although we've created an instance, we're ignoring any instance attribute data when we invoke the static method directly in this way; we deliberately used different owner codes to make this clear.

Now we'll instantiate the derived class:

```
>>> r = RefrigeratedShippingContainer('ESC', 'peas')
>>> r._make_bic_code('MAE', 1234)
'MAER00123
```

We can see from the 'R' in fourth place that the derived class implementation is called, so we can get polymorphic dispatch of static methods *only* when we call the method through an instance, not when we call the method through the class. To get the desired behaviour, we must modify `__init__()` in the base class to use polymorphic dispatch of the static method by calling through the instance, `self`:

```
  def __init__(self, owner_code, contents):
      self.contents = contents
      self.bic = self._make_bic_code(
          owner_code=owner_code,
          serial=ShippingContainer._get_next_serial())
```

With this change in place, we get polymorphic BIC code generation from the single constructor implementation:

```
>>> from shipping import *
>>> r2 = RefrigeratedShippingContainer('MAE', 'fish')
>>> r2.bic
'MAER0013370'
```

Be aware then, that by calling static methods through the class, you effectively prevent them being overridden, at least from the point of view of the base class. If you need polymorphic dispatch of static method invocations, call through the `self` instance!

Class methods with inheritance

The class methods we defined in the base class will be inherited by the subclass, and what is more, the `cls` argument of these methods will be set appropriately, so calling `create_empty()` on `RefrigeratedShippingContainer` will create an object of the appropriate subtype:

```
>>> from shipping import *
>>> r1 = RefrigeratedShippingContainer.create_empty("YML")
>>> r1
<shipping.RefrigeratedShippingContainer object at 0x102f6e990>
```

For those of you coming to Python from other popular object-oriented languages you should recognise this ability to have class methods behave polymorphically as a distinguishing feature of Python. The other factory method also works as expected. These invocations work because the base class `__init__()` initializer method is inherited into the subclass:

```
>>> r2 = RefrigeratedShippingContainer.create_with_items("YML", ["ice", "peas"])
>>> r2
<shipping.RefrigeratedShippingContainer object at 0x102f6e3d0>
>>> r2.contents
['ice', 'peas']
```

Adding temperature to refrigerated containers

Let's move on by making our refrigerated shipping container more interesting by adding a per-container temperature setting as an instance attribute. First we'll add a class attribute which defines the maximum temperature of a refrigerated container:

```
class RefrigeratedShippingContainer(ShippingContainer):

    MAX_CELSIUS = 4.0

    # ...
```

Next we'll need to override `__init__()` in the subclass. This overridden method does two things. First it calls the base-class version of `__init__()`, forwarding the `owner_code` and `contents` arguments to the base class initializer.

Unlike other object oriented languages where constructors at every level in an inheritance hierarchy will be called automatically, the same cannot be said for initializers in Python. If we want a base class initializer to be called when we override that initializer in a derived class, we must do so explicitly. Remember, explicit is better than implicit.

Using `super()` to call the base class initializer

To get a reference to the base class instance we call the built-in `super()` function. We then call `__init__()` on the returned reference and forward the constructor arguments. We'll be covering `super()` in a lot of detail in chapter 8, so don't concern yourself with it now — we're using it so the subclass version of `__init__()` can extend the base class version.

This done, we validate the `celsius` argument and assign the `celsius` instance attribute:

```
class RefrigeratedShippingContainer(ShippingContainer):

    MAX_CELSIUS = 4.0

    @staticmethod
    def _make_bic_code(owner_code, serial):
        return iso6346.create(owner_code=owner_code,
                              serial=str(serial).zfill(6),
                              category='R')

    def __init__(self, owner_code, contents, celsius):
        super().__init__(owner_code, contents)
        if celsius > RefrigeratedShippingContainer.MAX_CELSIUS:
            raise ValueError("Temperature too hot!")
        self.celsius = celsius
```

Let's try it:

```
>>> from shipping import *
>>> r3 = RefrigeratedShippingContainer.create_with_items('ESC', ['broccoli', 'caulifl\
ower', 'carrots'])
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "shipping.py", line 25, in create_with_items
  return cls(contents=list(items))
TypeError: __init__() missing 1 required positional argument: 'celsius'
```

Oops! There's no way the factory methods in the base class can know — or indeed should know — the signature of the `__init__()` function in derived classes, so they can't accommodate our extra `celsius` argument.

Using `*args` and `**kwargs` to accommodate extra arguments in the base class

Fortunately, we can use star-args and keyword-args to work around this. By having our factory functions accept both `*args` and `**kwargs` and forward them unmodified to the underlying constructors, we can have our base class factory functions accept arguments destined for derived class initializers:

```
class ShippingContainer:

    next_serial = 1337

    @staticmethod
    def _make_bic_code(owner_code, serial):
        return iso6346.create(owner_code=owner_code,
                              serial=str(serial).zfill(6))

    @classmethod
    def _get_next_serial(cls):
        result = cls.next_serial
        cls.next_serial += 1
        return result

    @classmethod
    def create_empty(cls, owner_code, *args, **kwargs):
        return cls(owner_code, contents=None, *args, **kwargs)
    @classmethod
    def create_with_items(cls, owner_code, items, *args, **kwargs):
        return cls(owner_code, contents=list(items), *args, **kwargs)

    def __init__(self, owner_code, contents):
        self.contents = contents
        self.bic = self._make_bic_code(
            owner_code=owner_code,
            serial=ShippingContainer._get_next_serial())
```

This works as expected:

```
>>> from shipping import *
>>> r3 = RefrigeratedShippingContainer.create_with_items('ESC', ['broccoli', 'caulifl\
ower', 'carrots'], celsius=2.0)
>>> r3
<shipping.RefrigeratedShippingContainer at 0x104120fd0>
>>> r3.contents
['broccoli', 'cauliflower', 'carrots']
>>> r3.celsius
2.0
>>> r3.bic
'ESCR0013370'
```

So far so good. We can construct instances of our derived class using a factory function defined in the base class and can gain access to our new `celsius` attribute as expected. Unfortunately, our design also ignores the constraint defined by the `MAX_CELSIUS` class attribute:

```
>>> r3.celsius = 12
```

The point of `MAX_CELSIUS` is that the temperature in a refrigerated container should never rise about that value. Setting the temperature to 12 violates a class invariant, and we need to find a way to prevent that.

Properties

Using the Python tools we already have at our disposal, one approach would be to rename `celsius` to `_celsius` in order to discourage meddling. We would then wrap the attribute with two methods called `get_celsius()` and `set_celsius()`, with the setter performing validation against `MAX_CELSIUS`. Such an approach would work, but would be considered deeply un-Pythonic! Furthermore, it would require all uses of the `celsius` attribute to be adjusted to use the new methods.

Fortunately, Python provides an altogether superior alternative to getter and setter methods called *properties*. Properties allow getters and setters to be exposed as seemingly regular attributes, permitting a graceful upgrade in capabilities. As with static- and class-methods, decorators are the basis of the property system. Let's take a look.

Defining a read-only property

First, we'll rename `celsius` to `_celsius` to indicate that it's no longer to be considered part of the public interface:

```
class RefrigeratedShippingContainer(ShippingContainer):

    MAX_CELSIUS = 4.0

    @staticmethod
    def _make_bic_code(owner_code, serial):
        return iso6346.create(owner_code=owner_code,
                              serial=str(serial).zfill(6),
                              category='R')

    def __init__(self, owner_code, contents, celsius):
        super().__init__(owner_code, contents)
        if celsius > RefrigeratedShippingContainer.MAX_CELSIUS:
            raise ValueError("Temperature too hot!")
        self._celsius = celsius
```

Then we'll define a new method `celsius()` which will retrieve the attribute. The method will be decorated with the built-in `@property` decorator:

```
    @property
    def celsius(self):
        return self._celsius
```

Back in the REPL, let's re-import our module and instantiate a new `RefrigeratedShippingContainer` with a suitable temperature. You'll see that, even though we've defined `celsius` as a method, we can access it as though it were a simple attribute:

```
>>> from shipping import *
>>> r4 = RefrigeratedShippingContainer.create_with_items('YML', ['fish'], celsius=-18\
.0)
>>> r4.celsius
-18.0
```

What has happened here is that the `@property` decorator has converted our `celsius()` *method* into something that behaves like an attribute when accessed. The details of exactly how this is achieved are beyond the scope of this book;[35] for the time being it's sufficient to understand that `@property` can be used to transform getter methods so they can be called as if they were attributes.

Creating a read-write property

If we attempt to assign to `celsius` we'll receive an `AttributeError` informing us that the attribute can't be set:

```
>>> r4.celsius = -5.0
Traceback (most recent call last):
  File "<console>", line 1, in <module>
    r4.celsius = -5.0
AttributeError: can't set attribute
```

To make assignment to properties work we need to define a *setter*. This uses another decorator, but first we need to cover some background information.

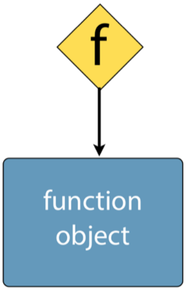
Review: the mechanics of decorators

Recall that decorators are functions which accept one function as an argument and return another object (which is usually a wrapper around the original function) which modifies its behaviour. Here, we show a regular function which is bound by the `def` statement to the name `f` and then processed by a decorator:

[35] For the details, see the discussion on *Descriptors* in chapter 4 in our book The Python Master

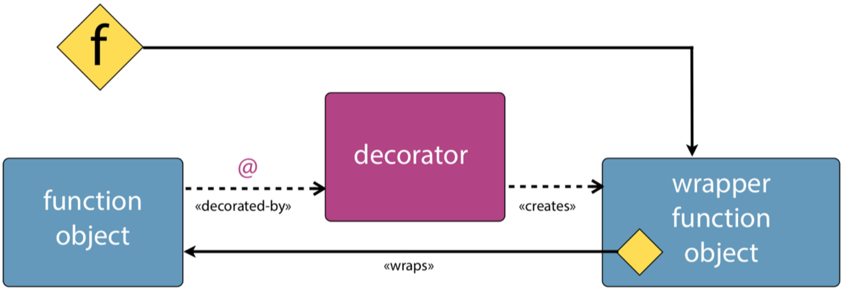
```
@decorator
def f():
    do_something()
```

Prior to decorating `f` the name assignment looks like this:



Initial name assignment

When Python processes this, it passes the function `f` to the callable `decorator`. Python then binds the return value of that call to the name `f`. Most of the time this new `f` will retain a reference to the original function:



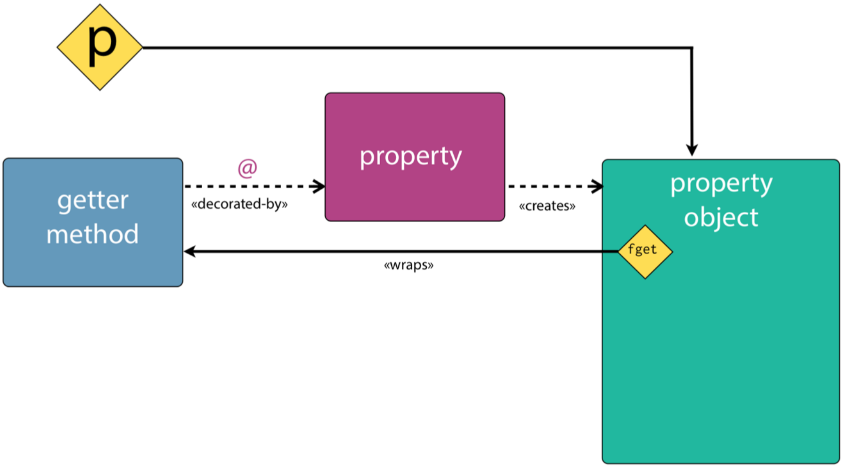
Name assignment after decoration

Using decorators to create properties

Moving on to the specifics of the properties, we'll start with an `Example` class into which we place a *getter* function `p()`:

```
class Example:
    @property
    def p(self):
        return self._p
```

We decorate this with the built-in `@property` decorator which creates a special *property object*, which contains a reference back to the original getter function. The `p` name is rebound to the property object. This much we've already seen with our `celsius` property. It looks something like this:



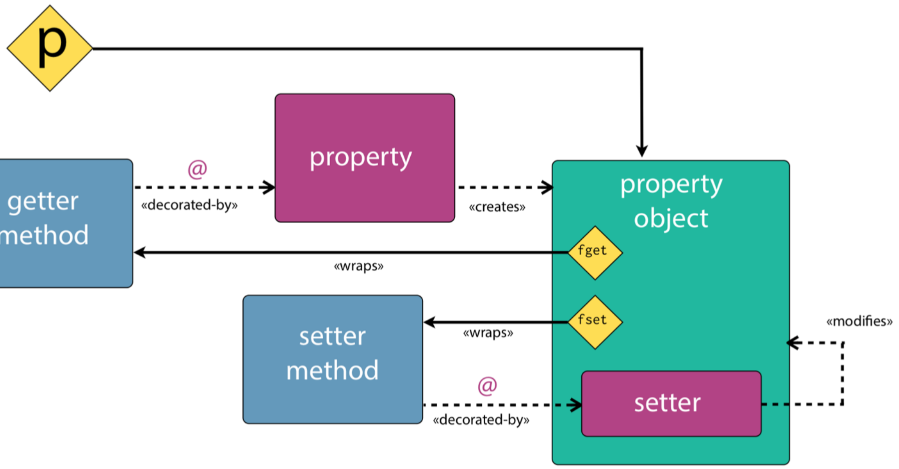
A read-only property

If needed, we can then create a *setter* method. This can also be called simply `p`, although this will also need to be decorated. This time, rather than the built-in `@property` decorator, we use an attribute of the object that was created when we defined the *getter*. This new decorator is always called `setter` and must be accessed via the property object, so it our case it's called `p.setter`:

```
class Example:
    @property
    def p(self):
        return self._p

    @p.setter
    def p(self, value):
        self._p = value
```

Decorating our *setter* function with the `p.setter` decorator causes the property object to be modified, associating it with our *setter* method in addition to the *getter* method. The resulting object structure looks like this:



A read-write property

Making `celsius` into a read-write property

When we decorate our `celsius` getter with `@property` the returned object is also bound to the name `celsius`. It is this returned property object which has the `setter` attribute attached to it. `celsius.setter` *is itself another decorator*, and we use it to decorate our setter definition.

This is all fairly mind-bending and we apologise if you've not yet consumed enough caffeine today for this to make sense. As usual, an example will clarify matters somewhat. Let's define a setter:

```
  @celsius.setter
  def celsius(self, value):
      if value > RefrigeratedShippingContainer.MAX_CELSIUS:
          raise ValueError("Temperature too hot!")
      self._celsius = value
```

We can now assign to the property using regular attribute syntax, and this will call the setter method and execute our validation code:

```
>>> from shipping import *
>>> r5 = RefrigeratedShippingContainer.create_with_items('YML', ['prawns'], celsius=-\
18.0)
>>> r5.celsius
-18.0
>>> r5.celsius = -19.0
>>> r5.celsius
-19.0
>>> r5.celsius = 5.0
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "shipping.py", line 42, in celsius
  raise ValueError("Temperature too hot!")
ValueError: Temperature too hot!
```

Adding a property for Fahrenheit temperatures

Shipping containers are moved around the world, between cultures which prefer the Celsius measurement scale and those which prefer the Fahrenheit scale. Let’s round off this section by adding support for Fahrenheit property access to the same underlying temperature data.

Here’s the full code for the revised class:

```
class RefrigeratedShippingContainer(ShippingContainer):

    MAX_CELSIUS = 4.0

    @staticmethod
    def _make_bic_code(owner_code, serial):
        return iso6346.create(owner_code=owner_code,
                              serial=str(serial).zfill(6),
                              category='R')

    @staticmethod
    def _c_to_f(celsius):
        return celsius * 9/5 + 32

    @staticmethod
    def _f_to_c(fahrenheit):
        return (fahrenheit - 32) * 5/9

    def __init__(self, owner_code, contents, celsius):
        super().__init__(owner_code, contents)
        if celsius > RefrigeratedShippingContainer.MAX_CELSIUS:
            raise ValueError("Temperature too hot!")
        self._celsius = celsius

    @property
    def celsius(self):
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        if value > RefrigeratedShippingContainer.MAX_CELSIUS:
            raise ValueError("Temperature too hot!")
        self._celsius = value

    @property
    def fahrenheit(self):
        return RefrigeratedShippingContainer._c_to_f(self.celsius)

    @fahrenheit.setter
    def fahrenheit(self, value):
        self.celsius = RefrigeratedShippingContainer._f_to_c(value)
```

Notice that we've add two static methods `_c_to_f()` and `_f_to_c()` to perform temperature conversions. These are good candidates for static methods since they don't depend on the instance or class objects but don't really belong at global scope in a module of shipping container classes either.

The getter and setter methods for our new `fahrenheit` property are implemented in terms of our new temperature conversion static methods. Significantly, rather than going directly to the stored `_celsius` attribute, these new methods are also implemented in terms of the existing `celsius` property. This is so we can reuse the validation logic in the existing property.

Using properties for class-wide validation

Finally, notice that we can simplify our subclass initializer by leaning on the `celsius` property setter validation here, too. We assign through the property rather than directly to the attribute and get validation for free:

```
  def __init__(self, owner_code, contents, celsius):
      super().__init__(owner_code, contents)
      self.celsius = celsius
```

Let's test these changes at the REPL:

```
>>> from shipping import *
>>> r6 = RefrigeratedShippingContainer.create_empty('YML', celsius=-20)
>>> r6.celsius
-20
>>> r6.fahrenheit
-4.0
>>> r6.fahrenheit = -10.0
>>> r6.celsius
-23.333333333333332
>>>
>>> r7 = RefrigeratedShippingContainer.create_empty('MAE', celsius=7.0)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
    r7 = RefrigeratedShippingContainer.create_empty('MAE', celsius=7.0)
  File "shipping.py", line 21, in create_empty
    return cls(owner_code, contents=None, *args, **kwargs)
  File "shipping.py", line 54, in __init__
    self.celsius = celsius
  File "shipping.py", line 63, in celsius
    raise ValueError("Temperature too hot!")
ValueError: Temperature too hot!
```

Thoughts on Properties in Object Oriented Design

Python properties provide a graceful upgrade path from public instance attributes to more encapsulated data wrapped by getters and setters. Bear in mind, though, that overuse of getters and setters can lead to poor object-oriented designs with tightly coupled classes. Such designs expose too many details, albeit thinly wrapped in properties. In general we recommend reducing coupling between objects by implementing methods which allow clients to *tell* objects what to do, rather than having clients request internal data so they can themselves perform actions.

Overriding properies

We'll now modify our `ShippingContainer` class to contain information about container sizes. Since the width and height are the same for all shipping containers, we'll implement those as class attributes. Since length can vary between containers we'll implement that as an instance attribute:

```
class ShippingContainer:

    HEIGHT_FT = 8.5
    WIDTH_FT = 8.0

    next_serial = 1337

    @staticmethod
    def _make_bic_code(owner_code, serial):
        return iso6346.create(owner_code=owner_code,
                              serial=str(serial).zfill(6))

    @classmethod
    def _get_next_serial(cls):
        result = cls.next_serial
        cls.next_serial += 1
        return result
    @classmethod
    def create_empty(cls, owner_code, length_ft, *args, **kwargs):
        return cls(owner_code, length_ft, contents=None, *args, **kwargs)

    @classmethod
    def create_with_items(cls, owner_code, length_ft, items, *args, **kwargs):
        return cls(owner_code, length_ft, contents=list(items), *args, **kwargs)

    def __init__(self, owner_code, length_ft, contents):
        self.contents = contents
        self.length_ft = length_ft
        self.bic = self._make_bic_code(
            owner_code=owner_code,
            serial=ShippingContainer._get_next_serial())
```

We'll add a read-only property which reports the volume in cubic feet of a container instance, making the simplifying assumption that the sides of the container have zero thickness:

```
  @property
  def volume_ft3(self):
      return ShippingContainer.HEIGHT_FT * ShippingContainer.WIDTH_FT * self.length_ft
```

Notice that the height and width are qualified with the class object, and the length uses the instance object. Constructing an empty 20-foot container, we can now determine that it has a volume of 1360 cubic feet:

```
>>> c = ShippingContainer.create_empty('YML', length_ft=20)
>>> c.volume_ft3
1360.0
```

We also need to modify the constructor of `RefrigeratedShippingContainer` to accept the new `length_ft` argument, like so:

```
  def __init__(self, owner_code, length_ft, contents, celsius):
      super().__init__(owner_code, length_ft, contents)
      self.celsius = celsius
```

Once we've done this the `volume_ft3` property is inherited into the `RefrigeratedShippingContainer` without issue:

```
>>> from shipping import *
>>> r = RefrigeratedShippingContainer.create_empty('YML', length_ft=20, celsius=-10.0)
>>> r.volume_ft3
1360
```

Overriding getters in subclasses

We know, however, that the cooling machinery in a `RefrigeratedShippingContainer` occupies 100 cubic feet of space, so we should subtract that from the total. Let's add a class attribute for the cooler's volume and override the `volume_ft3` property with the modified formula:

```
class RefrigeratedShippingContainer:

  # ...

  FRIDGE_VOLUME_FT3 = 100

  @property
  def volume_ft3(self):
      return (self.length_ft
              * ShippingContainer.HEIGHT_FT
              * ShippingContainer.WIDTH_FT
              - RefrigeratedShippingContainer.FRIDGE_VOLUME_FT3)
```

This works well enough:

```
>>> from shipping import *
>>> r = RefrigeratedShippingContainer.create_empty('YML', length_ft=20, celsius=-10.0)
>>> r.volume_ft3
1260.0
```

However, we've now duplicated the bulk volume calculation between the overridden property and its base class implementation. We'll address that by having the derived class version delegate to the base class. As before, this is done by using `super()` to retrieve the base-class property:

```
  @property
  def volume_ft3(self):
      return super().volume_ft3 - RefrigeratedShippingContainer.FRIDGE_VOLUME_FT3
```

So overriding property *getters* like `volume_ft3` is straightforward enough. We redefine the property in the derived class as normal, delegating to the base class if we need to.

Overriding setters in subclasses

Unfortunately, overriding property *setters* is more involved. To see why, we'll need a property for which it makes sense to override the setter. We'll introduce a third class into our class hierarchy, one for a `HeatedRefrigeratedShippingContainer`.[36] For the purposes of this exercise we'll assume that such containers should never fall below a fixed temperature of -20 celsius, which we'll represent with another class attribute:

```
class HeatedRefrigeratedShippingContainer(RefrigeratedShippingContainer):

    MIN_CELSIUS = -20.0
```

We don't need to override the `celsius` *getter* here, but we do need to override the `celsius` *setter*. Let's have a go:

```
  @celsius.setter
  def celsius(self, value):
      if not (HeatedRefrigeratedShippingContainer.MIN_CELSIUS
              <= value
              <= RefrigeratedShippingContainer.MAX_CELSIUS):
          raise ValueError("Temperature out of range")
      self._celsius = value
```

Unfortunately, this "obvious" approach doesn't work. The `celsius` object from which we retrieve the `setter` decorator is not visible in the scope of the derived class:

[36] We're not making this up — such things do exist, and their purpose is to maintain a temperatures within a wide range of ambient conditions.

```
>>> from shipping import *
Traceback (most recent call last):
  File "<console>", line 1, in <module>
    from shipping import *
  File "shipping.py", line 89, in <module>
    class HeatedRefrigeratedShippingContainer(RefrigeratedShippingContainer):
  File "shipping.py", line 93, in HeatedRefrigeratedShippingContainer
    @celsius.setter
NameError: name 'celsius' is not defined
```

We can solve this by fully qualifying the name of the `celsius` object with the base class name:

```
  @RefrigeratedShippingContainer.celsius.setter
  def celsius(self, value):
      if not (HeatedRefrigeratedShippingContainer.MIN_CELSIUS
              <= value
              <= RefrigeratedShippingContainer.MAX_CELSIUS):
          raise ValueError("Temperature out of range")
```

Now this works very well. We can create instances of the new class through our existing named constructor:

```
>>> from shipping import *
>>> h1 = HeatedRefrigeratedShippingContainer.create_empty('YML', length_ft=40, celsiu\
s=-18.0)
>>> h1
<shipping.HeatedRefrigeratedShippingContainer at 0x104163b90>
```

Any attempt to set a temperature below the minimum via the overridden property causes the `ValueError` to be raised:

```
>>> h1.celsius = -21.0
Traceback (most recent call last):
  File "<console>", line 1, in <module>
    h.celsius = -21.0
  File "shipping.py", line 98, in celsius
    raise ValueError("Temperature out of range")
ValueError: Temperature out of range
```

Similarly, attempting to construct an instance with an out-of-range temperature fails as well, even through we haven't even defined an `__init__()` method for the new class. Recall that the initializer assigns to the underlying `_celsius` attribute through the `celsius` property, so our overridden property validator is invoked during construction too, thanks to polymorphic dispatch:

```
>>> h2 = HeatedRefrigeratedShippingContainer.create_empty('YML', length_ft=40, celsiu\
s=-25.0)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
    h2 = HeatedRefrigeratedShippingContainer.create_empty('YML', length_ft=40, celsiu\
s=-25.0)
  File "shipping.py", line 24, in create_empty
    return cls(owner_code, length_ft, contents=None, *args, **kwargs)
  File "shipping.py", line 64, in __init__
    self.celsius = celsius
  File "shipping.py", line 98, in celsius
    raise ValueError("Temperature out of range")
ValueError: Temperature out of range
```

Reducing code duplication further

Our overridden property is interesting because it highlights the useful ability of the Python language to chain the relational operators. This means we can do `a < b < c` rather than `(a < b) and (b < c)`. That said, it is subtly violating the DRY [37] principle by duplicating the comparison with `MAX_CELSIUS` which is already implemented in the parent class. We could try to eliminate the duplication by delegating the test to the superclass, via `super()` like this:

[37] Short for **D**on't **R**epeat **Y**ourself

```
  @RefrigeratedShippingContainer.celsius.setter
  def celsius(self, value):
      if value < HeatedRefrigeratedShippingContainer.MIN_CELSIUS:
          raise ValueError("Temperature too cold!")
      super().celsius = value
```

But surprisingly, that doesn't work! We get the runtime error:

```
>>> from shipping import *
>>> h3 = HeatedRefrigeratedShippingContainer.create_empty('ESC', length_ft=40, celsiu\
s=5.0)
AttributeError: 'super' object has no attribute 'celsius'
```

With a combination of `super()` and properties there is much hidden machinery at play which we won't get into in this intermediate level book.[38] For now, this is solvable by retrieving the base class property setter *function*, `fset()`, from the base class property and calling it directly, remembering to explicitly pass `self`:

```
  @RefrigeratedShippingContainer.celsius.setter
  def celsius(self, value):
      if value < HeatedRefrigeratedShippingContainer.MIN_CELSIUS:
          raise ValueError("Temperature too cold!")
      RefrigeratedShippingContainer.celsius.fset(self, value)
```

A bonus of implementing access this way is that we get slightly more informative error messages which now tell us whether the requested temperature is "too hot" or "too cold", rather than just "out of range":

[38] We'll discuss `super()` in some detail in chapter 8. You can find a thorough discussion of more of the details in our book The Python Master.

```
>>> from shipping import *
>>> h4 = HeatedRefrigeratedShippingContainer.create_empty('ESC', length_ft=40, celsiu\
s=-18.0)
>>> h4.celsius = 10.0
Traceback (most recent call last):
  File "<console>", line 1, in <module>
    h.celsius = 10.0
  File "shipping.py", line 97, in celsius
    RefrigeratedShippingContainer.celsius.fset(self, value)
  File "shipping.py", line 73, in celsius
    raise ValueError("Temperature too hot!")
ValueError: Temperature too hot!
>>> h4.celsius = -26.0
Traceback (most recent call last):
  File "<console>", line 1, in <module>
    h.celsius = -26.0
  File "shipping.py", line 96, in celsius
    raise ValueError("Temperature too cold!")
ValueError: Temperature too cold!
```

Consistency through the use of properties

Notice that we've been careful to route all access to the `_celsius` attribute through the `celsius` property. As such, none of the other code which needs to respect the constraints needs to be modified. For example, the `fahrenheit` setter, although not itself overridden, now respects the lower temperature limit. For reference, -14 degrees Fahrenheit is a little below the limit of -25 degrees Celsius:

```
>>> h4.fahrenheit = -14.0
Traceback (most recent call last):
  File "<console>", line 1, in <module>
    h.fahrenheit = -14.0
  File "shipping.py", line 82, in fahrenheit
    self.celsius = RefrigeratedShippingContainer._f_to_c(value)
  File "shipping.py", line 96, in celsius
    raise ValueError("Temperature too cold!")
ValueError: Temperature too cold!
```

This works, but to be honest we think this implementation of the overridden `celsius` property is a bit of a mess, containing as it does two direct references to the base class. It's perhaps not so bad in this case, but the class defining the original property could have been many levels up in the inheritance hierarchy. Knowing this technique is useful, though, for times when you're not in a position to modify the base class implementation. Nonetheless, we'd like to find a more elegant — albeit more intrusive — solution, and that's what we'll pursue in next section.

The template method pattern

We've seen that it's straightforward to override property getters, but it's somewhat more involved — and quite syntactically messy — to override propetry setters. In this section we'll deploy a standard design pattern — the *Template Method* — to resolve these shortcomings and confer some addition benefits on our code. [39]

The template method is a very straightforward design pattern where we implement skeletal operations in base classes, deferring some details to subclasses. We do this by calling methods in the base class which are either not defined at all or which which have a trivial implementation (for example, raising `NotImplementedErrors`). Such methods must be overridden in subclasses in order to be useful.[40] An alternative is that we do supply useful details in the base class but allow them to be specialised in derived classes.

Template property getter

Let's start by using the template method to implement a *getter*. We'll use the `volume_ft3` property to demonstrate, because that is a property we for which we override the getter in a subclass. To do this, we extract the computation from the getter in the base class into a new function `_calc_volume()`:

[39] https://en.wikipedia.org/wiki/Template_method_pattern

[40] As the old saying goes, "there's no problem in computer science which can't be solved by an additional level of indirection".

```
class ShippingContainer:

    # ...

    def _calc_volume(self):
        return ShippingContainer.HEIGHT_FT * ShippingContainer.WIDTH_FT * self.length\
_ft

    @property
    def volume_ft3(self):
        return self._calc_volume()
```

The `volume_ft3` property is now a template method. It doesn't do anything itself except delegate to a regular method, which can be supplied or overridden in a derived class.

We'll now override this regular method in the derived class by converting the existing overridden property into a regular undecorated method:

```
  def _calc_volume(self):
      return super()._calc_volume() - RefrigeratedShippingContainer.FRIDGE_VOLUME_FT3
```

This new version leans on the base class implementation by using a call to `super()`.

With this change in place, we get the behaviour we're looking for without having overridden the property itself:

```
>>> from shipping import *
>>> c = ShippingContainer.create_empty(length_ft=20)
>>> c.volume_ft3
1360.0
>>> r = RefrigeratedShippingContainer.create_empty(length_ft=20, celsius=-18.0)
>>> r.volume_ft3
1260.0
```

Template property setter

We can use the same technique to override a property setter without having to remember any funky syntax. In this case we'll turn the `celsius` setter into a template method, which delegates to an undecorated `_set_celsius()` method:

```
  @celsius.setter
  def celsius(self, value):
      self._set_celsius(value)

  def _set_celsius(self, value):
      if value > RefrigeratedShippingContainer.MAX_CELSIUS:
          raise ValueError("Temperature too hot!")
      self._celsius = value
```

We can now remove the horrible property override construct and override the `_set_celsius()` method instead:

```
  def _set_celsius(self, value):
      if value < HeatedRefrigeratedShippingContainer.MIN_CELSIUS:
          raise ValueError("Temperature too cold!")
      super()._set_celsius(value)
```

In this case, we've decided to use `super()` to call the base class implementation. Here it is in action:

```
>>> from shipping import *
>>> h = HeatedRefrigeratedShippingContainer.create_empty('YML', length_ft=40, celsius\
=-18.0)
>>> h.celsius
-18.0
>>> h.celsius = -30
Traceback (most recent call last):
  File "<console>", line 1, in <module>
    h.celsius = -30
  File "shipping.py", line 72, in celsius
    self._set_celsius(value)
  File "shipping.py", line 98, in _set_celsius
    raise ValueError("Temperature too cold!")
ValueError: Temperature too cold!
>>> h.celsius = 5
Traceback (most recent call last):
  File "<console>", line 1, in <module>
    h.celsius = 5
  File "shipping.py", line 72, in celsius
    self._set_celsius(value)
  File "shipping.py", line 99, in _set_celsius
```

```
    super()._set_celsius(value)
  File "shipping.py", line 76, in _set_celsius
    raise ValueError("Temperature too hot!")
ValueError: Temperature too hot!
```

Summary

In this chapter we've looked at a number of topics:

- Class and instance attributes
  - Covered the distinction between class attributes and instance attributes
  - Demonstrated how class attributes are shared between all instances of a class
  - Shown how to refer to class attributes from within or without the class definition by fully qualifying them with the class name
  - Warned against attempting to assign to class attributes through the `self` instance, which actually creates a new instance attribute
- static- and class-methods
  - Used the `@staticmethod` decorator to define methods within the class which do not depend on either class or instance objects
  - Used the `@classmethod` decorator to define methods which operate on the class object
  - Implemented an idiom called *named constructors* using class methods
  - Shown how static and class method behave with respect to inheritance
  - Shown that static and class methods can support polymorphic method dispatch when invoked through an instance rather through a class
- Properties
  - Introduced properties to wrap attributes with getters and optional setter methods using the `@property` decorator
  - Finally, we showed an easy way to override properties by applying the *template method* design pattern

We've covered a lot of ground in this chapter, including some complex interactions of Python features. It's important you understand them before moving on. In the next part of the book, we'll change tack and look at how to make your classes more user- and developer-friendly, by controlling their string representations.

Chapter 5 - Strings and Representations

In this chapter we'll look at string representations of objects in Python, and in particular we'll cover the important but oft-confused differences between `repr()` and `str()`. Understanding the various string representations in Python is important for writing maintainable, debuggable, and human-friendly programs. We'll show you what you need to know to use them properly.

Two string representations

As you already know, Python supports two primary ways of making string representations of objects, the built-in functions `repr()` and `str()`. Each of these can take any object as an argument and produce a string representation of some form.

These two functions rely on the special methods `__repr__()` and `__str__()` of the object passed to them to generate the strings they produce, and it's possible for class designers to control these string representations by defining those functions.

Here's a quick example:

```python
class Point2D:

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return '({}, {})'.format(self.x, self.y)

    def __repr__(self):
        return 'Point2D(x={}, y={})'.format(self.x, self.y)
```

The class `Point2D` defines both `__str__()` which returns a simple format, and `__repr__()` which returns a more complete and unambiguous format.

If we print both string representations we can see how the free functions `str()` and `repr()` use these methods:

```
>>> p = Point2D(42, 69)
>>> str(p)
'(42, 69)'
>>> repr(p)
'Point2D(x=42, y=69)'
```

So the big question is: why are there two representations, and what are they used for?

Strings for developers with `repr()`

First let's look at `repr()`.[41] The `repr()` function is intended to make an *unambigious* representation of an object, within reason. By unambiguous we mean that it should include the type of the object along with any identifying fields.

Recall that the `Point2D.__repr__()` looks like this:

```
'Point2D(x=42, y=69)'
```

This representation clearly indicates the type, and it shows the two attributes of the object which identify it. Anyone who sees the `repr()` of a `Point2D` will know for sure what kind of object it is and what values it holds.

> Some people will go so far as to suggest that the `repr()` of an object should be legitimate source code. That is, that you should be able to take the `repr`, enter it into a REPL or source file, and have it reconstruct the object. This isn't realistic for many classes, but it's not a bad guideline, so keep it mind when designing `repr`s.

`repr()` is important for situations where exactness is more important than brevity or readability. For example, `repr()` is well suited for debugging because it tells you all of the important details of an object. If an object's `repr()` tells you its type and important details,

[41] "repr" is an abbreviation of "representation".

you can spend less time inspecting objects and more time debugging logic. Likewise, `repr()` is generally the best option for logging purposes for many of the same reasons.

Generally speaking, `repr()` should contain more information than the `str()` representation of an object, so `repr()` is best suited for situations where explicit information is needed.

The `repr()` of an object should tell a developer everything they need to know about an object to fully identify it and — as much as is practical — see where it came from and how it fits into the larger program context. More concretely, the `repr()` of an object is what a developer will see in a debugger, so when deciding what to put into a `repr()`, think about what you'd want to know when debugging code that uses the class you're developing. This is a helpful guideline and will result in classes that are, as you might imagine, easier to debug and work with in general.

Always implement a `repr` for your classes

It's a good idea to always implement `__repr__()` for any class you write. It doesn't take much work to write a good `repr()`, and the work pays off when you find yourself debugging or scanning logs.

All objects come with a default implementation of `__repr__()`, but you'll almost always want to override this. The default representation tells you the class name and the ID of the object, but it tells you nothing about the important attributes.

For example, here's what we get if we use the default `__repr__()` implementation for our `Point2D` class:

```
'<__main__.Point2D object at 0x101a9e650>'
```

We can see that it's a `Point2D`, but it doesn't tell us much else of any consequence.

Strings for clients with `str()`

Where `repr()` is used to provide unambiguous, debugger friendly output, `str()` is intended to provide readable, human-friendly output. Another way of thinking about this is to say that `repr()` is intended for developers where `str()` is intended for clients; this isn't a hard-and-fast rule, but it's a useful starting point.

The `str()` representation is used in situations where, for example, it might be integrated into normal text, or where the programming-level details such as “class” might be meaningless. Recall also that the `str()` function is actually the *constructor* for the `str` type!

For example, consider our `Point2D` class again. Its `str()` representation looks like this:

```
>>> p = Point2D(123, 456)
>>> str(p)
'(123, 456)'
```

That representation doesn’t tell you anything about the type of the object bring printed, but in the right context it tells a human reader everything they need to know. For example this is perfectly meaningful to a person:

```
>>> print('The circle is centered at {}.'.format(p))
The circle is centered at (123, 456).
```

Compare that to using `repr()` for the same purpose:

```
>>> print('The circle is centered at {}.'.format(repr(p)))
The circle is centered at Point2D(x=123, y=456).
```

This is factually correct, but it’s likely more information than a user needs in that context. It’s also likely to confuse a lot of people who don’t care about programming details.

When are the representation used?

`str()` and `repr`() give us two possible string representations of objects. This raises some interesting questions: When are they used by other parts of Python? And do they ever rely on each other?

`print()` uses `str()`

An obvious place to look is the `print()` function. `print()`, since it’s generally designed to provide console output to users, uses the human-friendly `str()` representation:

```
>>> print(Point2D(123, 456))
(123, 456)
```

`str()` defaults to `repr()`

Interestingly, the default implementation of `str()` simply calls `repr()`. That is, if you don't define `__str__()` for a class then that class will use `__repr__()` when `str()` is requested.

You can see this if you remove `__str__()` from `Point2D`:

```
class Point2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return 'Point2D(x={}, y={})'.format(self.x, self.y)
```

If we test this at the REPL, we see that `str()` just gives us the results from `__repr__()`:

```
>>> p = Point2D(234, 567)
>>> str(p)
'Point2D(x=234, y=567)'
```

However, the reverse does not hold true: The `__str__()` method is not invoked when calling `repr()` if you have not implemented `__repr__()`.

We can see this if we remove `__repr__()` from `Point2D`:

```
class Point2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return '({}, {})'.format(self.x, self.y)
```

Now the REPL shows us that our class gets the default implementation of `__repr__()` inherited from the `object` base class:

```
>>> p = Point2D(234, 567)
>>> str(p)
'(234, 567)'
>>> repr(p)
'<__main__.Point2D object at 0x101a9e9d0>'
```

> Be sure to add `__repr__` back to your `Point2D` implementation now if you've removed it!

Printing collections of objects

Another place where Python has to decide which representation to use is when it prints collections of objects. It turns out that Python uses the `repr()` of an object when it's printed as part of a `list`, `dict`, or any other built-in type:

```
>>> l = [Point2D(i, i * 2) for i in range(3)]
>>> str(l)
'[Point2D(x=0, y=0), Point2D(x=1, y=2), Point2D(x=2, y=4)]'
>>> repr(l)
'[Point2D(x=0, y=0), Point2D(x=1, y=2), Point2D(x=2, y=4)]'
>>> d = {i: Point2D(i, i * 2) for i in range(3)}
>>> str(d)
'{0: Point2D(x=0, y=0), 1: Point2D(x=1, y=2), 2: Point2D(x=2, y=4)}'
>>> repr(d)
'{0: Point2D(x=0, y=0), 1: Point2D(x=1, y=2), 2: Point2D(x=2, y=4)}'
```

As you can see, `repr()` is used for contained objects whether `repr()` or `str()` is used for the collection itself.

Precise control with `format()`

The `format()` method on strings is another place where string representations are called behind the scenes:

```
>>> 'This is a point: {}'.format(Point2D(1, 2))
'This is a point: (1, 2)'
```

When you run code like this it appears that `str()` is being used.

Actually, something a bit more complex is going on. When the `format()` method replaces curly braces with an object's representation, it actually calls the special `__format__()` method on that object.

We can see that by adding `__format__()` to `Point2D`:

```
class Point2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return '({}, {})'.format(self.x, self.y)

    def __repr__(self):
        return 'Point2D(x={}, y={})'.format(self.x, self.y)

    def __format__(self, f):
        return '[Formatted point: {}, {}, {}]'.format(self.x, self.y, f)
```

Now when we print a point via `format` we get yet another representation:

```
>>> 'This is a point: {}'.format(Point2D(1, 2))
'This is a point: [Formatted point: 1, 2, ]'
```

Formatting instructions for `__format__()`

Unlike `__str__()` and `__repr__()`, `__format__()` accepts an argument, `f`. This argument contains any special formatting options specified in the caller's format string. If the caller puts a colon inside the curly braces of a formatting string, anything after the colon is sent verbatim as the argument to `__format__()`.

For example, in `Point2D` we could implement `__format__()` to reverse `x` and `y` if `r` is passed in as the format string:

```
  def __format__(self, f):
      if f == 'r':
          return '{}, {}'.format(self.y, self.x)
      else:
          return '{}, {}'.format(self.x, self.y)
```

If we now use `{:r}` instead of the standard `{}` placeholder, `x` and `y` are swapped in the output:

```
>>> '{}'.format(Point2D(1,2))
'1, 2'
>>> '{:r}'.format(Point2D(1,2))
'2, 1'
```

In general, however, you don't need to implement `__format__()`. Most classes can rely on the default behavior, which is to call `__str__()`. This explains why string's `format()` function seems, at first, to just call `__str__()`.

Forcing `format()` to use `__repr__()` or `__str__()`

In some cases you might need to force `format()` to use `__repr__()` directly rather than having it call `__format__()`. You can do this by putting `!r` in the formatting placeholder:

```
>>> '{!r}'.format(Point2D(1,2))
'Point2D(x=1, y=2)'
```

Likewise, you can bypass `__format__()` and use `__str__()` directly by putting `!s` in the formatting placeholder:

```
>>> '{!s}'.format(Point2D(1,2))
'(1, 2)'
```

By and large though, you won't have to think about these details surrounding `__format__()`. Almost all of the time you can implement `__repr__()` and possibly `__str__()`, and you will have well-behaved, fully-functioning Python objects.

The format() build-in function

Given that the `repr()` built-in function calls the `__repr__()` method of its argument, and that the `str()` built-in function call `__str__()` method of its argument, it's probably not too surprising that there is a `format()` built-in function which calls the `__format__()` method of its argument.

This built-in format function isn't seen in use very often, but it can lead to less cluttered code. Consider the following ways to print pi to three decimal places, the first using the string method, and the second using the built-in function:

```
>>> import math
>>> "{:.3f}".format(math.pi)
'3.142'
>>> format(math.pi, ".3f")
'3.142'
```

We don't provide the curly brace placeholders to the `format()` function, so it can only format a single object. The curly brace and colon syntax for formatting of multiple objects is peculiar to `str.format()`.

Leveraging `reprlib` for large strings

Since we're on the topic of `repr()`, this is a good time to introduce the `reprlib` module. `reprlib` provides an alternative implementation of the built-in `repr` function.[42]

The primary feature of `reprlib`'s implementation is that it places limits on how large a string can be. For example, if it's used to print a very large list, it will only print a limited number of the elements. This is useful when you're dealing with large data structures whose normal representation might go on for many thousands of lines.

Basic usage of `reprlib`

The basic usage of `reprlib` involves using `reprlib.repr()`. This function is a drop-in replacement for the built-in `repr`.

For example, we can use it to print a huge list of our `Point2D` objects:

[42] `reprlib` is part of Python's standard library. You can read all about it in the Python docs

```
>>> import reprlib
>>> points = [Point2D(x, y) for x in range(1000) for y in range(1000)]
>>> len(points)
1000000
>>> reprlib.repr(points)
'[Point2D(x=0, y=0), Point2D(x=0, y=1), Point2D(x=0, y=2), Point2D(x=0, y=3), Point2D\
(x=0, y=4), Point2D(x=0, y=5), ...]'
```

Here we made a list of one million points. If we had used the built-in `repr()` to print it we would have had to print all one million entries.

Instead, `reprlib.repr()` just printed the first few elements followed by an ellipsis to indicate that there are more elements. For many purposes this is a *much* more useful representation. In a debugger, for example, seeing a string containing all one million entries would be worse than useless; it would often be extremely detrimental. So `reprlib` is useful for situations like that.

`reprlib.Repr`

While `reprlib.repr()` is the main entry point into `reprlib` for most people, there's significantly more to the module than that. `reprlib`'s functionality is built around a class, `reprlib.Repr`. This class implements all of the support for customizing representations. `Repr` is designed to be customized and subclassed, so you can create your own specialized `Repr` generators if you want to; the details of how to do that are beyond the scope of this book, but you can find all of the details in the Python standard library documentation.[43]

The `reprlib` module instantiates a singleton instance of this `Repr` class for you. It is named `reprlib.aRepr`, and `reprlib.repr()` actually just calls the `reprlib.aRepr.repr()` function. So you can manipulate this pre-made instance if you want to control default `reprlib` behavior throughout your program.

We think `reprlib` is a good module to know about, and while you may never actually need to work with it in detail, using just its basic functionality can be very useful.

The `ascii()`, `ord()` and `chr()` built-in functions

We'll finish up this chapter by looking a handful of functions that can be useful when dealing with string representations. These aren't required for implementing `__repr__()` or `__str_-`

[43] The section “Subclassing Repr Objects” gives an example.

`_()` by any stretch, but since we're talking about strings so much in this chapter it's a good place to mention them.

`ascii()`

The first function we'll look at is `ascii()`. This function takes a string as an argument and converts all of the non-ASCII characters into escape sequences. We've actually seen this function in chapter 3, though we didn't explain it then.

Here's how it looks in action:

```
>>> x = 'Hællø'
>>> type(x)
<class 'str'>
>>> y = ascii(x)
>>> y
"'H\\xe6ll\\xf8'"
>>> type(y)
<class 'str'>
```

`ascii()` takes in a Unicode string, replaces all of the non-ASCII characters with escape sequences, and they returns another Unicode string. This can be useful in situations where you need to serialize data as ASCII, or if you can't communicate encoding information but don't want to lose Unicode data.

Converting between integer codepoints and strings

Two other Unicode related functions are `ord()` and `chr()`. These are complementary functions they are inverses of each other. The `ord()` function takes a single-character string as input and returns the integer Unicode codepoint for that character.

For example, here we convert the glyph for three-quarters into the decimal code point 190:

```
>>> x = '¾'
>>> ord(x)
190
```

Likewise, `chr()` takes a Unicode codepoint and returns a single-character string containing the character. Here we convert 190 back into the glyph for three-quarters:

```
>>> chr(190)
'¾'
```

As mentioned earlier these clearly reverse one another, so `ord(chr(x))` always equals `x`, and `chr(ord(y))` always equals `y`:

```
>>> x
'¾'
>>> chr(ord(x))
'¾'
>>> ord(chr(190))
190
```

Case study: String representations of tabular data

As we discussed earlier in this chapter, the `repr` of an object is intended to be used by developers for logging, debugging, and other activities where an unambiguous format is more important that a human-friendly one. Very often this means that the `repr` of an object is larger than the `str`, if only because the `repr` contain extra identifying information.

However, there are times when it makes sense for a `repr` to be smaller than a `str`.

For example, consider a simple class for rendering tabular data. It comprises a list of header strings and a collection of lists of data for the table's columns:[44]

```
class Table:
    def __init__(self, header, *data):
        self.header = list(header)
        self.data = data
        assert len(header) == len(data)
```

A natural `str` representation for this class is a textual, multi-line table showing the headers and all of the data. That would look something like this:

[44] The assertion in this code would probably be an exception in production code, but the assertion is more expressive for the purposes of this example.

```
    def _column_width(self, i):
        rslt = max(len(str(x)) for x in self.data[i])
        return max(len(self.header[i]), rslt)

    def __str__(self):
        col_count = len(self.header)
        col_widths = [self._column_width(i) for i in range(col_count)]
        format_specs = ['{{:{}}}'.format(col_widths[i])
                        for i in range(col_count)]

        rslt = []

        rslt.append(
            format_specs[i].format(self.header[i])
            for i in range(col_count))

        rslt.append(
            ('=' * col_widths[i]
              for i in range(col_count)))

        for row in zip(*self.data):
            rslt.append(
                [format_specs[i].format(row[i])
                  for i in range(col_count)])

        print(rslt)

        rslt = (' '.join(r) for r in rslt)

        return '\n'.join(rslt)
```

There's quite a bit going on in this method, but most of it involves calculating column widths and then making sure that everything it printed with the correct widths. We won't cover this method in detail here, but it's well worth making sure you understand how it works. We'll leave that as an exercise for the curious reader.

In the end, you can see its results with a simple example:

```
>>> t = Table(['First name', 'Last name'],
...   ['Fred', 'George', 'Scooby'],
...   ['Flintstone', 'Jetson', 'Doo'])
>>> print(str(t))
First name Last name
========== ==========
Fred       Flintstone
George     Jetson
Scooby     Doo
```

And we're sure you can imagine tables with *much* more data than that.

But is this format really what you'd want for, say, debugging purposes? In the case of a table class like this, a good `repr` should probably just include the column headers; the actual data is not nearly as important. As a result, you can implement `repr` something like this:

```
  def __repr__(self):
      return 'Table(header={})'.format(self.header)
```

This is not only shorter to implement, but the string it produces is shorter as well:

```
>>> print(repr(t))
Table(header=['First name', 'Last name'])
```

So while you might generally find that your `repr`s are longer than your `str`s, that won't always be the case. The important thing to remember is that each of these functions serves a distinct purpose, and addressing these purposes is your real goal.

Summary

String representations may seem like a small issue to be concerned about, but this chapter shows that there are good reasons to pay attention to them. Let's review what we covered:

- `str()` and `repr()`
  - Python has two primary string representations for objects, `str()` and `repr()`
  - the `str()` function is used to create `str` representations, and it relies on the `__str__` method

  - The `repr()` function is used to create `repr` representations, and it relies on the `__repr__()` method
  - `__repr__()` should produce an unambiguous, precise representation of the object
  - `__repr__()` should include the type of and any identifying information for the object
  - The `repr()` form is useful for contexts like debugging and logging where information is more important that human readability
  - You should always implement `__repr__()` on your classes
  - The default `__repr__()` implementation is not very useful
  - The `str()` form is intended for human consumption, and doesn't need to be as precise as `repr()`
  - The `print()` function uses the `str` representation
  - By default, `__str__()` uses `__repr__()`
  - The default `__repr__()` does **not** use `__str__()`
  - Built-in collections like `list` use `repr()` to print their elements, even if the collection itself is printed with `str()`
  - Good `__repr__()` implementations are easy to write and can improve debugging
  - When reporting errors, the `repr()` of an object is generally more helpful than the `str()`
- `format()`
  - `str.format()` uses an object's `__format__()` method when inserting it into string templates
  - The default implementation of `__format__()` is to call `__str__()`
  - The argument to the `__format__()` method contains any special formatting instructions from the format string
    - These instructions must come after a colon between the curly braces for the object.
  - In general you do no need to implement `__format__()`
- `reprlib`
  - `reprlib` provides a drop-in replacement for `repr()` which limit output size
  - `reprlib` is useful when printing large data structures
  - `reprlib` provides the class `Repr` which implements most of `reprlib`'s functionality
  - `reprlib` instantiates a singleton `Repr` called `aRepr`
  - `reprlib.repr()` is the drop-in replacement for `repr()`
    - This function is just an alias for `reprlib.aRepr.repr()`

    - The `Repr` class is designed to be extended and customized via inheritance
- More string functions:
    - The function `ascii()` replace non-ASCII characters in a Unicode string with escape sequences
    - `ascii()` takes in a Unicode string and returns a Unicode string
    - The `ord()` function takes a single-character Unicode string and returns the integer codepoint of that character
    - The `chr()` function takes an integer codepoint and returns a single-character string containing that character
    - `ord()` and `chr()` are inverses of one another

Chapter 6 - Numeric and Scalar Types

In this chapter we'll dig deeper into some of the fundamentals of numerical computing. We'll take a look at the numeric types included in the Python language and the Python standard library, including those for dates and times. Let's start though, by reviewing and looking in a little more detail at some of the scalar types we have already encountered.

Python's basic numeric types

Throughout this book and our preceding book *The Python Apprentice*[45], we've extensively used two built-in numeric types: `int` and `float`. We've seen that Python 3 `int` objects can represent integers — that is whole numbers — of arbitrary magnitude limited only by practical constraints of available memory and the time required to manipulate large numbers. This sets Python apart from many other programming languages where the standard integer types have fixed size, storing only 16, 32 or 64 bits of precision.

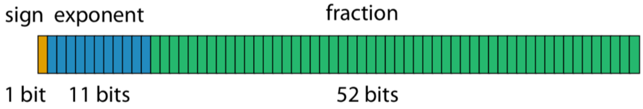
Python handles large integers with consummate ease:

```
>>> from math import factorial as fac
>>> fac(1000)
4023872600770937735437024339230039857193748642107146325437999104299385
1239862902059204420848696940480047998861019719605863166687299480855890
1323829669944590997424504087073759918823627727188732519779505950995276
1208749754624970436014182780946464962910563938874378864873371191810458
2578364784997701247663288983595573543251318532395846307555740911426241
7474349347553428646576611667797396668820291207379143853719588249808126
8678383745597317461360853795345242215865932019280908782973084313928444
0328123155861103697680135730421616874760967587134831202547858932076716
9132448426236131412508780208000261683151027341827977704784635868170164
3650241536913982812648102130927612448963599287051149649754199093422215
6683257208082133318611681155361583654698404670897560290095053761647584
7728421889679646244945160765353408198901385442487984959953319101723355
5566021394503997362807501378376153071277619268490343526252000158885351
4733161170210396817592151090778801939317811419454525722386554146106289
```

[45] https://leanpub.com/python-apprentice

```
218796022383897147608850627686296714667469756291123408243920816015378088989396451826324367161676217916890977991190375403127462228998800519544441428201218736174599264295658174662830295557029902432415318161721046583203678690611726015878352075151628422554026517048330422614397428693306169089796848259012545832716822645806652676995865268227280707578139185817888965220816434834482599326604336766017699961283186078838615027946595513115655203609398818061213855860030143569452722420634463179746059468257310379008402443243846565724501440282188525247093519062092902313649327349756551395872055965422874977401141334696271542284586237738753823048386568897646192738381490014076731044664025989949022222176590433990188601856652648506179970235619389701786004081188972991831102117122984590164192106888438712185564612496079872290851929681937238864261483965738229112312502418664935314397013742853192664987533721894069428143411852015801412334482801505139969429015348307764456909907315243327828826986460278986432113908350621709500259738986355427719674282224875758676575234422020757363056949882508796892816275384886339690995982628095612145099487170124451646126037902930912088908694202851064018215439945715680594187274899809425474217358240106367740459574178516082923013535808184009699637252423056085590370062427124341690900415369010593398383577939410970027753472000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Python’s `float` type — an abbreviation of “floating point number” — is specifically a 64-bit floating point number using a binary internal representation — officially known as *binary64* in the IEEE 754 standard. For those of you with background in C derived languages, this is commonly known as a ‘double’, although that terminology is unimportant in the context of much pure Python code [46] , and we do not use it here.



The bit-level binary structure of a Python float

Of the 64 bits within a Python `float`, one is allocated to representing the sign of the number, 11 are used to represent the exponent.[47] The remaining 52 are dedicated to representing the

[46] A double-precision floating point number has 64 bits of precision; twice that of a 32-bit single-precision float. The details are mostly important for interoperability with other programming languages which use C-compatiable types such the the single-precision `float` and the double-precision `double`. One Python context where these distinctions crop up is in libraries such as *Numpy* for handling large arrays of floating-point numbers. This is because *Numpy* is implemented in C.

[47] The value to which the fraction is raised.

fraction [48] — although owing to the way the encoding works in conjunction with the sign we get effectively 53 bits of precision. This means that thinking in decimal equivalents, Python floats have at least 15 digits of *decimal* precision and no more that 17 digits of decimal precision. In other words, you can convert any decimals with 15 significant figures into Python floats and back again without loss of information.

The limits of floats

Python floats support a very large range of values — larger than would be required in most applications. To determine the limits of the `float` type we can query the `sys.float_info` object from the built-in `sys` module:

```
>>> import sys
>>> sys.float_info
sys.float_info(max=1.7976931348623157e+308, max_exp=1024,
max_10_exp=308, min=2.2250738585072014e-308, min_exp=-1021,
min_10_exp=-307, dig=15, mant_dig=53, epsilon=2.220446049250313e-16,
radix=2, rounds=1)
```

You can see that the largest `float` is $1.7976931348623157 \times 10^{308}$ and the smallest `float` greater than zero is $2.2250738585072014 \times 10^{-308}$. If we want the most negative float or the greatest float smaller than zero we can negate these two values respectively:

```
>>> most_negative_float = -sys.float_info.max
>>> most_negative_float
-1.7976931348623157e+308
>>> greatest_negative_float = -sys.float_info.min
>>> greatest_negative_float
-2.2250738585072014e-308
```

So floats can represent a huge range of numbers, although you should be aware of their limitations. First of all, you shouldn't assume in general that any Python `int` can be converted without loss of information to a Python `float`. Conversion is obviously possible for small magnitude integers:

[48] Also known as the mantissa or significand.

```
>>> float(10)
10.0
```

However, because the mantissa has only 53 bits of binary precision we can't represent every integer above 2^{53}. Let's demonstrate that with a simple experiment at the REPL:

```
>>> 2**53
9007199254740992
>>> float(2**53)
9007199254740992.0
>>> float(2**53 + 1)
9007199254740992.0
>>> float(2**53 + 2)
9007199254740994.0
>>> float(2**53 + 3)
9007199254740996.0
>>> float(2**53 + 4)
9007199254740996.0
```

As you can see, only alternate integers can be represented in over the range of numbers we have tried. Furthermore, because the `float` type has finite precision, some fractional values can't be represented accurately, in much the same way that $1/3$ can't be represented as a finite-precision decimal. For example, neither 0.8 nor 0.7 can be represented in binary floating point, so computations involving them return incorrect answers rounded to a nearby value which *can* be represented:

```
>>> 0.8 --- 0.7
0.10000000000000009
```

If you're not familiar with floating point mathematics this can seem shocking at first, but is really no less reasonable than the fraction $2/3$ not displaying an infinitely recurring series of sixes:

```
>>> 2 / 3
0.6666666666666666
```

A full treatment of careful use of floating point arithmetic is well beyond the scope of this book, but we do want to alert you to some of the issues in order to motivate the introduction of

Python's alternative number types which avoid some of these problems by making different trade-offs. If you do need to learn understand more about floating point, we recommend David Goldberg's classic *What Every Computer Scientist Should Know About Floating-Point Arithmetic*[49].

The decimal module

As we have seen, the Python `float` type can result in problems with even the simplest of decimal values. This would be unacceptable in any application where exact arithmetic is needed, such as in a financial accounting setting. The `Decimal` type (uppercase 'D') in the `decimal` module (lowercase 'd') is a fast, correctly-rounded number type for performing arithmetic in base 10. Crucially, the `Decimal` type is still a floating point type (albeit with a base of ten rather than two) and has finite precision (although user configurable rather than fixed). Using `Decimal` in place of `float` for, say, an accounting application, can lead to significantly fewer hard-to-debug edge cases.

The decimal context

Let's take a look! We'll start by calling `decimal.getcontext()` to retrieve information about how the decimal system has been configured:

```
>>> import decimal
>>> decimal.getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
capitals=1, clamp=0, flags=[], traps=[InvalidOperation, DivisionByZero,
Overflow])
```

The most important figure here is `prec` which tells us that, by default, the decimal system is configured with 28 places of decimal precision. Some of the other values in here control rounding and error signalling modes, which can be important in certain applications.

The `Decimal` constructor

We create `Decimal` instances by calling the `Decimal` constructor. This is obvious enough when creating a decimal from an integer:

[49] http://dl.acm.org/citation.cfm?id=103163

```
>>> decimal.Decimal(5)
Decimal('5')
```

It’s a little awkward to use the module name every time, though, so let’s pull the `Decimal` type into the current scope:

```
>>> from decimal import Decimal
>>> Decimal(7)
Decimal('7')
```

Notice, than when the REPL echoes the the representation of the `Decimal` object back to us it places quotes around the seven. This indicates that the constructor also accepts strings:

```
>>> Decimal('0.8')
Decimal('0.8')
```

Let’s exercise that by replicating the computation that gave an inexact answer with `float` previously:

```
>>> Decimal('0.8') - Decimal('0.7')
Decimal('0.1')
```

Now this gives us an exact answer!

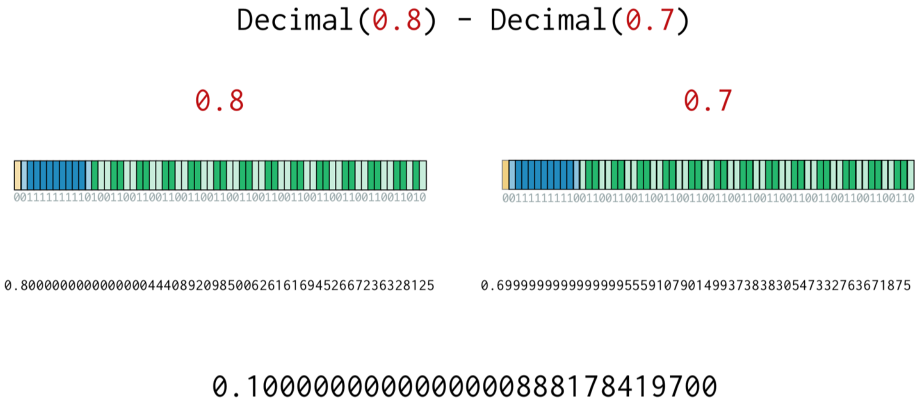
Constructing `Decimal`s from fractional values

For fractional values, passing the literal value to the constructor as a string can be very important. Consider this example *without* the quotes:

```
>>> Decimal(0.8) - Decimal(0.7)
Decimal('0.1000000000000000888178419700')
```

We’re back to the same problem we had with floats! To understand why, let’s deconstruct what’s going on here.

We have typed two numbers 0.8 and 0.7 in base 10 into the REPL. Each of these numbers represents a Python `float`, so Python converts our literal base-10 representations into internal base-2 representations within the `float` objects. Critically, neither of the values we have chosen can be represented exactly in base-2 so some rounding occurs.



Inadvertent construction of a `Decimal` from a `float` can lead to precision problems.

These rounded float values are then passed to the `Decimal` constructor and used to construct the internal base-10 representations which will be used for the computation. Finally, the subtraction is performed on the `Decimal` objects.

> Although the `Decimal` constructor supports conversion from `float`, you should always specify fractional `Decimal` literals as strings to avoid the creation of an inexact intermediate base-2 `float` object.

Signals

To avoid inadvertently constructing `Decimal` objects from `float`s we can modify the signal handling in the `decimal` module:

```
>>> decimal.getcontext().traps[decimal.FloatOperation] = True
>>> Decimal(0.8) --- Decimal(0.7)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
```

This also has the desirable effect of making comparisons between `Decimal` and `float` types raise an exception. Here we carefully construct a `Decimal` from a string on the left-hand side of the expression, but use a `float` on the right-hand-side:

```
>>> Decimal('0.8') > 0.7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
```

`Decimal` stores precision

`Decimal` (unlike `float`) preserves the precision of supplied number with trailing zeros:

```
>>> a = Decimal(3)
>>> b = Decimal('3.0')
>>> c = Decimal('3.00')
>>> a
Decimal('3')
>>> b
Decimal('3.0')
>>> c
Decimal('3.00')
```

This stored precision is propagated through computations:

```
>>> a * 2
Decimal('6')
>>> b * 2
Decimal('6.0')
>>> c * 2
Decimal('6.00')
```

The precision of constructed values is preserved whatever the precision setting in the module context, and this comes into play when we perform computations. First we'll reduce the precision down to just six significant figures:

```
>>> decimal.getcontext().prec = 6
```

and then create a value which exceeds that precision:

```
>>> d = Decimal('1.234567')
>>> d
Decimal('1.234567')
```

Now when performing a computation we see the limited context precision kick in:

```
>>> d + Decimal(1)
Decimal('2.23457')
```

Special values

We should also point out, that like the `float` type, `Decimal` supports the special values for infinity and not-a-number:

```
>>> Decimal('Infinity')
Decimal('Infinity')
>>> Decimal('-Infinity')
Decimal('-Infinity')
>>> Decimal('NaN')
Decimal('NaN')
```

These values propagate as you would expect through operations:

```
>>> Decimal('NaN') + Decimal('1.414')
Decimal('NaN')
```

Interaction with other numeric types

As we have seen, `Decimals` can be combined safely with Python integers, but the same cannot be said of `floats` or other number types we have met in this chapter. Operations with `floats` will raise a `TypeError`:

```
>>> Decimal('1.4') + 0.6
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'decimal.Decimal' and 'float'
```

This is by and large a good thing since it prevents inadvertent precision and representation problems creeping into programs.

Operations with `Decimal`

`Decimal` objects play very well with the rest of Python, and usually, once any input data has been converted to `Decimal` objects, program code can be very straightforward and proceed as for `floats` and `ints`. That said, there are a few differences to be aware of.

Modulus

One difference is that when using the modulus — or remainder — operator, the sign of the result is taken from the first operand (the dividend) rather than from the second operand (the divisor). Here's how things work with integer types:

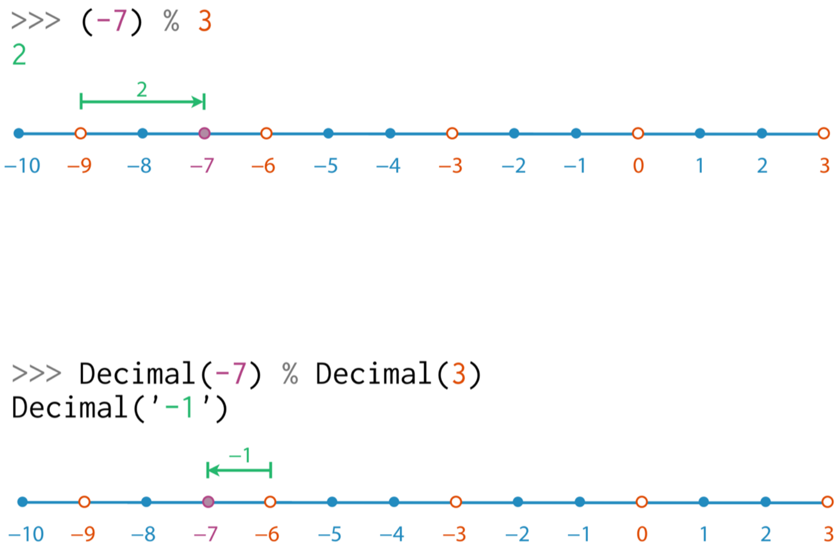
```
>>> (-7) % 3
2
```

This means that -7 is 2 greater than the largest multiple of 3 which is less than -7, which is -9. For Python integers the result of the modulus operation always has the same sign as the divisor; here 3 and 2 have the same sign.

However, the `Decimal` type modulus uses a different convention of returning a result which has the same sign as the dividend:

```
>>> from decimal import Decimal
>>> Decimal(-7) % Decimal(3)
Decimal('-1')
```

This means that -7 is one less than the next multiple of three towards zero from -7, which is -6.



```
>>> (-7) % 3
2
```

```
>>> Decimal(-7) % Decimal(3)
Decimal('-1')
```

A graphical view of modulus operations with Decimal on the number line

> It may seem capricious that Python has chosen different modulus conventions for different number types — and indeed it's somewhat arbitrary which convention different programming languages use — but it works this way so that `float` retains compatibility with legacy Python versions whereas `Decimal` is designed to implement the IEEE 854 Decimal Floating Point standard[a].
>
> [a] https://en.wikipedia.org/wiki/IEEE_854-1987

One result of this is that widespread implementations of common functions may not work as expected with different number types. Consider a function to test whether a number is odd, typically written like this:

```
>>> def is_odd(n):
...     return n % 2 == 1
...
```

This works well for `int`s:

```
>>> is_odd(2)
False
>>> is_odd(3)
True
>>> is_odd(-2)
False
>>> is_odd(-3)
True
```

It also works for `float`s:

```
>>> is_odd(2.0)
False
>>> is_odd(3.0)
True
>>> is_odd(-2.0)
False
>>> is_odd(-3.0)
True
```

But when used with `Decimal` it fails for negative odd numbers:

```
>>> is_odd(Decimal(2))
False
>>> is_odd(Decimal(3))
True
>>> is_odd(Decimal(-2))
False
>>> is_odd(Decimal(-3))
False
```

This is because $-1 \neq +1$:

```
>>> Decimal(-3) % 2
Decimal('-1')
```

To fix this we can rewrite `is_odd()` as a 'not even' test, which also work for negative decimals:

```
>>> def is_odd(n):
...     return n % 2 != 0
...
>>> is_odd(Decimal(-3))
True
```

Integer division

To maintain consistency and preserve the important identity:

```
x == (x // y) * y + x % y
```

the integer division operator also behaves differently. Consider this code:

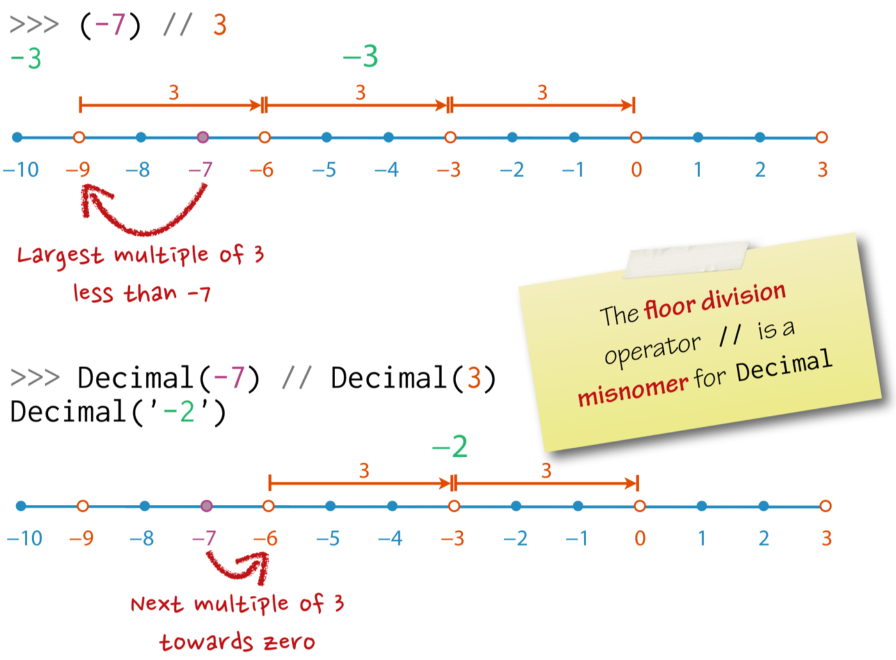
```
>>> -7 // 3
-3
```

It means that 3 divides into the largest multiple 3 of less than -7, which is -9, -3 times.

However, with decimals the result is different:

```
>>> Decimal(-7) // Decimal(3)
Decimal('-2')
```

This that 3 divides into the next multiple of 3 towards zero from -7, which is -6, -2 times.



A graphical view of integer division operations with Decimal on the number line

The `//` operator is known in Python as the "floor division operator". It's confusing, then, that it has not been implemented this way in the case of `Decimal` where it truncates towards zero. It's better to think of `//` as simply the integer division operator whose semantics are type dependent.

The `math` module

The functions of the `math` module cannot be used with the `Decimal` type, although some alternatives are provided as methods on the `Decimal` class. For example, to compute square roots, use `Decimal.sqrt()`:

```
>>> Decimal('0.81').sqrt()
Decimal('0.9')
```

A list of other methods supported by `Decimal` can be found in the Python documentation for `decimal`[50]

[50] http://docs.python.org/3/library/decimal.html

The need for more types

As we've shown, the `Decimal` type is crucial for accurately representing certain floating point values. For example, although `float` cannot exactly represent 0.7, this number *can* be exactly represented by `Decimal`. Nevertheless, many numbers, such as $2/3$, cannot be represented exactly in either binary or decimal floating point representations. To plug some of these gaps in the real number line, we must turn to a fourth number type for representing rational fractions.

The fractions module

The fractions module contains the `Fraction` type for representing so-called *rational* numbers which consist of the quotient of two integers. Examples of rational numbers are $2/3$ (with a numerator of two and a denominator of three) and $4/5$ (with a numerator of four and denominator of five). An important constraint on rational numbers is that the denominator must be non-zero.

The `Fraction` constructor

Let's see how to construct `Fraction`s. The first form of the constructor we'll look at accepts two integers for the numerator and denominator respectively:

```
>>> from fractions import Fraction
>>> two_thirds = Fraction(2, 3)
>>> two_thirds
Fraction(2, 3)
>>> four_fifths = Fraction(4, 5)
>>> four_fifths
Fraction(4, 5)
```

This is also the form in which the `Fraction` instance is echoed back to us by the REPL.

Attempting to construct with a zero denominator raises a `ZeroDivisionError`:

```
>>> Fraction(5, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    raise ZeroDivisionError('Fraction(%s, 0)' % numerator)
ZeroDivisionError: Fraction(5, 0)
```

Of course, given that the denominator can be 1, any `int`, however large, can be represented as a `Fraction`:

```
>>> Fraction(933262154439441526816992388562)
Fraction(933262154439441526816992388562, 1)
```

Fractions can also be constructed directly from `float` objects:

```
>>> Fraction(0.5)
Fraction(1, 2)
```

Be aware, though, that if the value you expect can't be exactly represented by the binary float, such as 0.1, you may not get the result you bargained for:

```
>>> Fraction(0.1)
Fraction(3602879701896397, 36028797018963968)
```

Fractions support interoperability with `Decimal` though, so if you can represent the value as a `Decimal`, you'll get an exact result:

```
>>> Fraction(Decimal('0.1'))
Fraction(1, 10)
```

Finally, as with decimals, `Fraction`s can be constructed from a string:

```
>>> Fraction('22/7')
Fraction(22, 7)
```

Arithmetic with `Fraction`

Arithmetic with fractions is without surprises:

```
>>> Fraction(2, 3) + Fraction(4, 5)
Fraction(22, 15)
>>> Fraction(2, 3) --- Fraction(4, 5)
Fraction(-2, 15)
>>> Fraction(2, 3) * Fraction(4, 5)
Fraction(8, 15)
>>> Fraction(2, 3) / Fraction(4, 5)
Fraction(5, 6)
>>> Fraction(2, 3) // Fraction(4, 5)
0
>>> Fraction(2, 3) % Fraction(4, 5)
Fraction(2, 3)
```

Unlike `Decimal`, the `Fraction` type does not support methods for square roots and such like. The reason for this is that the square root of a rational number, such as 2, may be an irrational number and not representable as a `Fraction` object. However, `Fraction` objects can be used with the `math.ceil()` and `math.floor()` functions which return integers:

```
>>> from math import floor
>>> floor(Fraction('4/3'))
1
```

Broad support for real numbers

Between them, Python `int`s, `float`s, `Decimal`s and `Fraction`s allow us to represent a wide variety of numbers on the real number line with various trade-offs in precision, exactness, convenience, and performance.

Later in this chapter we'll provide a compelling demonstration of the power of rational numbers for robust computation.

Complex Numbers

Python sports one more numeric type for complex numbers. This book isn't the place to explain complex numbers in depth — if you need to use complex numbers you probably already have a good understanding of them — so we'll quickly cover the syntactic specifics for Python.

Complex numbers are built into the Python language and don't need to be imported from a module. Each complex number has a real part and an imaginary part, and Python provides a special literal syntax to produce the imaginary part: placing a `j` suffix onto a number, where `j` represents the imaginary square-root of -1. Here we specify the number which is twice the square-root of -1:

```
>>> 2j
2j
```

> Depending on which background you have, you may have been expecting an `i` here rather than a `j`. Python uses the convention adopted by the electrical engineering community for denoting complex numbers — where complex numbers have important uses, as we'll see shortly — rather than the convention used by the mathematics community.

An imaginary number can be combined with a regular `float` representing a real number using the regular arithmetic operators:

```
>>> 3 + 4j
(3+4j)
```

Notice that this operation results in a complex number with non-zero real and imaginary parts, so Python displays both components of the number in parentheses to indicate that this is a single object. Such values have a type of `complex`:

```
>>> type(3 + 4j)
<class 'complex'>
```

The `complex` constructor

The `complex` constructor can also be used to produce complex number objects. It can be passed one or two numeric values representing the real and optional imaginary components of the number:

```
>>> complex(3)
(3+0j)
>>> complex(-2, 3)
(-2+3j)
```

It can also be passed a single argument containing a string delimited by optional parentheses but which must not contain whitespace:

```
>>> complex('(-2+3j)')
(-2+3j)
>>> complex('-2+3j')
(-2+3j)
>>> complex('-2 + 3j')
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
ValueError: complex() arg is a malformed string
```

> Note that `complex()` will accept any numeric type, so it can be used for conversion from other numeric types in much the same way as the `int` and `float` constructors can. However, the real and imaginary components are represented internally as floats with all the same advantages and limitations.

Operations on `complex`

To extract the real and imaginary components as `float`s use the `real` and `imag` attributes:

```
>>> c = 3 + 5j
>>> c.real
3.0
>>> c.imag
5.0
```

Complex numbers also support a method to produce the complex conjugate:

```
>>> c.conjugate()
(3-5j)
```

The `cmath` module

The functions of the `math` module cannot be used with complex numbers, so a module `cmath` is provided containing versions of the functions which both accept and return complex numbers. For example, although the regular `math.sqrt()` function cannot be used to compute the square roots of negative numbers:

```
>>> import math
>>> math.sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
```

The same operation works fine with `cmath.sqrt()`, returning an imaginary result:

```
>>> import cmath
>>> cmath.sqrt(-1)
1j
```

Polar coordinates

In addition to complex equivalents of all the standard `math` functions, `cmath` contains functions for converting between the standard cartesian form and polar coordinates.

To obtain the *phase* of a complex number — also known in mathematical circles as it's *argument* use — `cmath.phase()`:

```
>>> cmath.phase(1+1j)
0.7853981633974483
```

To get its modulus — or magnitude — use the built-in `abs()` function:

```
>>> abs(1+1j)
1.4142135623730951
```

We'll return to the `abs()` function shortly, in another context.

These two values can be returned as a tuple pair using the `cmath.polar()` function:

```
>>> cmath.polar(1+1j)
(1.4142135623730951, 0.7853981633974483)
```

which can of course be used in conjunction with tuple unpacking:

```
>>> modulus, phase = cmath.polar(1+1j)
>>> modulus
1.4142135623730951
>>> phase
0.7853981633974483
```

The operation can be reversed using the `cmath.rect()` function:

```
>>> cmath.rect(modulus, phase)
(1.0000000000000002+1j)
```

although note that repeated conversions may be subject to floating-point rounding error, as we have experienced here.

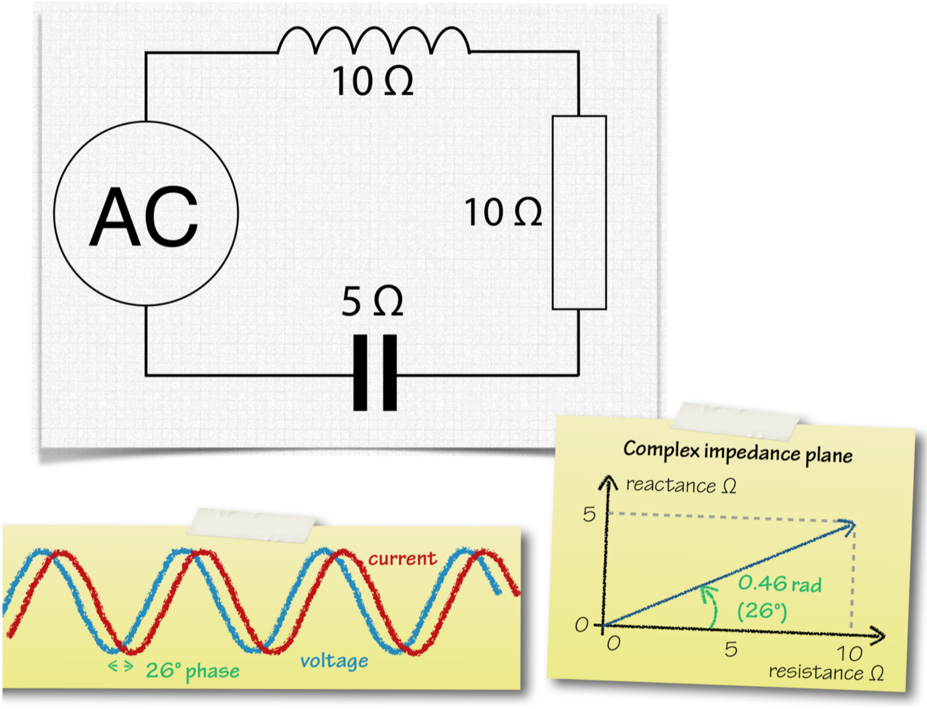
A practical example

To keep this firmly grounded in reality, here's an example of the practical application of complex numbers to electrical engineering: analysis of the phase relationship of voltage and current in AC circuits. First we create three functions to create complex values for the impedance of inductive, capacitive, and resistive electrical components respectively:

```
>>> def inductive(ohms):
...     return complex(0.0, ohms)
...
>>> def capacitive(ohms):
...     return complex(0.0, -ohms)
...
>>> def resistive(ohms):
...     return complex(ohms)
...
```

The impedance of a circuit is the sum of the the quantities for each component:

```
>>> def impedance(components):
...     z = sum(components)
...     return z
...
```



An alternating-current circuit we can analyze by representing impedance as a complex number

We can now model a simple series circuit with an inductor of 10 ohms reactance, a resistor of 10 ohms resistance and a capacitor with 5 ohms reactance:

```
>>> impedance([inductive(10), resistive(10), capacitive(5)])
(10+5j)
```

We now use `cmath.phase()` to extract the phase angle from the previous result:

```
>>> cmath.phase(_)
0.4636476090008061
```

and convert this from radians to degrees using a handy function in the `math` module:

```
>>> math.degrees(_)
26.56505117707799
```

This means that the voltage cycle lags the current cycle by a little over 26 degrees in this circuit.

Built-in functions relating to numbers

As we've seen, Python includes a large number of built-in functions and we'd like you to have seen them all — excluding a few we think you should avoid — by the end of this book. Several of the built-in functions are operations on numeric types, so it's appropriate to cover them here.

`abs()`

We already briefly encountered `abs()` when looking at complex numbers where it returned the magnitude of the number, which is always positive. When used with integers, floats, decimals or fractions, it simply returns the absolute value of the number, which is the non-negative magnitude without regards to its sign. In effect, for all number types including `complex`, `abs()` returns the distance from zero:

```
>>> abs(-5)
5
>>> abs(-5.0)
5.0
>>> abs(Decimal(-5))
Decimal('5')
>>> abs(Fraction(-5, 1))
Fraction(5, 1)
>>> abs(complex(0, -5))
5.0
```

`round()`

Another built-in is `round()` which rounds to a given number of *decimal* digits. For example:

```
>>> round(0.2812, 3)
0.281
>>> round(0.625, 1)
0.6
```

To avoid bias, when there are two equally close alternatives rounding is towards even numbers. So `round(1.5)` rounds up, and `round(2.5)` rounds down:

```
>>> round(1.5)
2
>>> round(2.5)
2
```

As with `abs()`, `round()` is implemented for `int` (where it has no effect), `float` (we have already seen), and `Decimal`:

```
>>> round(Decimal('3.25'), 1)
Decimal('3.2')
```

It's also implemented for `Fraction`:

```
>>> round(Fraction(57, 100), 2)
Fraction(57, 100)
>>> round(Fraction(57, 100), 1)
Fraction(3, 5)
>>> round(Fraction(57, 100), 0)
Fraction(1, 1)
```

`round()` is not supported for `complex`, however.

> Be aware that when used with `float`, which uses a binary representation, `round()` — which is fundamentally a decimal operation — can give surprising results. For example, rounding 2.675 to two places should yield 2.68 since 2.675 is midway between 2.67 and 2.68 and the algorithm round towards the even digit. However, in practice, we get an unexpectedly rounded down result:
>
> ```
> >>> round(2.675, 2)
> 2.67
> ```

> As we have seen before, this is caused by the fact that our literal float represented in base ten can't be exactly represented in base two, so what is getting rounded is the binary value which is close to, but not quite, the value we specified. If avoiding these quirks is important for your application, you know what to do: Use the `decimal` type!

All this talk of number bases brings us on to another set of built-in functions — the base conversions.

Base conversions

In chapter 1 of *The Python Apprentice*[51] we saw that Python supports integer literals in base 2 (binary) using a `0b` prefix:

```
>>> 0b101010
42
```

It also supports base 8 (octal) using a `0o` prefix:

```
>>> 0o52
42
```

And it supports base 16 (hexadecimal) using a `0x` prefix:

```
>>> 0x2a
42
```

Using the `bin()`, `oct()` and `hex()` functions, we can convert in the other direction, with each function returning a string containing a valid Python expression:

[51] https://github.com/python-apprentice

```
>>> bin(42)
'0b101010'
>>> oct(42)
'0o52'
>>> hex(42)
'0x2a'
```

If you don't want the prefix, you can strip it off using string slicing:

```
>>> hex(42)[2:]
'2a'
```

The `int()` constructor and conversion function also accepts an optional `base` argument. Here we use it to parse a string containing a hexadecimal number without the prefix into an integer object:

```
>>> int("2a", base=16)
42
```

The valid values of the `base` argument are zero, and then 2 to 36 inclusive. For numbers in base 2 to 36, as many digits as required from the sequence of 0-9 followed by a-z are used, although letters may be in lowercase or uppercase:

```
>>> int("acghd", base=18)
1125247
```

When specifying binary, octal, or hexadecimal, strings the standard Python prefix may be included:

```
>>> int("0b111000", base=2)
56
```

Finally, base zero tells Python to interpret the string according to whatever the prefix is or, if no prefix is present, assume it is decimal:

```
>>> int("0o664", base=0)
436
```

Note that base one — or unary — systems of counting are not supported.

Dates and times with the datetime module

The last important scalar types we consider in this chapter come from the `datetime` module. The types in this module should be the first resort when you need to represent time related quantities. The types are:

`date`
: a Gregorian calendar date.[52]

`time`
: the time within an ideal day, which ignores leap seconds.

`datetime`
: a composite of date and time.[53]

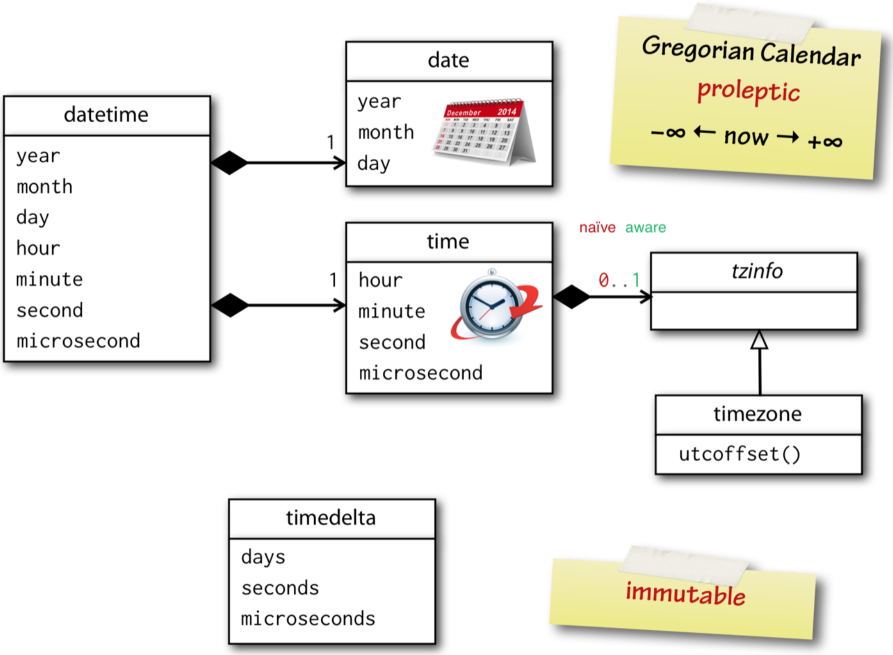
`tzinfo` (abstract) and `timezone` (concrete)
: classes are used for representing the time zone information required for 'aware' time objects.

`timedelta`
: a duration expressing the difference between two `date` or `datetime` instances.

As with the other number and scalar types we have looked at, all objects of these types are immutable — once created their values cannot be modified.

[52] Note that the type assumes a proleptic Gregorian calendar that extends backwards for all eternity and into the infinite future. For historical dates this must be used with some care. The last country to adopt the Gregorian calendar was Turkey, in 1927.

[53] Both `time` and `datetime` can be used in so-called 'naïve' or 'aware' modes. In naïve mode, the values lack time zone and daylight saving time information, and their meaning with respect to other time values is purely by convention within a particular program. In other words part of the meaning of the time is implicit. On the other hand, in 'aware' mode these objects have knowledge of both time zone and daylight saving time and so can be located with respect to other time objects.



The relationships between the major types in the `datetime` module

Dates

Let's start by importing the `datetime` module and representing some calendar dates:

```
>>> import datetime
>>> datetime.date(2014, 1, 6)
datetime.date(2014, 1, 6)
```

The year, month, and day are specified in order of descending size of unit duration, although if you can't remember the order, you can always be more explicit with keyword arguments:

```
>>> datetime.date(year=2014, month=1, day=6)
datetime.date(2014, 1, 6)
```

Each value is an integer, and the month and day values are *one-based* so, as in the example here, January is month one, and the sixth day of January is day six — just like regular dates.

Named constructors

For convenience, the `date` class provides a number of named constructors (or factory methods) implemented as class methods. The first of these is `today()` which returns the current date:

```
>>> datetime.date.today()
datetime.date(2014, 1, 6)
```

There's also a constructor which can create a date from a POSIX timestamp, which is the number of seconds since 1st January 1970. For example, the billionth second fell on the 9th of September 2001:

```
>>> datetime.date.fromtimestamp(1000000000)
datetime.date(2001, 9, 9)
```

The third named constructor is `fromordinal()` which accepts an integer number of days starting with one on at 1st January in year one, assuming the Gregorian calendar extends back that far:

```
>>> datetime.date.fromordinal(720669)
datetime.date(1974, 2, 15)
```

The year, month, and day values can be extracted with the attributes of the same name:

```
>>> d = datetime.date.today()
>>> d.year
2014
>>> d.month
1
>>> d.day
6
```

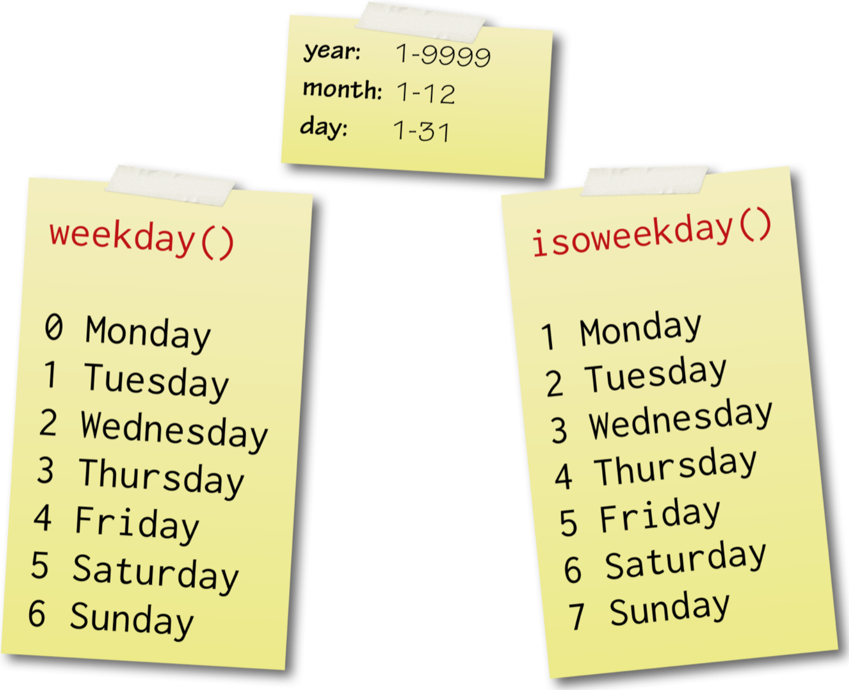
Instance methods

There are many useful instance methods on `date`. We cover some of the more frequently used ones here. To determine the weekday use either the `weekday()` or `isoweekday()` methods. The former returns a zero-based day number, in the range zero to six inclusive, where Monday is zero and Sunday is six:

```
>>> d.weekday()
0
```

The `isoweekday()` method uses a one-based system where Monday is one and Sunday is seven:

```
>>> d.isoweekday()
1
```



Different weekday numbering conventions in the `datetime` module

To return a string is ISO 8601 format — by far the most sensible way to represents dates as text — use the `isoformat()` method:

```
>>> d.isoformat()
'2014-01-06'
```

For more control over date formatting as strings you can you can use the `strftime()` method — read as “string-format-time” — using a wide variety of placeholders, as specified in the Python documentation[54]:

```
>>> d.strftime('%A %d %B %Y')
'Monday 06 January 2014'
```

Similarly, you can use the `format()` method of the string type with a suitable format placeholder format string:

```
>>> "The date is {:%A %d %B %Y}".format(d)
'Monday 06 January 2014'
```

> Unfortunately, both of these techniques delegate to the underlying platform dependent libraries underpinning your Python interpreter, so the format strings can be fragile with respect to portable code. Furthermore, many platforms do not provide tools to modify the result in subtle ways, such as omitting the leading zero on month days less than ten. On this computer we can insert a hyphen to suppress leading zeros:
>
> ```
> >>> d.strftime('%A %-d %B %Y')
> 'Monday 6 January 2014'
> ```
>
> But this is not portable, even between different versions of the same operating system. A better, and altogether more Pythonic solution, is to extract the date components individually and pick and choose between date-specific formatting operators and date attribute access for each component:
>
> ```
> >>> "{date:%A} {date.day} {date:%B} {date.year}".format(date=d)
> 'Monday 6 January 2014'
> ```
>
> This is both more powerful and portable.

Finally, the limits of date instances can be determined with the `min` and `max` class attributes:

[54] https://docs.python.org/3.6/library/datetime.html#strftime-strptime-behavior

```
>>> datetime.date.min
datetime.date(1, 1, 1)
>>> datetime.date.max
datetime.date(9999, 12, 31)
```

The interval between successive dates retrieved from the `resolution` class attribute:

```
>>> datetime.date.resolution
datetime.timedelta(1)
```

The response from `resolution()` is in terms of the `timedelta` type which we'll look at shortly.

Times

The `time` class is used to represent a time within an unspecified day with optional time zone information. Each time value is specified in terms of attributes for hours, minutes, seconds and microseconds. Each of these is optional, although of course the preceding values must be provided if positional arguments are used:

```
>>> datetime.time(3)
datetime.time(3, 0)
>>> datetime.time(3, 1)
datetime.time(3, 1)
>>> datetime.time(3, 1, 2)
datetime.time(3, 1, 2)
>>> datetime.time(3, 1, 2, 232)
datetime.time(3, 1, 2, 232)
```

As is so often the case, keyword arguments can lend a great deal of clarity to the code:

```
>>> datetime.time(hour=23, minute=59, second=59, microsecond=999999)
datetime.time(23, 59, 59, 999999)
```

All values are zero-based integers (recall that for `date` they were one-based), and the value we have just created represents that last representable instant of any day.

Curiously, there are no named constructors for `time` objects.

The components of the time can be retrieved through the expected attributes:

```
>>> t = datetime.time(10, 32, 47, 675623)
>>> t.hour
10
>>> t.minute
32
>>> t.second
47
>>> t.microsecond
675623
```

Formatting

As for dates, an ISO 8601 string representation can be obtained with the `isoformat()` method:

```
>>> t.isoformat()
'10:32:47.675623'
```

More sophisticated formatting is available through the `strftime()` method and the regular `str.format()` method, although the same caveats about delegating to the underlying C library apply, with the portability traps for the unwary:

```
>>> t.strftime('%Hh%Mm%Ss')
'10h32m47s'
```

We prefer the more Pythonic:

```
>>> "{t.hour}h{t.minute}m{t.second}s".format(t=t)
'10h32m47s'
```

Range details

The minimum/maximum times and the resolution can be obtained using the same class attributes as for dates:

```
>>> datetime.time.min
datetime.time(0, 0)
>>> datetime.time.max
datetime.time(23, 59, 59, 999999)
>>> datetime.time.resolution
datetime.timedelta(0, 0, 1)
```

Datetimes

You may have noticed that throughout this section we have fully qualified the types in the `datetime` module with the module name, and the reason will now become apparent. The composite type which combines date and time into a single object is also called `datetime` with a lowercase 'd'. For this reason, you should avoid doing:

```
>>> from datetime import datetime
```

If you do this, the `datetime` name will refer to the *class* rather than to the enclosing *module*. As such, trying to get hold of the `time` type then results in retrieval of the `time()` *method* of the `datetime` class:

```
>>> datetime.time
<method 'time' of 'datetime.datetime' objects>
```

To avoid this nonsense you could import the `datetime` class and bind it to an alternative name:

```
>>> from datetime import datetime as Datetime
```

Another common option is to use a short module name by doing:

```
>>> import datetime as dt
```

We'll continue what we've been doing and fully qualify the name.

Constructors

As you might expect, the compound `datetime` constructor accepts year, month, day, hour, minute, second, and microsecond values, of which at least year, month, and day must be supplied. The argument ranges are the same as for the separate `date` and `time` constructors:

```
>>> datetime.datetime(2003, 5, 12, 14, 33, 22, 245323)
datetime.datetime(2003, 5, 12, 14, 33, 22, 245323)
```

In addition, the `datetime` class sports a rich selection of named constructors implemented as class methods. The `today()` and `now()` methods are almost synonymous, although `now()` may be more precise on some systems. In addition, the `now()` method allows specification of a timezone, but we'll return to that topic later:

```
>>> datetime.datetime.today()
datetime.datetime(2014, 1, 6, 14, 4, 20, 450922)
>>> datetime.datetime.now()
datetime.datetime(2014, 1, 6, 14, 4, 26, 130817)
```

> Remember that these functions, and all the other constructors we've seen so far, return the *local* time according to your machine, without any record of where that might be. You can get a standardised time using the `utcnow()` function which returns the current Coordinated Universal Time (UTC) taking into account the timezone of your current locale:
>
> ```
> >>> datetime.datetime.utcnow()
> datetime.datetime(2014, 1, 6, 13, 4, 33, 548969)
> ```
>
> We're in Norway, which in the winter is one hour ahead of UTC, so `utcnow()` returns a time just after 1 PM rather than 2 PM. Note that even `utcnow()` returns a naïve datetime which doesn't *know* it is represented in UTC. We'll cover a time zone aware alternative shortly.

As with the `date` class, `datetime` supports the `fromordinal()`:

```
>>> datetime.datetime.fromordinal(5)
datetime.datetime(1, 1, 5, 0, 0)
```

It also supports the `fromtimestamp()` and `utcfromtimestamp()` methods:

```
>>> datetime.datetime.fromtimestamp(3635352)
datetime.datetime(1970, 2, 12, 2, 49, 12)
>>> datetime.datetime.utcfromtimestamp(3635352)
datetime.datetime(1970, 2, 12, 1, 49, 12)
```

If you wish to combine separate `date` and `time` objects into a single `datetime` instance you can use the `combine()` classmethod. For example, to represent 8:15 this morning you can do:

```
>>> d = datetime.date.today()
>>> t = datetime.time(8, 15)
>>> datetime.datetime.combine(d, t)
datetime.datetime(2014, 1, 6, 8, 15)
```

The final named constructor, `strptime()` — read as string-parse-time — can be used to parse a date in string format according to a supplied format string. This uses the same syntax as used for rendering dates and times to strings in the other direction with `strftime()`:

```
>>> dt = datetime.datetime.strptime("Monday 6 January 2014, 12:13:31", "%A %d %B %Y, \
%H:%M:%S")
>>> dt
datetime.datetime(2014, 1, 6, 12, 13, 31)
```

Useful methods

To obtain separate `date` and `time` objects from a `datetime` object use the `date()` and `time()` methods:

```
>>> dt.date()
datetime.date(2014, 1, 6)
>>> dt.time()
datetime.time(12, 13, 31)
```

Beyond that the `datetime` type essentially supports the combination of the attributes and methods supported by `date` and `time` individually such as the `day` attribute:

```
>>> dt.day
```

and `isoformat()` for ISO 8601 date-times:

```
>>> dt.isoformat()
'2014-01-06T12:13:31'
```

Durations

Durations are modelled in Python by the `timedelta` type, which is the difference between two `date`s or `datetime`s.

Constructors

The `timedelta` constructor is superficially similar to the constructor for the other types, but has some important differences. The constructor accepts any combination of days, seconds, microseconds, milliseconds, minutes, hours and weeks. Although positional arguments could be used we *strongly* urge you to use keyword arguments for the sake of anybody reading your code in future, including yourself!

The constructor normalises and sums the arguments, so specifying one millisecond and 1000 microseconds results in a total of 2000 microseconds:

```
>>> datetime.timedelta(milliseconds=1, microseconds=1000)
datetime.timedelta(0, 0, 2000)
```

> Notice that only three numbers are stored internally, which represent days, seconds and microseconds:
>
> ```
> >>> td = datetime.timedelta(weeks=1, minutes=2, milliseconds=5500)
> >>> td
> datetime.timedelta(7, 125, 500000)
> >>> td.days
> 7
> >>> td.seconds
> 125
> >>> td.microseconds
> 500000
> ```

String conversion

No special string formatting operations are provided for `timedelta`s although you can use the `str()` function to get a friendly representation:

```
>>> str(td)
'7 days, 0:02:05.500000'
```

Compare that to the `repr()` which we have already seen:

```
>>> repr(td)
'datetime.timedelta(7, 125, 500000)'
```

Time arithmetic

Time delta objects arise when performing arithmetic on `datetime` or `date` objects. For example, subtracting two `datetime`s results in a `timedelta`:

```
>>> a = datetime.datetime(year=2014, month=5, day=8, hour=14, minute=22)
>>> b = datetime.datetime(year=2014, month=3, day=14, hour=12, minute=9)
>>> a - b
datetime.timedelta(55, 7980)
>>> d = a - b
>>> d
datetime.timedelta(55, 7980)
>>> d.total_seconds()
4759980.0
```

Or to find the date in three weeks time by adding a `timedelta` to a `date`:

```
>>> datetime.date.today() + datetime.timedelta(weeks=1) * 3
datetime.date(2014, 1, 27)
```

Be aware that arithmetic on `time` objects is not supported:

```
>>> f = datetime.time(14, 30, 0)
>>> g = datetime.time(15, 45, 0)
>>> f --- g
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'datetime.time' and 'datetime.time'
```

Time zones

So far, all of the time related objects we have created have been so- called *naïve* times which represent times in local time. To create time zone 'aware' objects we must attach instances of a `tzinfo` object to our time values. Time zones, and daylight saving time, are a very complex domain mired in international politics and which could change at any time. As such, the Python standard library does not include exhaustive time zone data. If you need up-to-date time zone data you'll need to use the third-party `pytz` or `dateutil` modules. That said, Python 3 — although not Python 2 — contains rudimentary support for timezone specification. The `tzinfo` abstraction, on which more complete timezone support can be added, is supported in both Python 2 and Python 3.

`timezone`

The `tzinfo` class is abstract, and so cannot be instantiated directly. Fortunately, Python 3 includes a simple `timezone` concrete class which can be used to represent timezones which are a fixed offset from UTC.

For example, here in Norway we are currently in the Central European Time or CET time zone which is UTC+1. Let's construct a `timezone` object to represent this:

```
>>> cet = datetime.timezone(datetime.timedelta(hours=1), "CET")
>>> cet
datetime.timezone(datetime.timedelta(0, 3600), 'CET')
```

I can now specify this `tzinfo` instance when constructing a `time` or a `datetime` object. Here's the departure time of my flight to London tomorrow:

```
>>> departure = datetime.datetime(year=2014, month=1, day=7, hour=11, minute=30, tzin\
fo=cet)
```

The `timezone` class has an attribute called `utc` which is an instance of `timezone` configured with a zero offset from UTC, useful for representing UTC times. In the wintertime London is on UTC, so I'll specify my arrival in UTC:

```
>>> arrival = datetime.datetime(year=2014, month=1, day=7, hour=13, minute=5, tzinfo=\
datetime.timezone.utc)
```

The flight duration is 9300 seconds:

```
>>> arrival - departure
datetime.timedelta(0, 9300)
```

This is more usably formatted as 2 hours 35 minutes:

```
>>> str(arrival - departure)
'2:35:00'
```

For more complete time zone support including correct handling of daylight saving time which is *not* handled by the basic `timezone` class, you'll need to either subclass the `tzinfo` base class yourself, for which instructions are provided in the Python documentation, or employ one of the third-party packages such as `pytz`.

Case study: Rational numbers and computational geometry

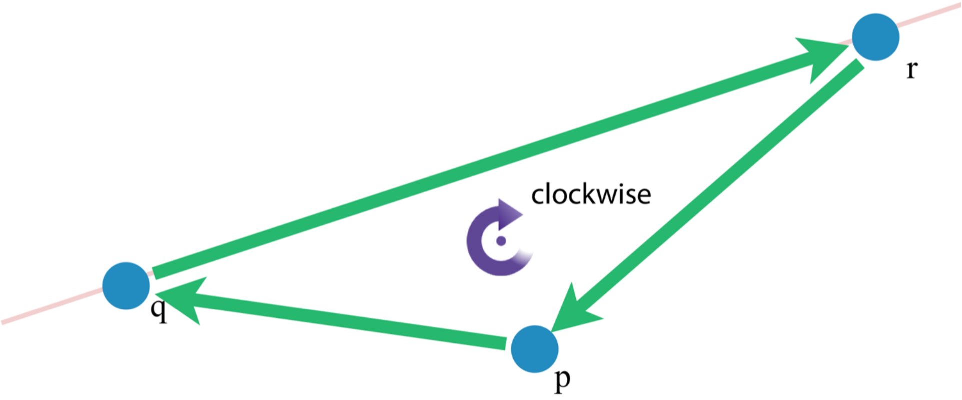
Rational numbers have an interesting role to play in the field of computational geometry — a world where lines have zero thickness, circles are perfectly round and points are dimensionless. Creating robust geometric algorithms using finite precision number types such as Python's `float` is fiendishly difficult because it's not possible to exactly represent numbers such as $1/3$. This rather gets in the way of performing simple operations like dividing a line into exactly three equal segments.

As such, rational numbers modelled by Python's `Fraction`s can be useful for implementing *robust* geometric algorithms. These algorithms are often deeply elegant and surprising because they must avoid any detour into the realm of irrational numbers which cannot be represented in finite precision, which means that using seemingly innocuous operations like square root — for example to determine the length of a line using Pythagoras — is not permitted.

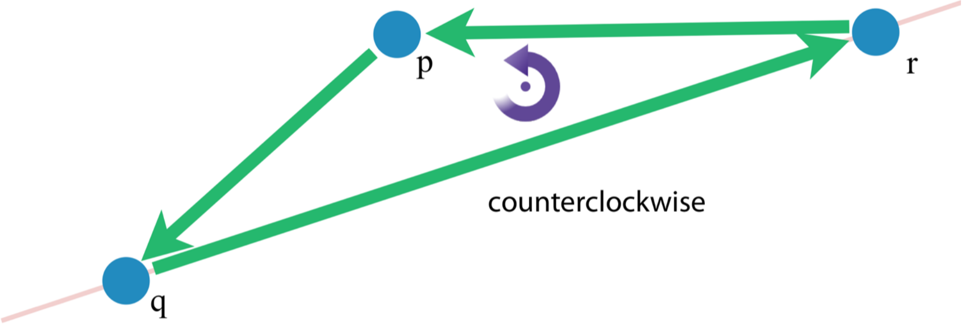
Testing for collinearity

One example of an algorithm which benefits from rational numbers is a simple collinearity test, that is, a test to determine whether three points lie on the same straight line. This can be further refined to consider whether a query point, called *p* is above, exactly on, or below the line.

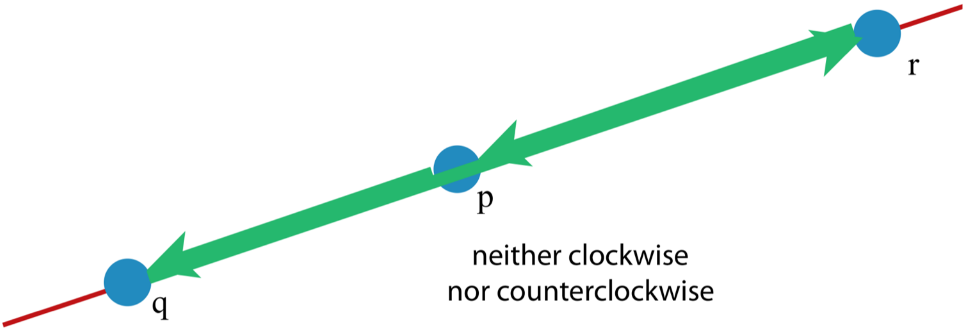
A robust technique for implementing collinearity is to use an orientation test, which determines whether three points are arranged counterclockwise, in a straight line so they are neither clockwise nor counterclockwise, or clockwise.



Clockwise orientation



Counter-clockwise orientation



Collinear

> You don't need to understand the mathematics of the orientation test to appreciate the point of what we're about to demonstrate, suffice to say that the orientation of three two-dimensional points can be computed from the sign of the determinant of a three by three matrix containing the x and y coordinates of the points in question, where the determinant happens to be the signed area of the triangle formed by the three points:
>
> $$orientation(p, q, r) = sgn\begin{vmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{vmatrix}$$

This function returns `+1` if the polyline p, q, r executes a left turn and the loop is counterclockwise, or `0` if the polyline is straight or `-1` if the polyline executes a right turn and the loop is clockwise. These values can in turn be interpreted in terms of whether the query point p is above, on, or below the line through q and r.

To cast this formula in Python, we need a sign function and a means of computing the determinant. Both of these are straightforward, although perhaps not obvious, and give us the opportunity to learn some new Python. First, the `sign()` function.

Calculating a number's sign

You may be surprised to learn — and you wouldn't be alone — that there is no built-in or library function in Python which returns the sign of a number as `-1`, `0` or `+1`. As such, we need to roll our own. The simplest solution is probably something like this:

```
>>> def sign(x):
...     if x < 0:
...         return -1
...     elif x > 0:
...         return 1
...     return 0
...
>>> sign(5)
1
>>> sign(-5)
-1
>>> sign(0)
0
```

This works well enough, though a more elegant solution would be to exploit an interesting behaviour of the `bool` type, specifically how it behaves under subtraction. Let's do a few experiments:

```
>>> False - False
0
>>> False - True
-1
>>> True - False
1
>>> True - True
0
```

Intriguingly, subtraction of `bool` objects has an integer result! In fact, when used in arithmetic operations this way, `True` is equivalent to positive one and `False` is equivalent to zero. We can use this behaviour to implement a most elegant `sign()` function:

```
>>> def sign(x):
...     return (x > 0) - (x < 0)
...
>>> sign(-5)
-1
>>> sign(5)
1
>>> sign(0)
0
```

Computing a determinant

Now we need to compute the determinant[55]. In our case this turns out to reduce down to simply:

```
det = (qx − px)(ry − py) − (qy − py)(rx − px)
```

So the definition of our `orientation()` function using tuple coordinate pairs for each point becomes simply:

```
def orientation(p, q, r):
    d = (q[0] - p[0]) * (r[1] - p[1]) - (q[1] - p[1]) * (r[0] - p[0])
    return sign(d)
```

Let's test this on on some examples. First we set up three points `a`, `b` and `c`:

```
>>> a = (0, 0)
>>> b = (4, 0)
>>> c = (4, 3)
```

Now we test the orientation of (a, b, c):

```
>>> orientation(a, b, c)
1
```

This represents a left turn, so the function returns positive one. On the other hand the orientation of (a, c, b) is negative one:

[55] Calculating determinants is a fairly straightforward operation.

```
>>> orientation(a, c, b)
-1
```

Let's introduce a fourth point, `d` which is collinear with `a` and `c`. As expected our `orientation()` function returns zero for the group (a, c, d):

```
>>> d = (8, 6)
>>> orientation(a, c, d)
0
```

Using `float` for computational geometry

Everything we have done so far uses integers which, in Python, have arbitrary precision. Since our function doesn't use any division which could result in `float` values, all of that precision is preserved. But what happens if we use floating point values as our input data?

Let's try some different values using `floats`. Here are three points which lie on a diagonal line:

```
>>> e = (0.5, 0.5)
>>> f = (12.0, 12.0)
>>> g = (24.0, 24.0)
```

As we would expect, our orientation test determines that these points are collinear:

```
>>> orientation(e, f, g)
0
```

Furthermore, moving the point `e` up a little by increasing its *y* coordinate — by even a tiny amount — gives the answer we would expect:

```
>>> e = (0.5, 0.5000000000000018)
>>> orientation(e, f, g)
1
```

Now let's increase the *y* coordinate just a little more. In fact, we'll increase it by the smallest possible amount to the next representable floating point number:

```
>>> e = (0.5, 0.5000000000000019)
>>> orientation(e, f, g)
0
```

Wow! According to our orientation function the points `e`, `f` and `g` are collinear again. This cannot possibly be! In fact, we can go through the next 23 successive floating point values with our function still reporting that the three points are collinear:

```
>>> e = (0.5, 0.5000000000000044)
>>> orientation(e, f, g)
0
```

Then we reach a value where things settle down and become well behaved again:

```
>>> e = (0.5, 0.5000000000000046)
>>> orientation(e, f, g)
1
```

Analyzing the shortcomings of `float`

What's happening here is that we've run into problems with the finite precision of Python `float`s a points very close the diagonal line. The mathematical assumptions we make in our formula about how numbers work break down due to rounding problems.

We can write a simple program to take a slice through this space, printing the value of our orientation function for all representable points on a vertical line which extends just above and below the diagonal line:

```
def sign(x):
    """Determine the sign of x.

    Returns:
        -1 if x is negative, +1 if x is positive or 0 if x is zero.
    """
    return (x > 0) - (x < 0)


def orientation(p, q, r):
    """Determine the orientation of three points in the plane.
    Args:
      p, q, r: Two-tuples representing coordinate pairs of three points.

    Returns:
        -1 if p, q, r is a turn to the right, +1 if p, q, r is a turn to the
        left, otherwise 0 if p, q, and r are collinear.
    """
    d = (q[0] - p[0]) * (r[1] - p[1]) - (q[1] - p[1]) * (r[0] - p[0])
    return sign(d)


def main():
    """
    Test whether points immediately above and below the point (0.5, 0.5)
    lie above, on, or below the line through (12.0, 12.0) and (24.0, 24.0).
    """
    px = 0.5

    pys = [0.49999999999999,
          0.49999999999999006,
          0.4999999999999901,
          0.4999999999999902,
          0.49999999999999023,
          0.4999999999999903,
          0.49999999999999034,
          0.4999999999999904,
          0.49999999999999045,
          0.4999999999999905,
          0.49999999999999056,
          0.4999999999999906,
          0.4999999999999907,
          0.49999999999999073,
          0.4999999999999908,
          0.49999999999999084,
          0.4999999999999909,
          0.49999999999999095,
          0.499999999999991,
          0.49999999999999106,
          0.4999999999999911,
          0.4999999999999912,
          0.49999999999999123,
          0.4999999999999913,
    0.49999999999999134,
    0.4999999999999914,
    0.49999999999999145,
    0.4999999999999915,
    0.49999999999999156,
    0.4999999999999916,
    0.4999999999999917,
    0.49999999999999173,
    0.4999999999999918,
    0.49999999999999184,
    0.4999999999999919,
    0.49999999999999195,
    0.499999999999992,
    0.49999999999999206,
    0.4999999999999921,
    0.4999999999999922,
    0.49999999999999223,
    0.4999999999999923,
    0.49999999999999234,
    0.4999999999999924,
    0.49999999999999245,
    0.4999999999999925,
    0.49999999999999256,
    0.4999999999999926,
    0.4999999999999927,
    0.49999999999999273,
    0.4999999999999928,
    0.49999999999999284,
    0.4999999999999929,
    0.49999999999999295,
    0.499999999999993,
    0.49999999999999306,
    0.4999999999999931,
    0.49999999999999317,
    0.4999999999999932,
    0.4999999999999933,
    0.49999999999999334,
    0.4999999999999934,
    0.49999999999999345,
    0.4999999999999935,
    0.49999999999999356,
    0.4999999999999936,
    0.49999999999999367,
    0.4999999999999937,
    0.4999999999999938,
    0.49999999999999384,
    0.4999999999999939,
    0.49999999999999395,
    0.499999999999994,
    0.49999999999999406,
    0.4999999999999941,
    0.49999999999999417,
    0.4999999999999942,
    0.4999999999999943,
    0.49999999999999434,
    0.4999999999999944,
    0.49999999999999445,
    0.4999999999999945,
    0.49999999999999456,
    0.4999999999999946,
    0.49999999999999467,
    0.4999999999999947,
    0.4999999999999948,
    0.49999999999999484,
    0.4999999999999949,
    0.49999999999999495,
    0.499999999999995,
    0.49999999999999506,
    0.4999999999999951,
    0.49999999999999517,
    0.4999999999999952,
    0.4999999999999953,
    0.49999999999999534,
    0.4999999999999954,
    0.49999999999999545,
    0.4999999999999955,
    0.49999999999999556,
    0.4999999999999956,
    0.49999999999999567,
    0.4999999999999957,
    0.4999999999999958,
    0.49999999999999584,
    0.4999999999999959,
    0.49999999999999595,
    0.499999999999996,
    0.49999999999999606,
        0.4999999999999961,
        0.49999999999999617,
        0.4999999999999962,
        0.4999999999999963,
        0.49999999999999634,
        0.4999999999999964,
        0.49999999999999645,
        0.4999999999999965,
        0.49999999999999656,
        0.4999999999999966,
        0.49999999999999667,
        0.4999999999999967,
        0.4999999999999968,
        0.49999999999999684,
        0.4999999999999969,
        0.49999999999999695,
        0.499999999999997,
        0.49999999999999706,
        0.4999999999999971,
        0.49999999999999717,
        0.4999999999999972,
        0.4999999999999973,
        0.49999999999999734,
        0.4999999999999974,
        0.49999999999999745,
        0.4999999999999975,
        0.49999999999999756,
        0.4999999999999976,
        0.49999999999999767,
        0.4999999999999977,
        0.4999999999999978,
        0.49999999999999784,
        0.4999999999999979,
        0.49999999999999795,
        0.499999999999998,
        0.49999999999999806,
        0.4999999999999981,
        0.49999999999999817,
        0.4999999999999982,
        0.4999999999999983,
        0.49999999999999833,
        0.4999999999999984,
        0.49999999999999845,
    0.4999999999999985,
    0.49999999999999856,
    0.4999999999999986,
    0.49999999999999867,
    0.4999999999999987,
    0.4999999999999988,
    0.49999999999999883,
    0.4999999999999989,
    0.49999999999999895,
    0.499999999999999,
    0.49999999999999906,
    0.4999999999999991,
    0.49999999999999917,
    0.4999999999999992,
    0.4999999999999993,
    0.49999999999999933,
    0.4999999999999994,
    0.49999999999999944,
    0.4999999999999995,
    0.49999999999999956,
    0.4999999999999996,
    0.49999999999999967,
    0.4999999999999997,
    0.4999999999999998,
    0.49999999999999983,
    0.4999999999999999,
    0.49999999999999994,  # The previous representable float less than 0.5
    0.5,
    0.5000000000000001,   # The next representable float greater than 0.5
    0.5000000000000002,
    0.5000000000000003,
    0.5000000000000004,
    0.5000000000000006,
    0.5000000000000007,
    0.5000000000000008,
    0.5000000000000009,
    0.500000000000001,
    0.5000000000000011,
    0.5000000000000012,
    0.5000000000000013,
    0.5000000000000014,
    0.5000000000000016,
    0.5000000000000017,
    0.5000000000000018,
    0.5000000000000019,
    0.500000000000002,
    0.5000000000000021,
    0.5000000000000022,
    0.5000000000000023,
    0.5000000000000024,
    0.5000000000000026,
    0.5000000000000027,
    0.5000000000000028,
    0.5000000000000029,
    0.500000000000003,
    0.5000000000000031,
    0.5000000000000032,
    0.5000000000000033,
    0.5000000000000034,
    0.5000000000000036,
    0.5000000000000037,
    0.5000000000000038,
    0.5000000000000039,
    0.500000000000004,
    0.5000000000000041,
    0.5000000000000042,
    0.5000000000000043,
    0.5000000000000044,
    0.5000000000000046,
    0.5000000000000047,
    0.5000000000000048,
    0.5000000000000049,
    0.500000000000005,
    0.5000000000000051,
    0.5000000000000052,
    0.5000000000000053,
    0.5000000000000054,
    0.5000000000000056,
    0.5000000000000057,
    0.5000000000000058,
    0.5000000000000059,
    0.500000000000006,
    0.5000000000000061,
    0.5000000000000062,
    0.5000000000000063,
    0.5000000000000064,
          0.5000000000000066,
          0.5000000000000067,
          0.5000000000000068,
          0.5000000000000069,
          0.500000000000007,
          0.5000000000000071,
          0.5000000000000072,
          0.5000000000000073,
          0.5000000000000074,
          0.5000000000000075,
          0.5000000000000077,
          0.5000000000000078,
          0.5000000000000079,
          0.500000000000008,
          0.5000000000000081,
          0.5000000000000082,
          0.5000000000000083,
          0.5000000000000084,
          0.5000000000000085,
          0.5000000000000087,
          0.5000000000000088,
          0.5000000000000089,
          0.500000000000009,
          0.5000000000000091,
          0.5000000000000092,
          0.5000000000000093,
          0.5000000000000094,
          0.5000000000000095,
          0.5000000000000097,
          0.5000000000000098,
          0.5000000000000099,
          0.50000000000001]

    q = (12.0, 12.0)
    r = (24.0, 24.0)

    for py in pys:
        p = (px, py)
        o = orientation(p, q, r)
        print("orientation(({p[0]:>3}, {p[1]:<19}) q, r) -> {o:>2}".format(p=p, o=o))


if __name__ == '__main__':
    main()
```

The program includes definitions of our `sign()` and `orientation()` functions, together with a `main()` function which runs the test. The main function includes a list of the 271 nearest representable *y*-coordinate values to 0.5. We haven't included the code to generate these values successive float values because it's far from straightforward to do in Python, and somewhat besides the point.

Then the program iterates over these `py` values and performs the orientation test each time, printing the result. The complex format string is used to get nice looking output which lines up in columns. When we look at that output we see an intricate pattern of results emerge which isn't even symmetrical around the central 0.5 value:

```
orientation((0.5, 0.50000000000001   ) q, r) ->  1
orientation((0.5, 0.5000000000000099 ) q, r) ->  1
orientation((0.5, 0.5000000000000098 ) q, r) ->  1
orientation((0.5, 0.5000000000000097 ) q, r) ->  1
orientation((0.5, 0.5000000000000095 ) q, r) ->  1
orientation((0.5, 0.5000000000000094 ) q, r) ->  1
orientation((0.5, 0.5000000000000093 ) q, r) ->  1
orientation((0.5, 0.5000000000000092 ) q, r) ->  1
orientation((0.5, 0.5000000000000091 ) q, r) ->  1
orientation((0.5, 0.500000000000009  ) q, r) ->  1
orientation((0.5, 0.5000000000000089 ) q, r) ->  1
orientation((0.5, 0.5000000000000088 ) q, r) ->  1
orientation((0.5, 0.5000000000000087 ) q, r) ->  1
orientation((0.5, 0.5000000000000085 ) q, r) ->  1
orientation((0.5, 0.5000000000000084 ) q, r) ->  1
orientation((0.5, 0.5000000000000083 ) q, r) ->  1
orientation((0.5, 0.5000000000000082 ) q, r) ->  1
orientation((0.5, 0.5000000000000081 ) q, r) ->  1
orientation((0.5, 0.500000000000008  ) q, r) ->  1
orientation((0.5, 0.5000000000000079 ) q, r) ->  1
orientation((0.5, 0.5000000000000078 ) q, r) ->  1
orientation((0.5, 0.5000000000000077 ) q, r) ->  1
orientation((0.5, 0.5000000000000075 ) q, r) ->  1
orientation((0.5, 0.5000000000000074 ) q, r) ->  1
orientation((0.5, 0.5000000000000073 ) q, r) ->  1
orientation((0.5, 0.5000000000000072 ) q, r) ->  1
orientation((0.5, 0.5000000000000071 ) q, r) ->  1
orientation((0.5, 0.500000000000007  ) q, r) ->  1
orientation((0.5, 0.5000000000000069 ) q, r) ->  1
orientation((0.5, 0.5000000000000068 ) q, r) ->  1
orientation((0.5, 0.5000000000000067 ) q, r) ->  1
orientation((0.5, 0.5000000000000066 ) q, r) ->  1
orientation((0.5, 0.5000000000000064 ) q, r) ->  1
orientation((0.5, 0.5000000000000063 ) q, r) ->  1
orientation((0.5, 0.5000000000000062 ) q, r) ->  1
orientation((0.5, 0.5000000000000061 ) q, r) ->  1
orientation((0.5, 0.500000000000006  ) q, r) ->  1
orientation((0.5, 0.5000000000000059 ) q, r) ->  1
orientation((0.5, 0.5000000000000058 ) q, r) ->  1
orientation((0.5, 0.5000000000000057 ) q, r) ->  1
orientation((0.5, 0.5000000000000056 ) q, r) ->  1
orientation((0.5, 0.5000000000000054 ) q, r) ->  1
orientation((0.5, 0.5000000000000053 ) q, r) ->  1
orientation((0.5, 0.5000000000000052 ) q, r) ->  1
orientation((0.5, 0.5000000000000051 ) q, r) ->  1
orientation((0.5, 0.500000000000005  ) q, r) ->  1
orientation((0.5, 0.5000000000000049 ) q, r) ->  1
orientation((0.5, 0.5000000000000048 ) q, r) ->  1
orientation((0.5, 0.5000000000000047 ) q, r) ->  1
orientation((0.5, 0.5000000000000046 ) q, r) ->  1
orientation((0.5, 0.5000000000000044 ) q, r) ->  0
orientation((0.5, 0.5000000000000043 ) q, r) ->  0
orientation((0.5, 0.5000000000000042 ) q, r) ->  0
orientation((0.5, 0.5000000000000041 ) q, r) ->  0
orientation((0.5, 0.500000000000004  ) q, r) ->  0
orientation((0.5, 0.5000000000000039 ) q, r) ->  0
orientation((0.5, 0.5000000000000038 ) q, r) ->  0
orientation((0.5, 0.5000000000000037 ) q, r) ->  0
orientation((0.5, 0.5000000000000036 ) q, r) ->  0
orientation((0.5, 0.5000000000000034 ) q, r) ->  0
orientation((0.5, 0.5000000000000033 ) q, r) ->  0
orientation((0.5, 0.5000000000000032 ) q, r) ->  0
orientation((0.5, 0.5000000000000031 ) q, r) ->  0
orientation((0.5, 0.500000000000003  ) q, r) ->  0
orientation((0.5, 0.5000000000000029 ) q, r) ->  0
orientation((0.5, 0.5000000000000028 ) q, r) ->  0
orientation((0.5, 0.5000000000000027 ) q, r) ->  0
orientation((0.5, 0.5000000000000026 ) q, r) ->  0
orientation((0.5, 0.5000000000000024 ) q, r) ->  0
orientation((0.5, 0.5000000000000023 ) q, r) ->  0
orientation((0.5, 0.5000000000000022 ) q, r) ->  0
orientation((0.5, 0.5000000000000021 ) q, r) ->  0
orientation((0.5, 0.500000000000002  ) q, r) ->  0
orientation((0.5, 0.5000000000000019 ) q, r) ->  0
orientation((0.5, 0.5000000000000018 ) q, r) ->  1
orientation((0.5, 0.5000000000000017 ) q, r) ->  1
orientation((0.5, 0.5000000000000016 ) q, r) ->  1
orientation((0.5, 0.5000000000000014 ) q, r) ->  1
orientation((0.5, 0.5000000000000013 ) q, r) ->  1
orientation((0.5, 0.5000000000000012 ) q, r) ->  1
orientation((0.5, 0.5000000000000011 ) q, r) ->  1
orientation((0.5, 0.500000000000001  ) q, r) ->  1
orientation((0.5, 0.5000000000000009 ) q, r) ->  0
orientation((0.5, 0.5000000000000008 ) q, r) ->  0
orientation((0.5, 0.5000000000000007 ) q, r) ->  0
orientation((0.5, 0.5000000000000006 ) q, r) ->  0
orientation((0.5, 0.5000000000000004 ) q, r) ->  0
orientation((0.5, 0.5000000000000003 ) q, r) ->  0
orientation((0.5, 0.5000000000000002 ) q, r) ->  0
orientation((0.5, 0.5000000000000001 ) q, r) ->  0
orientation((0.5, 0.5                ) q, r) ->  0
orientation((0.5, 0.49999999999999994) q, r) ->  0
orientation((0.5, 0.4999999999999999 ) q, r) ->  0
orientation((0.5, 0.49999999999999983) q, r) ->  0
orientation((0.5, 0.4999999999999998 ) q, r) ->  0
orientation((0.5, 0.4999999999999997 ) q, r) ->  0
orientation((0.5, 0.49999999999999967) q, r) ->  0
orientation((0.5, 0.4999999999999996 ) q, r) ->  0
orientation((0.5, 0.49999999999999956) q, r) ->  0
orientation((0.5, 0.4999999999999995 ) q, r) ->  0
orientation((0.5, 0.49999999999999944) q, r) ->  0
orientation((0.5, 0.4999999999999994 ) q, r) ->  0
orientation((0.5, 0.49999999999999933) q, r) ->  0
orientation((0.5, 0.4999999999999993 ) q, r) ->  0
orientation((0.5, 0.4999999999999992 ) q, r) ->  0
orientation((0.5, 0.49999999999999917) q, r) ->  0
orientation((0.5, 0.4999999999999991 ) q, r) ->  0
orientation((0.5, 0.49999999999999906) q, r) -> -1
orientation((0.5, 0.499999999999999  ) q, r) -> -1
orientation((0.5, 0.49999999999999895) q, r) -> -1
orientation((0.5, 0.4999999999999989 ) q, r) -> -1
orientation((0.5, 0.49999999999999883) q, r) -> -1
orientation((0.5, 0.4999999999999988 ) q, r) -> -1
orientation((0.5, 0.4999999999999987 ) q, r) -> -1
orientation((0.5, 0.49999999999999867) q, r) -> -1
orientation((0.5, 0.4999999999999986 ) q, r) -> -1
orientation((0.5, 0.49999999999999856) q, r) -> -1
orientation((0.5, 0.4999999999999985 ) q, r) -> -1
orientation((0.5, 0.49999999999999845) q, r) -> -1
orientation((0.5, 0.4999999999999984 ) q, r) -> -1
orientation((0.5, 0.49999999999999833) q, r) -> -1
orientation((0.5, 0.4999999999999983 ) q, r) -> -1
orientation((0.5, 0.4999999999999982 ) q, r) -> -1
orientation((0.5, 0.49999999999999817) q, r) ->  0
orientation((0.5, 0.4999999999999981 ) q, r) ->  0
orientation((0.5, 0.49999999999999806) q, r) ->  0
orientation((0.5, 0.499999999999998  ) q, r) ->  0
orientation((0.5, 0.49999999999999795) q, r) ->  0
orientation((0.5, 0.4999999999999979 ) q, r) ->  0
orientation((0.5, 0.49999999999999784) q, r) ->  0
orientation((0.5, 0.4999999999999978 ) q, r) ->  0
orientation((0.5, 0.4999999999999977 ) q, r) ->  0
orientation((0.5, 0.49999999999999767) q, r) ->  0
orientation((0.5, 0.4999999999999976 ) q, r) ->  0
orientation((0.5, 0.49999999999999756) q, r) ->  0
orientation((0.5, 0.4999999999999975 ) q, r) ->  0
orientation((0.5, 0.49999999999999745) q, r) ->  0
orientation((0.5, 0.4999999999999974 ) q, r) ->  0
orientation((0.5, 0.49999999999999734) q, r) ->  0
orientation((0.5, 0.4999999999999973 ) q, r) ->  0
orientation((0.5, 0.4999999999999972 ) q, r) ->  0
orientation((0.5, 0.49999999999999717) q, r) ->  0
orientation((0.5, 0.4999999999999971 ) q, r) ->  0
orientation((0.5, 0.49999999999999706) q, r) ->  0
orientation((0.5, 0.499999999999997  ) q, r) ->  0
orientation((0.5, 0.49999999999999695) q, r) ->  0
orientation((0.5, 0.4999999999999969 ) q, r) ->  0
orientation((0.5, 0.49999999999999684) q, r) ->  0
orientation((0.5, 0.4999999999999968 ) q, r) ->  0
orientation((0.5, 0.4999999999999967 ) q, r) ->  0
orientation((0.5, 0.49999999999999667) q, r) ->  0
orientation((0.5, 0.4999999999999966 ) q, r) ->  0
orientation((0.5, 0.49999999999999656) q, r) ->  0
orientation((0.5, 0.4999999999999965 ) q, r) ->  0
orientation((0.5, 0.49999999999999645) q, r) ->  0
orientation((0.5, 0.4999999999999964 ) q, r) ->  0
orientation((0.5, 0.49999999999999634) q, r) ->  0
orientation((0.5, 0.4999999999999963 ) q, r) ->  0
orientation((0.5, 0.4999999999999962 ) q, r) ->  0
orientation((0.5, 0.49999999999999617) q, r) ->  0
orientation((0.5, 0.4999999999999961 ) q, r) ->  0
orientation((0.5, 0.49999999999999606) q, r) ->  0
orientation((0.5, 0.499999999999996  ) q, r) ->  0
orientation((0.5, 0.49999999999999595) q, r) ->  0
orientation((0.5, 0.4999999999999959 ) q, r) ->  0
orientation((0.5, 0.49999999999999584) q, r) ->  0
orientation((0.5, 0.4999999999999958 ) q, r) ->  0
orientation((0.5, 0.4999999999999957 ) q, r) ->  0
orientation((0.5, 0.49999999999999567) q, r) ->  0
orientation((0.5, 0.4999999999999956 ) q, r) ->  0
orientation((0.5, 0.49999999999999556) q, r) ->  0
orientation((0.5, 0.4999999999999955 ) q, r) -> -1
orientation((0.5, 0.49999999999999545) q, r) -> -1
orientation((0.5, 0.4999999999999954 ) q, r) -> -1
orientation((0.5, 0.49999999999999534) q, r) -> -1
orientation((0.5, 0.4999999999999953 ) q, r) -> -1
orientation((0.5, 0.4999999999999952 ) q, r) -> -1
orientation((0.5, 0.49999999999999517) q, r) -> -1
orientation((0.5, 0.4999999999999951 ) q, r) -> -1
orientation((0.5, 0.49999999999999506) q, r) -> -1
orientation((0.5, 0.499999999999995  ) q, r) -> -1
orientation((0.5, 0.49999999999999495) q, r) -> -1
orientation((0.5, 0.4999999999999949 ) q, r) -> -1
orientation((0.5, 0.49999999999999484) q, r) -> -1
orientation((0.5, 0.4999999999999948 ) q, r) -> -1
orientation((0.5, 0.4999999999999947 ) q, r) -> -1
orientation((0.5, 0.49999999999999467) q, r) -> -1
orientation((0.5, 0.4999999999999946 ) q, r) -> -1
orientation((0.5, 0.49999999999999456) q, r) -> -1
orientation((0.5, 0.4999999999999945 ) q, r) -> -1
orientation((0.5, 0.49999999999999445) q, r) -> -1
orientation((0.5, 0.4999999999999944 ) q, r) -> -1
orientation((0.5, 0.49999999999999434) q, r) -> -1
orientation((0.5, 0.4999999999999943 ) q, r) -> -1
orientation((0.5, 0.4999999999999942 ) q, r) -> -1
orientation((0.5, 0.49999999999999417) q, r) -> -1
orientation((0.5, 0.4999999999999941 ) q, r) -> -1
orientation((0.5, 0.49999999999999406) q, r) -> -1
orientation((0.5, 0.499999999999994  ) q, r) -> -1
orientation((0.5, 0.49999999999999395) q, r) -> -1
orientation((0.5, 0.4999999999999939 ) q, r) -> -1
orientation((0.5, 0.49999999999999384) q, r) -> -1
orientation((0.5, 0.4999999999999938 ) q, r) -> -1
orientation((0.5, 0.4999999999999937 ) q, r) -> -1
orientation((0.5, 0.49999999999999367) q, r) -> -1
orientation((0.5, 0.4999999999999936 ) q, r) -> -1
orientation((0.5, 0.49999999999999356) q, r) -> -1
orientation((0.5, 0.4999999999999935 ) q, r) -> -1
orientation((0.5, 0.49999999999999345) q, r) -> -1
orientation((0.5, 0.4999999999999934 ) q, r) -> -1
orientation((0.5, 0.49999999999999334) q, r) -> -1
orientation((0.5, 0.4999999999999933 ) q, r) -> -1
orientation((0.5, 0.4999999999999932 ) q, r) -> -1
orientation((0.5, 0.49999999999999317) q, r) -> -1
orientation((0.5, 0.4999999999999931 ) q, r) -> -1
orientation((0.5, 0.49999999999999306) q, r) -> -1
orientation((0.5, 0.499999999999993  ) q, r) -> -1
orientation((0.5, 0.49999999999999295) q, r) -> -1
orientation((0.5, 0.4999999999999929 ) q, r) -> -1
orientation((0.5, 0.49999999999999284) q, r) -> -1
orientation((0.5, 0.4999999999999928 ) q, r) -> -1
orientation((0.5, 0.49999999999999273) q, r) -> -1
orientation((0.5, 0.4999999999999927 ) q, r) -> -1
orientation((0.5, 0.4999999999999926 ) q, r) -> -1
orientation((0.5, 0.49999999999999256) q, r) -> -1
orientation((0.5, 0.4999999999999925 ) q, r) -> -1
orientation((0.5, 0.49999999999999245) q, r) -> -1
orientation((0.5, 0.4999999999999924 ) q, r) -> -1
orientation((0.5, 0.49999999999999234) q, r) -> -1
orientation((0.5, 0.4999999999999923 ) q, r) -> -1
orientation((0.5, 0.49999999999999223) q, r) -> -1
orientation((0.5, 0.4999999999999922 ) q, r) -> -1
orientation((0.5, 0.4999999999999921 ) q, r) -> -1
orientation((0.5, 0.49999999999999206) q, r) -> -1
orientation((0.5, 0.499999999999992  ) q, r) -> -1
orientation((0.5, 0.49999999999999195) q, r) -> -1
orientation((0.5, 0.4999999999999919 ) q, r) -> -1
orientation((0.5, 0.49999999999999184) q, r) -> -1
orientation((0.5, 0.4999999999999918 ) q, r) -> -1
orientation((0.5, 0.49999999999999173) q, r) -> -1
orientation((0.5, 0.4999999999999917 ) q, r) -> -1
orientation((0.5, 0.4999999999999916 ) q, r) -> -1
orientation((0.5, 0.49999999999999156) q, r) -> -1
orientation((0.5, 0.4999999999999915 ) q, r) -> -1
orientation((0.5, 0.49999999999999145) q, r) -> -1
orientation((0.5, 0.4999999999999914 ) q, r) -> -1
orientation((0.5, 0.49999999999999134) q, r) -> -1
orientation((0.5, 0.4999999999999913 ) q, r) -> -1
orientation((0.5, 0.49999999999999123) q, r) -> -1
orientation((0.5, 0.4999999999999912 ) q, r) -> -1
orientation((0.5, 0.4999999999999911 ) q, r) -> -1
orientation((0.5, 0.49999999999999106) q, r) -> -1
orientation((0.5, 0.499999999999991  ) q, r) -> -1
orientation((0.5, 0.49999999999999095) q, r) -> -1
orientation((0.5, 0.4999999999999909 ) q, r) -> -1
orientation((0.5, 0.49999999999999084) q, r) -> -1
orientation((0.5, 0.4999999999999908 ) q, r) -> -1
orientation((0.5, 0.49999999999999073) q, r) -> -1
orientation((0.5, 0.4999999999999907 ) q, r) -> -1
orientation((0.5, 0.4999999999999906 ) q, r) -> -1
orientation((0.5, 0.49999999999999056) q, r) -> -1
orientation((0.5, 0.4999999999999905 ) q, r) -> -1
orientation((0.5, 0.49999999999999045) q, r) -> -1
orientation((0.5, 0.4999999999999904 ) q, r) -> -1
orientation((0.5, 0.49999999999999034) q, r) -> -1
orientation((0.5, 0.4999999999999903 ) q, r) -> -1
orientation((0.5, 0.49999999999999023) q, r) -> -1
orientation((0.5, 0.4999999999999902 ) q, r) -> -1
orientation((0.5, 0.4999999999999901 ) q, r) -> -1
orientation((0.5, 0.49999999999999006) q, r) -> -1
orientation((0.5, 0.49999999999999   ) q, r) -> -1
```

By this point you should at least be wary of using floating point arithmetic for geometric computation. Lest you think this can easily be solved by introducing a tolerance value, or some other clunky solution, we'll save you the bother by pointing out that doing do merely moves these fringing effects to the edge of the tolerance zone.

What to do? Fortunately, as we alluded to at the beginning of this tale, Python gives us a solution into the form of the rational numbers, implemented as the fractional type.

Using `Fraction`s to avoid rounding errors

Let's make a small change to our program, converting all numbers to `Fraction`s before proceeding with the computation. We'll do this by modifying the `orientation()` to convert each of it's three arguments from a tuple containing a pair of numeric objects into specifically a pair of `Fraction`s. As we know, the `Fraction` constructor accepts a selection of numeric types, including `float`:

```
def orientation(p, q, r):
    """Determine the orientation of three points in the plane.

    Args:
      p, q, r: Two-tuples representing coordinate pairs of three points.

    Returns:
        -1 if p, q, r is a turn to the right, +1 if p, q, r is a turn to the
        left, otherwise 0 if p, q, and r are collinear.
    """
    p = (Fraction(p[0]), Fraction(p[1]))
    q = (Fraction(q[0]), Fraction(q[1]))
    r = (Fraction(r[0]), Fraction(r[1]))

    d = (q[0] - p[0]) * (r[1] - p[1]) - (q[1] - p[1]) * (r[0] - p[0])
    return sign(d)
```

The variable `d` will now also be a `Fraction` and the `sign()` function will work as expected with this type since it only uses comparison to zero.

Let's run our modified example:

```
orientation((0.5, 0.49999999999999   ) q, r) -> -1
orientation((0.5, 0.49999999999999006) q, r) -> -1
orientation((0.5, 0.4999999999999901 ) q, r) -> -1
orientation((0.5, 0.4999999999999902 ) q, r) -> -1
orientation((0.5, 0.49999999999999023) q, r) -> -1
orientation((0.5, 0.4999999999999903 ) q, r) -> -1
orientation((0.5, 0.49999999999999034) q, r) -> -1
orientation((0.5, 0.4999999999999904 ) q, r) -> -1
orientation((0.5, 0.49999999999999045) q, r) -> -1
orientation((0.5, 0.4999999999999905 ) q, r) -> -1
orientation((0.5, 0.49999999999999056) q, r) -> -1
orientation((0.5, 0.4999999999999906 ) q, r) -> -1
orientation((0.5, 0.4999999999999907 ) q, r) -> -1
orientation((0.5, 0.49999999999999073) q, r) -> -1
orientation((0.5, 0.4999999999999908 ) q, r) -> -1
orientation((0.5, 0.49999999999999084) q, r) -> -1
orientation((0.5, 0.4999999999999909 ) q, r) -> -1
orientation((0.5, 0.49999999999999095) q, r) -> -1
orientation((0.5, 0.499999999999991  ) q, r) -> -1
orientation((0.5, 0.49999999999999106) q, r) -> -1
orientation((0.5, 0.4999999999999911 ) q, r) -> -1
orientation((0.5, 0.4999999999999912 ) q, r) -> -1
orientation((0.5, 0.49999999999999123) q, r) -> -1
orientation((0.5, 0.4999999999999913 ) q, r) -> -1
orientation((0.5, 0.49999999999999134) q, r) -> -1
orientation((0.5, 0.4999999999999914 ) q, r) -> -1
orientation((0.5, 0.49999999999999145) q, r) -> -1
orientation((0.5, 0.4999999999999915 ) q, r) -> -1
orientation((0.5, 0.49999999999999156) q, r) -> -1
orientation((0.5, 0.4999999999999916 ) q, r) -> -1
orientation((0.5, 0.4999999999999917 ) q, r) -> -1
orientation((0.5, 0.49999999999999173) q, r) -> -1
orientation((0.5, 0.4999999999999918 ) q, r) -> -1
orientation((0.5, 0.49999999999999184) q, r) -> -1
orientation((0.5, 0.4999999999999919 ) q, r) -> -1
orientation((0.5, 0.49999999999999195) q, r) -> -1
orientation((0.5, 0.499999999999992  ) q, r) -> -1
orientation((0.5, 0.49999999999999206) q, r) -> -1
orientation((0.5, 0.4999999999999921 ) q, r) -> -1
orientation((0.5, 0.4999999999999922 ) q, r) -> -1
orientation((0.5, 0.49999999999999223) q, r) -> -1
orientation((0.5, 0.4999999999999923 ) q, r) -> -1
orientation((0.5, 0.49999999999999234) q, r) -> -1
orientation((0.5, 0.4999999999999924 ) q, r) -> -1
orientation((0.5, 0.49999999999999245) q, r) -> -1
orientation((0.5, 0.4999999999999925 ) q, r) -> -1
orientation((0.5, 0.49999999999999256) q, r) -> -1
orientation((0.5, 0.4999999999999926 ) q, r) -> -1
orientation((0.5, 0.4999999999999927 ) q, r) -> -1
orientation((0.5, 0.49999999999999273) q, r) -> -1
orientation((0.5, 0.4999999999999928 ) q, r) -> -1
orientation((0.5, 0.49999999999999284) q, r) -> -1
orientation((0.5, 0.4999999999999929 ) q, r) -> -1
orientation((0.5, 0.49999999999999295) q, r) -> -1
orientation((0.5, 0.499999999999993  ) q, r) -> -1
orientation((0.5, 0.49999999999999306) q, r) -> -1
orientation((0.5, 0.4999999999999931 ) q, r) -> -1
orientation((0.5, 0.49999999999999317) q, r) -> -1
orientation((0.5, 0.4999999999999932 ) q, r) -> -1
orientation((0.5, 0.4999999999999933 ) q, r) -> -1
orientation((0.5, 0.49999999999999334) q, r) -> -1
orientation((0.5, 0.4999999999999934 ) q, r) -> -1
orientation((0.5, 0.49999999999999345) q, r) -> -1
orientation((0.5, 0.4999999999999935 ) q, r) -> -1
orientation((0.5, 0.49999999999999356) q, r) -> -1
orientation((0.5, 0.4999999999999936 ) q, r) -> -1
orientation((0.5, 0.49999999999999367) q, r) -> -1
orientation((0.5, 0.4999999999999937 ) q, r) -> -1
orientation((0.5, 0.4999999999999938 ) q, r) -> -1
orientation((0.5, 0.49999999999999384) q, r) -> -1
orientation((0.5, 0.4999999999999939 ) q, r) -> -1
orientation((0.5, 0.49999999999999395) q, r) -> -1
orientation((0.5, 0.499999999999994  ) q, r) -> -1
orientation((0.5, 0.49999999999999406) q, r) -> -1
orientation((0.5, 0.4999999999999941 ) q, r) -> -1
orientation((0.5, 0.49999999999999417) q, r) -> -1
orientation((0.5, 0.4999999999999942 ) q, r) -> -1
orientation((0.5, 0.4999999999999943 ) q, r) -> -1
orientation((0.5, 0.49999999999999434) q, r) -> -1
orientation((0.5, 0.4999999999999944 ) q, r) -> -1
orientation((0.5, 0.49999999999999445) q, r) -> -1
orientation((0.5, 0.4999999999999945 ) q, r) -> -1
orientation((0.5, 0.49999999999999456) q, r) -> -1
orientation((0.5, 0.4999999999999946 ) q, r) -> -1
orientation((0.5, 0.49999999999999467) q, r) -> -1
orientation((0.5, 0.4999999999999947 ) q, r) -> -1
orientation((0.5, 0.4999999999999948 ) q, r) -> -1
orientation((0.5, 0.49999999999999484) q, r) -> -1
orientation((0.5, 0.4999999999999949 ) q, r) -> -1
orientation((0.5, 0.49999999999999495) q, r) -> -1
orientation((0.5, 0.499999999999995  ) q, r) -> -1
orientation((0.5, 0.49999999999999506) q, r) -> -1
orientation((0.5, 0.4999999999999951 ) q, r) -> -1
orientation((0.5, 0.49999999999999517) q, r) -> -1
orientation((0.5, 0.4999999999999952 ) q, r) -> -1
orientation((0.5, 0.4999999999999953 ) q, r) -> -1
orientation((0.5, 0.49999999999999534) q, r) -> -1
orientation((0.5, 0.4999999999999954 ) q, r) -> -1
orientation((0.5, 0.49999999999999545) q, r) -> -1
orientation((0.5, 0.4999999999999955 ) q, r) -> -1
orientation((0.5, 0.49999999999999556) q, r) -> -1
orientation((0.5, 0.4999999999999956 ) q, r) -> -1
orientation((0.5, 0.49999999999999567) q, r) -> -1
orientation((0.5, 0.4999999999999957 ) q, r) -> -1
orientation((0.5, 0.4999999999999958 ) q, r) -> -1
orientation((0.5, 0.49999999999999584) q, r) -> -1
orientation((0.5, 0.4999999999999959 ) q, r) -> -1
orientation((0.5, 0.49999999999999595) q, r) -> -1
orientation((0.5, 0.499999999999996  ) q, r) -> -1
orientation((0.5, 0.49999999999999606) q, r) -> -1
orientation((0.5, 0.4999999999999961 ) q, r) -> -1
orientation((0.5, 0.49999999999999617) q, r) -> -1
orientation((0.5, 0.4999999999999962 ) q, r) -> -1
orientation((0.5, 0.4999999999999963 ) q, r) -> -1
orientation((0.5, 0.49999999999999634) q, r) -> -1
orientation((0.5, 0.4999999999999964 ) q, r) -> -1
orientation((0.5, 0.49999999999999645) q, r) -> -1
orientation((0.5, 0.4999999999999965 ) q, r) -> -1
orientation((0.5, 0.49999999999999656) q, r) -> -1
orientation((0.5, 0.4999999999999966 ) q, r) -> -1
orientation((0.5, 0.49999999999999667) q, r) -> -1
orientation((0.5, 0.4999999999999967 ) q, r) -> -1
orientation((0.5, 0.4999999999999968 ) q, r) -> -1
orientation((0.5, 0.49999999999999684) q, r) -> -1
orientation((0.5, 0.4999999999999969 ) q, r) -> -1
orientation((0.5, 0.49999999999999695) q, r) -> -1
orientation((0.5, 0.499999999999997  ) q, r) -> -1
orientation((0.5, 0.49999999999999706) q, r) -> -1
orientation((0.5, 0.4999999999999971 ) q, r) -> -1
orientation((0.5, 0.49999999999999717) q, r) -> -1
orientation((0.5, 0.4999999999999972 ) q, r) -> -1
orientation((0.5, 0.4999999999999973 ) q, r) -> -1
orientation((0.5, 0.49999999999999734) q, r) -> -1
orientation((0.5, 0.4999999999999974 ) q, r) -> -1
orientation((0.5, 0.49999999999999745) q, r) -> -1
orientation((0.5, 0.4999999999999975 ) q, r) -> -1
orientation((0.5, 0.49999999999999756) q, r) -> -1
orientation((0.5, 0.4999999999999976 ) q, r) -> -1
orientation((0.5, 0.49999999999999767) q, r) -> -1
orientation((0.5, 0.4999999999999977 ) q, r) -> -1
orientation((0.5, 0.4999999999999978 ) q, r) -> -1
orientation((0.5, 0.49999999999999784) q, r) -> -1
orientation((0.5, 0.4999999999999979 ) q, r) -> -1
orientation((0.5, 0.49999999999999795) q, r) -> -1
orientation((0.5, 0.499999999999998  ) q, r) -> -1
orientation((0.5, 0.49999999999999806) q, r) -> -1
orientation((0.5, 0.4999999999999981 ) q, r) -> -1
orientation((0.5, 0.49999999999999817) q, r) -> -1
orientation((0.5, 0.4999999999999982 ) q, r) -> -1
orientation((0.5, 0.4999999999999983 ) q, r) -> -1
orientation((0.5, 0.49999999999999833) q, r) -> -1
orientation((0.5, 0.4999999999999984 ) q, r) -> -1
orientation((0.5, 0.49999999999999845) q, r) -> -1
orientation((0.5, 0.4999999999999985 ) q, r) -> -1
orientation((0.5, 0.49999999999999856) q, r) -> -1
orientation((0.5, 0.4999999999999986 ) q, r) -> -1
orientation((0.5, 0.49999999999999867) q, r) -> -1
orientation((0.5, 0.4999999999999987 ) q, r) -> -1
orientation((0.5, 0.4999999999999988 ) q, r) -> -1
orientation((0.5, 0.49999999999999883) q, r) -> -1
orientation((0.5, 0.4999999999999989 ) q, r) -> -1
orientation((0.5, 0.49999999999999895) q, r) -> -1
orientation((0.5, 0.499999999999999  ) q, r) -> -1
orientation((0.5, 0.49999999999999906) q, r) -> -1
orientation((0.5, 0.4999999999999991 ) q, r) -> -1
orientation((0.5, 0.49999999999999917) q, r) -> -1
orientation((0.5, 0.4999999999999992 ) q, r) -> -1
orientation((0.5, 0.4999999999999993 ) q, r) -> -1
orientation((0.5, 0.49999999999999933) q, r) -> -1
orientation((0.5, 0.4999999999999994 ) q, r) -> -1
orientation((0.5, 0.49999999999999944) q, r) -> -1
orientation((0.5, 0.4999999999999995 ) q, r) -> -1
orientation((0.5, 0.49999999999999956) q, r) -> -1
orientation((0.5, 0.4999999999999996 ) q, r) -> -1
orientation((0.5, 0.49999999999999967) q, r) -> -1
orientation((0.5, 0.4999999999999997 ) q, r) -> -1
orientation((0.5, 0.4999999999999998 ) q, r) -> -1
orientation((0.5, 0.49999999999999983) q, r) -> -1
orientation((0.5, 0.4999999999999999 ) q, r) -> -1
orientation((0.5, 0.49999999999999994) q, r) -> -1
orientation((0.5, 0.5                ) q, r) ->  0
orientation((0.5, 0.5000000000000001 ) q, r) ->  1
orientation((0.5, 0.5000000000000002 ) q, r) ->  1
orientation((0.5, 0.5000000000000003 ) q, r) ->  1
orientation((0.5, 0.5000000000000004 ) q, r) ->  1
orientation((0.5, 0.5000000000000006 ) q, r) ->  1
orientation((0.5, 0.5000000000000007 ) q, r) ->  1
orientation((0.5, 0.5000000000000008 ) q, r) ->  1
orientation((0.5, 0.5000000000000009 ) q, r) ->  1
orientation((0.5, 0.500000000000001  ) q, r) ->  1
orientation((0.5, 0.5000000000000011 ) q, r) ->  1
orientation((0.5, 0.5000000000000012 ) q, r) ->  1
orientation((0.5, 0.5000000000000013 ) q, r) ->  1
orientation((0.5, 0.5000000000000014 ) q, r) ->  1
orientation((0.5, 0.5000000000000016 ) q, r) ->  1
orientation((0.5, 0.5000000000000017 ) q, r) ->  1
orientation((0.5, 0.5000000000000018 ) q, r) ->  1
orientation((0.5, 0.5000000000000019 ) q, r) ->  1
orientation((0.5, 0.500000000000002  ) q, r) ->  1
orientation((0.5, 0.5000000000000021 ) q, r) ->  1
orientation((0.5, 0.5000000000000022 ) q, r) ->  1
orientation((0.5, 0.5000000000000023 ) q, r) ->  1
orientation((0.5, 0.5000000000000024 ) q, r) ->  1
orientation((0.5, 0.5000000000000026 ) q, r) ->  1
orientation((0.5, 0.5000000000000027 ) q, r) ->  1
orientation((0.5, 0.5000000000000028 ) q, r) ->  1
orientation((0.5, 0.5000000000000029 ) q, r) ->  1
orientation((0.5, 0.500000000000003  ) q, r) ->  1
orientation((0.5, 0.5000000000000031 ) q, r) ->  1
orientation((0.5, 0.5000000000000032 ) q, r) ->  1
orientation((0.5, 0.5000000000000033 ) q, r) ->  1
orientation((0.5, 0.5000000000000034 ) q, r) ->  1
orientation((0.5, 0.5000000000000036 ) q, r) ->  1
orientation((0.5, 0.5000000000000037 ) q, r) ->  1
orientation((0.5, 0.5000000000000038 ) q, r) ->  1
orientation((0.5, 0.5000000000000039 ) q, r) ->  1
orientation((0.5, 0.500000000000004  ) q, r) ->  1
orientation((0.5, 0.5000000000000041 ) q, r) ->  1
orientation((0.5, 0.5000000000000042 ) q, r) ->  1
orientation((0.5, 0.5000000000000043 ) q, r) ->  1
orientation((0.5, 0.5000000000000044 ) q, r) ->  1
orientation((0.5, 0.5000000000000046 ) q, r) ->  1
orientation((0.5, 0.5000000000000047 ) q, r) ->  1
orientation((0.5, 0.5000000000000048 ) q, r) ->  1
orientation((0.5, 0.5000000000000049 ) q, r) ->  1
orientation((0.5, 0.500000000000005  ) q, r) ->  1
orientation((0.5, 0.5000000000000051 ) q, r) ->  1
orientation((0.5, 0.5000000000000052 ) q, r) ->  1
orientation((0.5, 0.5000000000000053 ) q, r) ->  1
orientation((0.5, 0.5000000000000054 ) q, r) ->  1
orientation((0.5, 0.5000000000000056 ) q, r) ->  1
orientation((0.5, 0.5000000000000057 ) q, r) ->  1
orientation((0.5, 0.5000000000000058 ) q, r) ->  1
orientation((0.5, 0.5000000000000059 ) q, r) ->  1
orientation((0.5, 0.500000000000006  ) q, r) ->  1
orientation((0.5, 0.5000000000000061 ) q, r) ->  1
orientation((0.5, 0.5000000000000062 ) q, r) ->  1
orientation((0.5, 0.5000000000000063 ) q, r) ->  1
orientation((0.5, 0.5000000000000064 ) q, r) ->  1
orientation((0.5, 0.5000000000000066 ) q, r) ->  1
orientation((0.5, 0.5000000000000067 ) q, r) ->  1
orientation((0.5, 0.5000000000000068 ) q, r) ->  1
orientation((0.5, 0.5000000000000069 ) q, r) ->  1
orientation((0.5, 0.500000000000007  ) q, r) ->  1
orientation((0.5, 0.5000000000000071 ) q, r) ->  1
orientation((0.5, 0.5000000000000072 ) q, r) ->  1
orientation((0.5, 0.5000000000000073 ) q, r) ->  1
orientation((0.5, 0.5000000000000074 ) q, r) ->  1
orientation((0.5, 0.5000000000000075 ) q, r) ->  1
orientation((0.5, 0.5000000000000077 ) q, r) ->  1
orientation((0.5, 0.5000000000000078 ) q, r) ->  1
orientation((0.5, 0.5000000000000079 ) q, r) ->  1
orientation((0.5, 0.500000000000008  ) q, r) ->  1
orientation((0.5, 0.5000000000000081 ) q, r) ->  1
orientation((0.5, 0.5000000000000082 ) q, r) ->  1
orientation((0.5, 0.5000000000000083 ) q, r) ->  1
orientation((0.5, 0.5000000000000084 ) q, r) ->  1
orientation((0.5, 0.5000000000000085 ) q, r) ->  1
orientation((0.5, 0.5000000000000087 ) q, r) ->  1
orientation((0.5, 0.5000000000000088 ) q, r) ->  1
orientation((0.5, 0.5000000000000089 ) q, r) ->  1
orientation((0.5, 0.500000000000009  ) q, r) ->  1
orientation((0.5, 0.5000000000000091 ) q, r) ->  1
orientation((0.5, 0.5000000000000092 ) q, r) ->  1
orientation((0.5, 0.5000000000000093 ) q, r) ->  1
orientation((0.5, 0.5000000000000094 ) q, r) ->  1
orientation((0.5, 0.5000000000000095 ) q, r) ->  1
orientation((0.5, 0.5000000000000097 ) q, r) ->  1
orientation((0.5, 0.5000000000000098 ) q, r) ->  1
orientation((0.5, 0.5000000000000099 ) q, r) ->  1
orientation((0.5, 0.50000000000001   ) q, r) ->  1
```

Using `Fraction`s internally, our `orientation()` function gets the full benefit of exact arithmetic with effectively infinite precision and consequently produces an *exact* result with only one position of `p` being reported as collinear with `q` and `r`.

Representing rounding issues graphically

Going further, we can map out th e behaviour of our orientation functions by hooking up our program to the BMP image file writer we created in chapter 9 of *The Python Apprentice*[56]. By evaluating `orientation()` for every point in a tiny square region straddling out diagonal line, we can produce a view of how our diagonal line is represented using different number types.

The code change is straightforward. First, we import our `bmp` module we creating in *The Python Apprentice*; we've included a copy in the example code for this book too:

```
import bmp
```

Then we replace the code which iterates through our line transect with code to so something similar in two dimensions. This new code uses nested list comprehensions to produce nested lists representing pixel data. We use a dictionary of three entries called `color` to map from -1, 0, and +1 orientation values to black, mid-grey, and white respectively:

```
color = {-1: 0, 0: 127, +1: 255}
```

The inner comprehension produces the pixels within each row, and the outer comprehension assembles the rows into an image. We reverse the order of the rows using a call to `reversed()` to get our co-ordinate axes to correspond to image format conventions:

```
pixels = [[color[orientation((px, py), q, r)] for px in pys] for py in reversed(pys)]
```

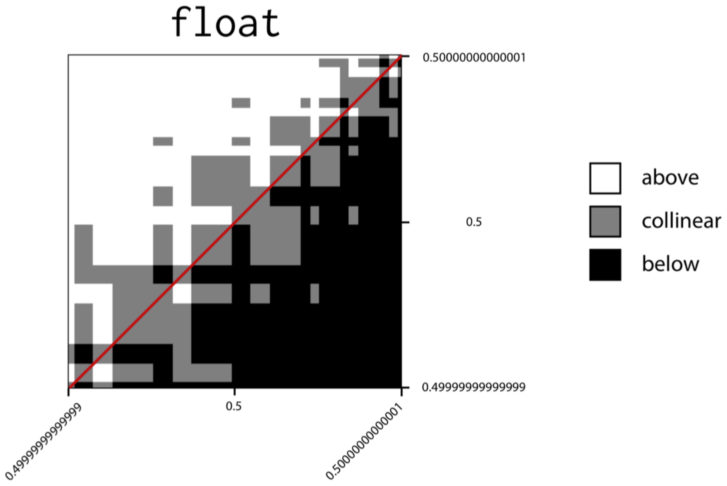
The result of this call is that `pixels` is now a list of lists of integers, where each integer has a value of 0, 127 or 255 depending on whether the pixel is below, on, or above the line. These in turn will correspond to the shades of black, grey and white in the image.

Finally, the pixel data structure is written out as a BMP file with a call to `bmp.write_-grayscale()`:

[56] https://leanpub.com/python-apprentice

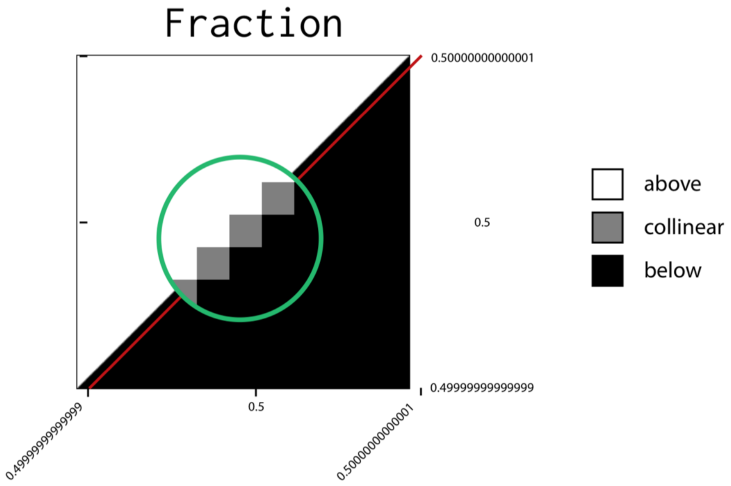
```
bmp.write_grayscale('above_below.bmp', pixels)
```

If we now temporarily disable out use of `Fraction` in the orientation number, we get a map of the above and below the diagonal for `float` computations:



Collinearity tests with `float`

Whereas with the rational number code active, we get a much more sensible and intuitive result.



Collinearity tests with `Fraction`

Summary

In this chapter on numeric and scalar types we covered a lot of ground:

- Basic numeric types
  - Reviewed the capabilities of the `int` and `float` types and looked at how to query `sys.float_info` to get details of the `float` implementation.
  - Understood the limitations of finite precision of the `float` type and the impact this has on representing numbers in binary.
- `decimal.Decimal`
  - We introduced the `decimal` module which provides another floating point numeric type founded on base 10 and with user configurable precision.
  - We explained that `Decimal` is preferred for certain financial applications, such as accounting where the problem domain is inherently decimal in nature.
  - We highlighted some key differences in behaviour between `Decimal` and the other numeric types, particularly in the behaviour of integer division and modulus operators, which for `Decimal` truncates towards zero, but for `int` and `float` round towards negative infinity. This has implications for correctly writing functions which may need to work correctly with various number types.
- We demonstrated support for rational numbers using the `Fraction` type from the `fractions` module, showing how to construct and manipulate `Fraction` values using arithmetic.
- `complex`
  - We introduced the built-in `complex` type and gave an overview of complex number support in Python, including the `cmath` module which includes complex equivalents of the functions in the `math` module.
  - We gave a brief demonstration of the utility of complex numbers in Python by using them to solve an simple electrical engineering problem, determining the properties of AC current in a simple circuit.
- built-in features
  - `abs()` for computing the distance of a number from zero — which also works for `complex` numbers.
  - `round()` which rounds to a specified *decimal* precision, and the surprises this can lead to when used on `float`s which are internally represented in binary.
  - We reviewed the literal syntax for numbers in different bases.
  - And showed how to convert to strings in these literal forms using the built-in `bin()`, `oct()` and `hex()` functions.

  - We demonstrated how to convert from strings in bases 2 to 36 inclusive by using the `base` argument to the `int()` constructor, and how to convert from any numeric literal in string form by specifying base zero.
- Date and time
  - We covered the representation of dates, times and compound date-time objects using the facilities of the `datetime` module.
  - We explained the difference between naïve and timezone-aware times
  - And the many named constructors available for constructing time related objects.
  - We showed string formatting of time and date objects and how to parse these strings back into datetime objects.
  - We explained how durations can be modelled with the `timedelta` object and how to perform arithmetic on `datetime`s.
  - We demonstrated the basic time zone support available in Python 3 with the `timezone` class and referred you to the third-party `pytz` package for more comprehensive support.
- We showed how regular `float`s can be unsuitable for geometric computation owing to finite precision, and how to solve this by deploying the `Fraction` type in a geometric computation.

Chapter 7 - Iterables and Iteration

In this chapter we'll be taking a deeper look at iterables and iteration in Python, including topics such as more advanced comprehensions, some language and library features which support a more functional style of programming[57], and the protocols underlying iteration. This chapter builds upon the contents of chapter 7 of "The Python Apprentice"[58], and, as with that previous material, you'll find that these techniques and tools can help you write more expressive, elegant, and even beautiful code.

Review of comprehensions

Comprehensions are a sort of short-hand syntax for creating collections and iterable objects of various types. For example, a list comprehension creates a new list object from an existing sequence, and looks like this:

```
l = [i * 2 for i in range(10)]
```

Here, we've take a range sequence — in this case the integers from zero to nine — and created a new list where each entry is twice the corresponding value from the original sequence. This new list is a completely normal list, just like any other list made using any other approach.

There are comprehension syntaxes for creating dictionaries, sets, and generators as well as lists, and all of the syntaxes work in essentially the same way:

[57] In imperative programming languages, such as Python, the program makes progress by executing statements. In functional programming languages the program progresses purely by evaluating expressions. Since Python can evaluate expressions too, it is possible to program in a functional style in Python.

[58] https://leanpub.com/python-apprentice

```
>>> d = {i: i * 2 for i in range(10)}
>>> type(d)
<class 'dict'>
>>> s = {i for i in range(10)}
>>> type(s)
<class 'set'>
>>> g = (i for i in range(10))
>>> type(g)
<class 'generator'>
```

Multi-input comprehensions

All of the comprehension examples we've seen up to now use only a single input sequence, but comprehensions actually allow you to use as many input sequences as you want. Likewise, a comprehension can use as many if-clauses as you need as well.

For example, this comprehension uses two input ranges to create a set of points on a 5-by-3 grid:

```
>>> [(x, y) for x in range(5) for y in range(3)]
[(0, 0), (0, 1), (0, 2),
 (1, 0), (1, 1), (1, 2),
 (2, 0), (2, 1), (2, 2),
 (3, 0), (3, 1), (3, 2),
 (4, 0), (4, 1), (4, 2)]
```

This produces list containing the so-called *cartesian product* of the two input ranges, `range(5)` and `range(3)`.

The way to read this is as a set of nested for-loops, where the later for-clauses are nested inside the earlier for-clauses, and the result-expression of the comprehension is executed inside the innermost — or last — for-loop.

Equivalence with nested for-loops

To help clarify this, the corresponding nested for-loop structure would look like this:

```
points = []
for x in range(5):
    for y in range(3):
        points.append((x, y))
```

The outer for-loop which binds to the `x` variable corresponds to the first for-clause in the comprehension. The inner for-loop which binds to the `y` variable corresponds to the second for-clause in the comprehension. The output expression in the comprehension where we create the tuple is nested inside the inner-most for-loop.

The obvious benefit of the comprehension syntax is that you don't need to create the list variable and then repeatedly append elements to it; Python takes care of that for you in a more efficient and readable manner with comprehensions.

Multiple if-clauses

As we mentioned earlier, you can have multiple if-clauses in a comprehension along with multiple for-clauses. These are handled in essentially the same way as for-clauses: later clauses in the comprehension are nested inside earlier clauses.

For example, consider this comprehension:

```
values = [x / (x - y) for x in range(100) if x > 50 for y in range(100) if x - y != 0]
```

This is actually fairly difficult to read and might be at the limit of the utility for comprehensions, so let's improve the layout a bit:

```
values = [x / (x - y)
          for x in range(100)
          if x > 50
          for y in range(100)
          if x - y != 0]
```

That's much better. This calculates a simple statement involving two variables and two if-clauses. By interpreting the comprehension as a set of nested for-loops, the non-comprehension form of this statement is like this:

```
values = []
for x in range(100):
    if x > 50:
        for y in range(100):
            if x - y != 0:
                values.append(x / (x - y))
```

This can be extended to as many statements as you want in the comprehension, though, as you can see, you might need to spread your comprehension across multiple lines to keep it readable.

Looping variable binding

This last example also demonstrates an interesting property of comprehensions where later clauses can refer to variables bound in earlier clauses. In this case, the last if-statement refers to x which is bound in the first for-clause.

The for-clauses in a comprehension can refer to variables bound in earlier parts of the comprehension. Consider this example which constructs a sort of 'triangle' of coordinates in a flat list (We've re-formatted the REPL output for clarity):

```
>>> [(x, y) for x in range(10) for y in range(x)]
[(1, 0),
 (2, 0), (2, 1),
 (3, 0), (3, 1), (3, 2),
 (4, 0), (4, 1), (4, 2), (4, 3),
 (5, 0), (5, 1), (5, 2), (5, 3), (5, 4),
 (6, 0), (6, 1), (6, 2), (6, 3), (6, 4), (6, 5),
 (7, 0), (7, 1), (7, 2), (7, 3), (7, 4), (7, 5), (7, 6),
 (8, 0), (8, 1), (8, 2), (8, 3), (8, 4), (8, 5), (8, 6), (8, 7),
 (9, 0), (9, 1), (9, 2), (9, 3), (9, 4), (9, 5), (9, 6), (9, 7), (9, 8)]
```

Here the second for-clause which binds to the y variable refers to the x variable defined in the first for-clause. If this is confusing, remember that you can think of this a set of nested for-loops:

```
result = []
for x in range(10):
    for y in range(x):
        result.append((x, y))
```

In this formulation it's entirely natural for the inner for-loop to refer to the outer.

Nested comprehensions

There's one more form of nesting in comprehensions that's worth noting, although it doesn't really involve new syntax or anything beyond what we've already seen. We've been looking at the use of multiple for- and if-clauses in a comprehension, but it's also entirely possible to put comprehensions in the *output expression* for a comprehension. That is, each *element* of the collection produced by a comprehension can itself be a comprehension.

For example, here we have two for-clauses, but each belongs to a different comprehension entirely:

```
vals = [[y * 3 for y in range(x)] for x in range(10)]
```

The outer comprehension uses a second comprehension to create list for each entry in its result. Rather than a flat list, then, this produces a list-of-lists. The expansion of this comprehension looks like this:

```
outer = []
for x in range(10):
    inner = []
    for y in range(x):
        inner.append(y * 3)
    outer.append(inner)
```

And the resulting list-of-lists looks like this:

```
[[],
 [0],
 [0, 3],
 [0, 3, 6],
 [0, 3, 6, 9],
 [0, 3, 6, 9, 12],
 [0, 3, 6, 9, 12, 15],
 [0, 3, 6, 9, 12, 15, 18],
 [0, 3, 6, 9, 12, 15, 18, 21],
 [0, 3, 6, 9, 12, 15, 18, 21, 24]]
```

This is similar to, but different from, multi-sequence comprehensions. Both forms involve more than one iteration loop, but the structures they produce are very different. Which form you choose will, of course, depend on the kind of structure you need, so it's good to know how to use both forms.

Generator expressions, set comprehensions, and dictionary comprehensions

In our discussion of multi-input and nested comprehensions we've only shown list comprehensions in the examples. However, everything we've talked about applies equally to set comprehensions, dict comprehension, and generator expressions.

For example, you can use a set comprehension to create the set of all products of two numbers between 0 and 9 like this:

```
>>> {x * y for x in range(10) for y in range(10)}
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 24, 25, 27, 28,
 30, 32, 35, 36, 40, 42, 45, 48, 49, 54, 56, 63, 64, 72, 81}
```

Or you can create a generator version of the triangle coordinates we constructed earlier:

```
>>> g = ((x, y) for x in range(10) for y in range(x))
>>> type(g)
<class 'generator'>
>>> list(g)
[(1, 0),
 (2, 0), (2, 1),
 (3, 0), (3, 1), (3, 2),
 (4, 0), (4, 1), (4, 2), (4, 3),
 (5, 0), (5, 1), (5, 2), (5, 3), (5, 4),
 (6, 0), (6, 1), (6, 2), (6, 3), (6, 4), (6, 5),
 (7, 0), (7, 1), (7, 2), (7, 3), (7, 4), (7, 5), (7, 6),
 (8, 0), (8, 1), (8, 2), (8, 3), (8, 4), (8, 5), (8, 6), (8, 7),
 (9, 0), (9, 1), (9, 2), (9, 3), (9, 4), (9, 5), (9, 6), (9, 7), (9, 8)]
```

Functional-style tools

Python’s concept of iteration and iterable objects is fairly simple and abstract, not involving much more than the idea of a sequence of elements that can be accessed one at a time, in order. This high level of abstraction allows us to develop tools that work on iterables at an equally high-level, and Python provides you with a number of functions that serve as simple building blocks for combining and working with iterables in sophisticated ways.

> A lot of these ideas were originally developed in the functional programming community, so some people refer to the use of these techniques as “functional-style” Python. Whether you think of these as a separate programming paradigm or just as more tools in your programming arsenal, these functions can be very useful and are often the best way to express certain computations.

`map()`

The `map()` function if probably one of the most widely recognized functional programming tools in Python. At its core, `map()` does a very simple thing: given a callable object and a sequence of objects, it calls the function once for every element in the source series, producing a new series containing the return values of the function. In functional programming jargon, we “map” a function over a sequence to produce a new sequence.

Let's see a simple example. Suppose we wanted to find the Unicode codepoint for each character in a string. The `map()` expression for that would look like this:

```
map(ord, 'The quick brown fox')
```

This essentially says "for every element in the string, call the function `ord()` with that element as an argument. Generate a new sequence comprising the return values of `ord()`, in the same order as the input sequence.""

Graphically it looks like this:

```
T  ->  ord()  ->  84
h  ->  ord()  ->  104
e  ->  ord()  ->  101
   ->  ord()  ->  32
q  ->  ord()  ->  113
u  ->  ord()  ->  117
i  ->  ord()  ->  105
c  ->  ord()  ->  99
k  ->  ord()  ->  107
   ->  ord()  ->  32
b  ->  ord()  ->  98
r  ->  ord()  ->  114
o  ->  ord()  ->  111
w  ->  ord()  ->  119
n  ->  ord()  ->  110
   ->  ord()  ->  32
f  ->  ord()  ->  102
o  ->  ord()  ->  111
x  ->  ord()  ->  120
```

Let's try this out in the REPL:

```
>>> map(ord, 'The quick brown fox')
<map object at 0x1007cad10>
```

Rather than return a list as you might expect, we instead get a `map object`. The `map()` function, it turns out, performs lazy evaluation. That is, it doesn't produce any output until it's needed.

Another way to say this is that `map()` will not call its function or access any elements from its input sequence until they're actually needed. The `map` object returned by `map()` is itself an iterator object, and only by iterating over it can you start to produce output.

While most people (us included) typically refer to `map` as a function, it is in fact a class! So when you call `map()`, you're not calling a function *per se* but rather constructing an instance of `map`. You can see this at the REPL:

```
>>> map
<class 'map'>
>>> m = map(ord, 'turtles all the way down')
>>> isinstance(m, map)
True
```

Tracing `map` execution

To make this a bit more clear, let's reuse our `Trace` decorator from chapter 3 to print out a message whenever `map` calls its function. We won't be using `Trace` as a decorator, but rather we'll leverage the fact that we can call a `Trace` instance to get a callable that does tracing for us.

Here's how we use `Trace` to invoke the `ord` function as we `map` over a string:

```
>>> result = map(Trace()(ord), 'The quick brown fox')
>>> result
<map object at 0x10f22b7d0>
```

The function returned by `Trace.__call__()` will print a bit of text each time its called, so we can see how `map()` works.

It's not until we start to iterate over `result` that we see our function executed:

```
>>> next(result)
Calling <built-in function ord>
84
>>> next(result)
Calling <built-in function ord>
104
>>> next(result)
Calling <built-in function ord>
101
```

Again, this gets to the heart of lazy evaluation: `map()` does not call the function or iterate over the input sequence until it needs to in order to produce output. Here we're driving the process by manually iterating over the `map object` returned by `map()`.

More commonly, you will not manually drive the `map object`, but instead you'll iterate over it using some higher-level method. For example, you could use the `list()` constructor to read all of the elements produced by `map()`:

```
>>> list(map(ord, 'The quick brown fox'))
[84, 104, 101, 32, 113, 117, 105, 99, 107, 32, 98, 114, 111, 119, 110,
 32, 102, 111, 120]
```

Or you could use a for-loop:

```
>>> for o in map(ord, 'The quick brown fox'):
...     print(o)
...
84
104
101
32
113
117
105
99
107
32
98
114
111
119
```

```
110
32
102
111
120
```

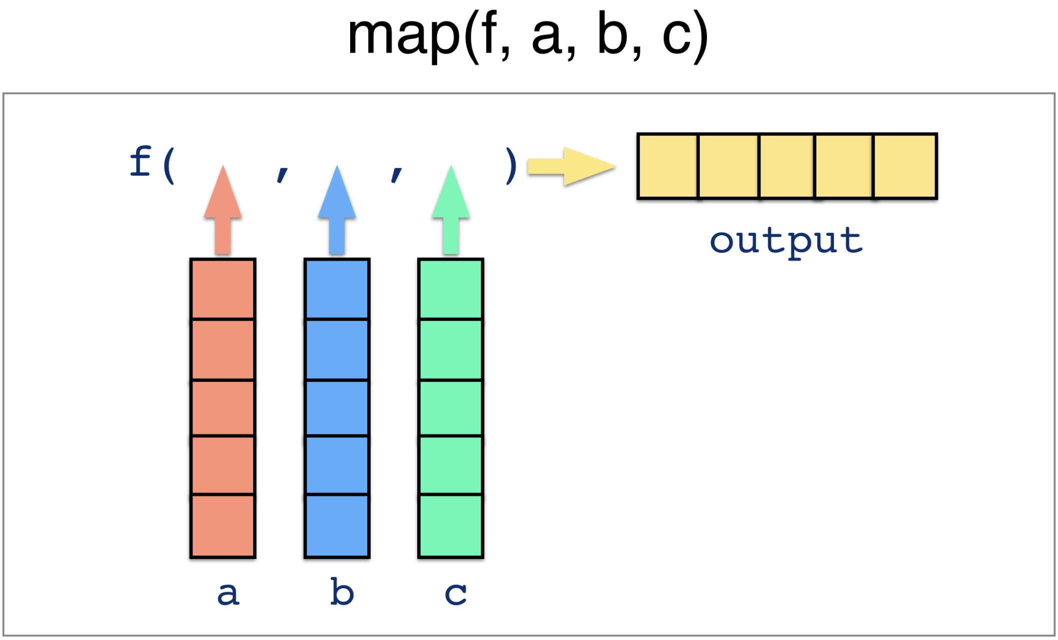
The point is that `map()`'s lazy evaluation requires you to iterate over its return value in order to actually produce the output sequence. Until you access the values in the sequence, they are not evaluated.

Multiple input sequences

So far we've seen examples using `map()` over single input sequences, but in fact `map()` can be used with as many input sequences as your mapped function needs.

If the function you pass to `map()` requires two arguments, then you need to provide two input sequences to `map()`; if the function requires three arguments, then you need to provide three input sequences. Generally, you need to provide as many input sequences as there are arguments in the mapped function.

When `map()` is given multiple input sequences, it takes an element from each sequence and passes it as the corresponding argument to the mapped function to produce each output value. In other words, for each output value that `map()` needs to produce, it takes the next element from each input sequence. It then passes these, in order, as the arguments to the mapped function, and the return value from the function is the next output value from `map()`.



Mapping multiple input sequence

An example will help make this clear. First we will define a three sequences:

```
>>> sizes = ['small', 'medium', 'large']
>>> colors = ['lavender', 'teal', 'burnt orange']
>>> animals = ['koala', 'platypus', 'salamander']
```

Then we will create a function that takes three arguments:

```
>>> def combine(size, color, animal):
...     return '{} {} {}'.format(size, color, animal)
```

Now if we map `combine()` over the input series, we see that an element from each series is passed to `combine()` for each output value:

```
>>> list(map(combine, sizes, colors, animals))
['small lavender koala', 'medium teal platypus', 'large burnt orange salamander']
```

Inputs of different sizes

It's possible, of course, that the input series are not all the same size. In fact, some of them might be infinite series. To account for this, `map()` will terminate as soon as any of the input sequences is exhausted. To see this, let's modify our `combine()` function to accept a quantity argument:

```
>>> def combine(quantity, size, color, animal):
...     return '{} x {} {} {}'.format(quantity, size, color, animal)
```

We can generate an infinite series of integers using the `count()` function from the standard library `itertools` module. Let's pass just such an infinite series to `map()` and see that it terminates after the finite inputs are exhausted:

```
>>> list(map(combine, itertools.count(), sizes, colors, animals))
['0 x small lavender koala',
 '1 x medium teal platypus',
 '2 x large burnt orange salamander']
```

`map()` versus comprehensions

You may have noticed that `map()` provides some of the same functionality as comprehensions. For example, both of these snippets produce the same lists:

```
>>> [str(i) for i in range(5)]
['0', '1', '2', '3', '4']
>>> list(map(str, range(5)))
['0', '1', '2', '3', '4']
```

Likewise, both this generator expression and this call to `map` produce equivalent sequences:

```
>>> i = (str(i) for i in range(5))
>>> list(i)
['0', '1', '2', '3', '4']
>>> i = map(str, range(5))
>>> list(i)
['0', '1', '2', '3', '4']
```

In cases where either approach will work, there's no clear choice which is better. Neither approach is necessarily faster than the other,[59] and, while many people find comprehensions more readable, others feel that the functional style is cleaner. Your choice will have to depend on your specific situation, your audience, or perhaps just your personal taste.

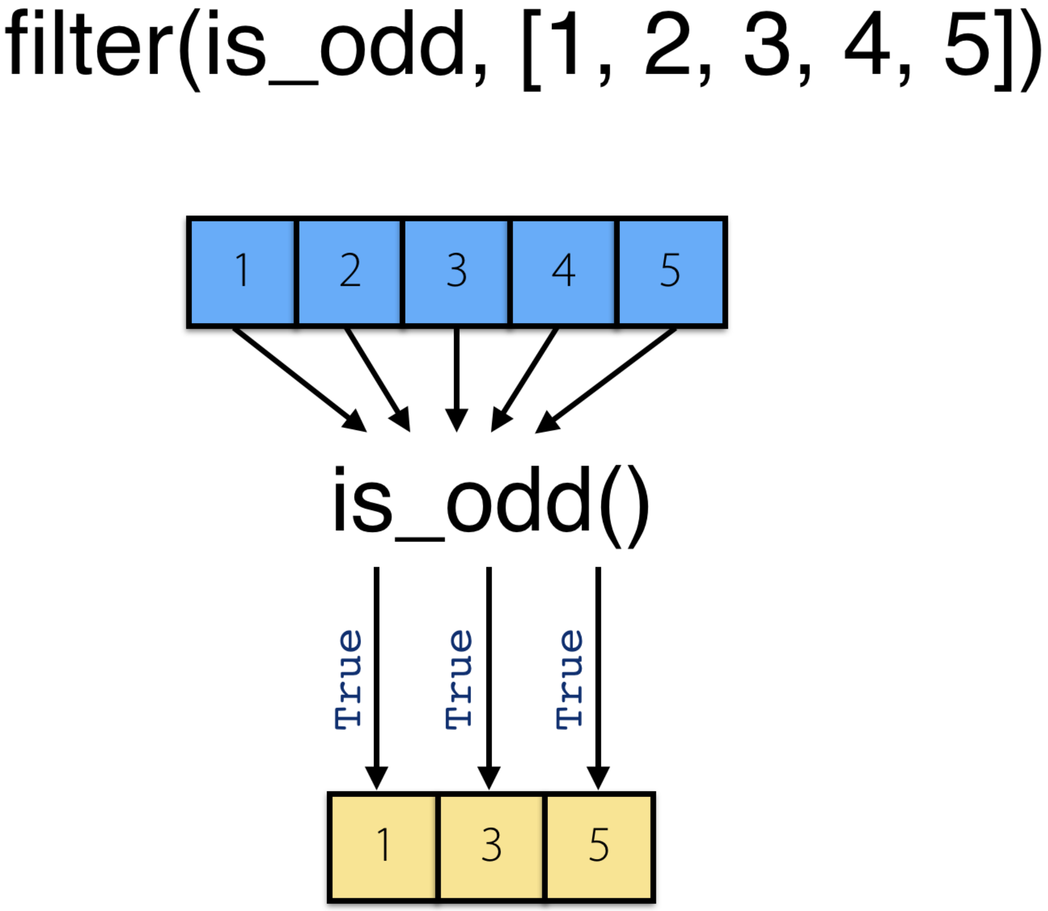
`filter()`

The next functional-style tool we'll look at is `filter()`. As its name implies, `filter()` looks at each element in an iterable series and skips those which don't meet some criteria. Like `map()`, `filter()` applies a callable to each element in a series, and also like `map()`, `filter()` produces its results lazily. Unlike `map()`, `filter()` only accepts a single input sequence, and the callable it takes must only accept a single argument.

The general form of `filter()` takes a callable of one argument as its first parameter and a iterable series as its second. It returns an iterable object of type `filter`. The `filter()` function[60] applies its first argument to each element in the input series, and returns a series containing only those elements of the input for which the function returns `True`.

[59] In some cases you may find significant differences in speed, though. If the performance of a piece of code is critical, it might be worth timing both versions.

[60] As with `map`, `filter` is a really a class.



Filter removes elements from a sequence

For example, here we use a lambda as the first argument to `filter`. This lambda returns `True` for positive arguments, and `False` for everything else, meaning that this call to `filter` will return a sequence containing only the positive elements of its input series:

```
>>> positives = filter(lambda x: x > 0, [1, -5, 0, 6, -2, 8])
>>> positives
<filter object at 0x101838490>
```

And indeed that is precisely what we see:

```
>>> list(positives)
[1, 6, 8]
```

Remember that the return value from `filter` is, being an iterable, lazy, so we have to use the list constructor or some other technique to force evaluation of the results.

You can optionally pass `None` as the first argument to `filter`, in which case `filter` will filter out all input elements which evaluate to `False` in a boolean context.

For example:

```
>>> trues = filter(None, [0, 1, False, True, [], [1, 2, 3], '', 'hello'])
>>> list(trues)
[1, True, [1, 2, 3], 'hello']
```

Since `0`, `False`, `[]` (an empty list), and `''` (an empty string) are all “falsey”, they were removed from the output sequence.

Differences between Python 2 and 3

While this book is focused on Python version 3, the `map` and `filter` functions represent one area where it’s useful to discuss a difference between Python 2 and 3. As we’ve just shown, in Python 3 the `map` and `filter` functions return an lazily-produced series of values. In Python 2, however, these functions actually return list sequences. If you find yourself writing code for Python 2, it’s important to keep this in mind

`functools.reduce()`

The final functional-style tool we’ll look at is the `reduce()` function in the `functools` standard library module. `reduce()` can be a bit confusing at first, but it’s actually quite simple and broadly useful.[61] `reduce()` repeatedly applies a function of two arguments to an interim *accumulator* value and the elements of the series (each in turn), updating — or accumulating — the interim value at each step with the result of the called function. The initial value of the accumulator can either be the first element of the sequence or an optional

[61] You can, in fact, implement both `map` and `filter` in terms of `reduce`

value that you supply. Ultimately the final value of the accumulator is returned, thereby *reducing* the series down to a single value.

`reduce()` is not unique to Python by any means, and you may have come across it in other languages you've worked with. For example, `reduce()` is the same as `fold` in many functional languages. Likewise, it's equivalent to `Aggregate()` in .NET's LINQ and `accumulate()` in the C++ Standard Template Library.

Summation with `reduce()`

The canonical example of `reduce()` is the summation of a sequence, and in fact `reduce()` is a generalization of summation. Here how that looks, using the standard library's `operator.add()` which is a regular function version of the familiar infix `+` operator:

```
>>> import operator
>>> reduce(operator.add, [1, 2, 3, 4, 5])
15
```

Conceptually what's happening is something like this:

```
numbers = [1, 2, 3, 4, 5]
accumulator = numbers[0]
for item in numbers[1:]:
    accumulator = operator.add(accumulator, item)
```

To get a better idea of how `reduce()` is calling the function, we can use a function which prints out its progress:

```
>>> def mul(x, y):
...     print('mul {} {}'.format(x, y))
...     return x * y
...
>>> reduce(mul, range(1, 10))
mul 1 2
mul 2 3
mul 6 4
mul 24 5
mul 120 6
mul 720 7
mul 5040 8
mul 40320 9
362880
```

Here we see that the interim result — the *accumulator* — is passed as the first argument to the reducing function, and the next value in the input sequence is passed as the second.

`reduce()` details

The `reduce()` function has a few edges that are worth noting. First, if you pass an empty sequence to `reduce()` it will raise a `TypeError`:

```
>>> reduce(mul, [])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: reduce() of empty sequence with no initial value
```

Second, if you pass a sequence with only one element, then that element is returned from `reduce()` without ever calling the reducing function:

```
>>> reduce(mul, [1])
1
```

`reduce()` accepts an optional argument specifying the initial value. This value is conceptually added to the beginning of the input sequence, meaning that it will be returned if the input sequence is empty. This also means that the optional initial value serves as the first accumulator value for the reduction.

This optional value is useful. For example, you can use it if you can't be sure if your input will have any values:

```
>>> values = [1, 2, 3]
>>> reduce(operator.add, values, 0)
6
>>> values = []
>>> reduce(operator.add, values, 0)
0
```

Take care when selecting initial values since the correct value, of course, depends on the function you are applying. For example, you use an initial value of `0` for summation:

```
>>> values = [1, 2, 3]
>>> reduce(operator.add, values, 0)
```

But you would use a value of `1` for products:

```
>>> values = [1, 2, 3]
>>> reduce(operator.mul, values, 1)
```

Combining functional concepts: map-reduce

You may have been wondering about the naming of these functions, and in particular you may have been wondering if there's any relationship between Python's `map` and `reduce` functions and the popular map-reduce algorithm.

The answer is "yes": Python's `map` and `reduce` are very much the same as those terms in map-reduce. To see this, let's write a small example that counts words across a set of documents.

First, we need a function which counts words in a document:

```
def count_words(doc):
    normalised_doc = ''.join(c.lower() if c.isalpha() else ' ' for c in doc)
    frequencies = {}
    for word in normalised_doc.split():
        frequencies[word] = frequencies.get(word, 0) + 1
    return frequencies
```

This produces a dictionary mapping words to the number of times that word was found in the input string. For example:

```
>>> count_words('It was the best of times, it was the worst of times.')
{'best': 1, 'it': 2, 'of': 2, 'the': 2, 'times': 2, 'was': 2, 'worst': 1}
```

With this function, we can now "map" it across a collection of documents. This will result in a sequence of dictionaries with word frequencies for the individual documents. Here's how that looks:

```
documents = [
    'It was the best of times, it was the worst of times.',
    'I went to the woods because I wished to live deliberately, to front only the ess\
ential facts of life...',
    'Friends, Romans, countrymen, lend me your ears; I come to bury Caesar, not to pr\
aise him.',
    'I do not like green eggs and ham. I do not like them, Sam-I-Am.',
]

counts = map(count_words, documents)
```

Next, we need a function which takes two word-count dictionaries and combines them:

```
def combine_counts(d1, d2):
    d = d1.copy()
    for word, count in d2.items():
        d[word] = d.get(word, 0) + count
    return d
```

With this we have all of the tools we need to run a proper map-reduce. Here's how that looks:

```
>>> total_counts = reduce(combine_counts, counts)
>>> total_counts
{'life': 1, 'sam': 1, 'friends': 1, 'eggs': 1, 'essential': 1, 'am': 1, 'and': 1, 'be\
cause': 1, 'do': 2, 'ham': 1, 'them': 1, 'romans': 1, 'lend': 1, 'praise': 1, 'worst'\
: 1, 'me': 1, 'caesar': 1, 'of': 3, 'was': 2, 'woods': 1, 'countrymen': 1, 'facts': 1\
, 'front': 1, 'not': 3, 'green': 1, 'him': 1, 'live': 1, 'your': 1, 'it': 2, 'come': \
1, 'went': 1, 'deliberately': 1, 'the': 4, 'bury': 1, 'ears': 1, 'like': 2, 'to': 5, \
'best': 1, 'i': 6, 'wished': 1, 'times': 2, 'only': 1}
```

It's interesting to see a somewhat "real-world" application of the relatively abstract notions we've presented so far in this module. And how knows, maybe someday someone will find a way to make a business using techniques like this!

The iteration protocols

In chapter 7 of “The Python Apprentice”[62] we looked at the basics of how iteration is implemented in Python. There are simple protocols for using iterables and iterators, and these protocols are driven by the use of the `iter()` and `next()` functions. First, `iter()` is called on an iterable object to retrieve an iterator. Then `next()` is called on the iterator to sequentially retrieve elements from the iterable. When no more elements are available, `next()` will raise `StopIteration`. It’s a very simple — but surprisingly powerful — protocol, and it’s pervasive in Python.

In this section we’ll look at how to *implement* iterable objects and iterators. You’ll see that it’s actually fairly simple and follows very much directly from the high-level `iter()` and `next()` API.

Iterables

An *iterable object* in Python is simply an object that supports the *iterable protocol* by implementing `__iter__()`. When you call `iter(obj)`, this ultimately results in a call to `obj.__iter__()`, and `iter()` returns the result of this call . `__iter__()` is required to return an *iterator*, a concept that we’ll discuss in a moment. There’s not much more to say about iterables right now; the protocol is very simple, and doesn’t make much sense on its own. To really see the point of iterables, we need to look at iterators.

Iterators

An iterator is simply an object that fulfills the *iterator protocol*. The first part of the iterator protocol is the iterable protocol that we just described. In other words, all *iterators* must also be *iterable*. This means that all iterators must implement `__iter__()`. Often iterators just return themselves from `__iter__()`, though that’s not always the case.

Iterators are also required to implement `__next__()`, the function called by `next()`. Like `iter()` and `__iter__()`, `next(obj)` results in a call to `obj.__next__()`, the result of which gets returned by `next()`. So to implement the iterator protocol, an object needs to implement `__next__()` as well as `__iter__()`. `__next__()` should return the next value in whatever sequence the iterator represents or, if the sequence is exhausted, raise the `StopIteration` exception.

To demonstrate the basic mechanics of developing an iterator, here’s a simple example:

[62] https://leanpub.com/python-apprentice

```
class ExampleIterator:
    def __init__(self):
        self.index = 0
        self.data = [1, 2, 3]

    def __iter__(self):
        return self

    def __next__(self):
        if self.index >= len(self.data):
            raise StopIteration()

        rslt = self.data[self.index]
        self.index += 1
        return rslt
```

This class is created with a list of three items. Each call to `__next__()` returns a new value in this list until it's exhausted, after which `StopIteration` is raised.

Here's how it looks when we drive it with the `next()` function:

```
>>> i = ExampleIterator()
>>> next(i)
1
>>> next(i)
2
>>> next(i)
3
>>> next(i)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "iterator_example.py", line 8, in __next__
    raise StopIteration()
StopIteration
```

Since iterators are also iterables, we can even use our iterator in a for-loop:

```
>>> for i in ExampleIterator():
...   print(i)
...
1
2
3
```

Putting `__next__` and `__iter__` together

Now that we know about both the iterable and iterator protocols, let's put them together to create a complete — if somewhat contrived — example. First, we'll modify `ExampleIterator` to take a sequence of values as an argument and iterate over it:

```
class ExampleIterator:
    def __init__(self, data):
        self.index = 0
        self.data = data

    def __iter__(self):
        return self

    def __next__(self):
        if self.index >= len(self.data):
            raise StopIteration()

        rslt = self.data[self.index]
        self.index += 1
        return rslt
```

Next, our `ExampleIterable` class will manage a collection of integers, returning an `ExampleIterator` object when asked for an iterator:

```
class ExampleIterable:
    def __init__(self):
        self.data = [1, 2, 3]

    def __iter__(self):
        return ExampleIterator(self.data)
```

Now we can see the iterable and iterator protocols working together in a for-loop:

```
>>> for i in ExampleIterable():
...     print(i)
...
1
2
3
```

as we can see them in a comprehension:

```
>>> [i * 3 for i in ExampleIterable()]
[3, 6, 9]
```

Excellent! These classes won't win any design awards, but they do demonstrate the fundamentals of how to construct your own iterators and iterables. It's all actually very simple in the end, and it requires that you keep in mind the simple protocols involved.

The alternative iterable protocol: `__getitem__()`

While most iterable objects are implemented using the protocol described above, there is another way to implement them. Rather than implementing the `__iter__` method, an object can implement the `__getitem__` method. For this to work, `__getitem__` must return values for consecutive integer indices starting at `0`. When the index argument is out of the iterables range of data, then `__getitem__` must raise `IndexError`. For many container types which already support `__getitem__`, this alternate protocol means that they don't need to write any special code to support iteration.

Here's a simple example of this protocol:

```
class AlternateIterable:
    def __init__(self):
        self.data = [1, 2, 3]

    def __getitem__(self, idx):
        return self.data[idx]
```

We see that `__getitem__` simply returns values from the specified index in an underlying list. Since the list will naturally work for consecutive indices starting at zero, and since it will raise `IndexError` when the index is out of bounds, this works for iteration.

Let's see this in action:

```
>>> [i for i in AlternateIterable()]
[1, 2, 3]
```

Extended `iter()` form

Finally, let's close out our discussion of iterators and iterables by looking at the so-called *extended form* of `iter()`. Normally `iter()` is called on objects that support the `__iter__` method of the iterable protocol. But `iter()` supports a two-argument form that lets you iterate over some objects which don't directly support the iterable protocol.

In this extended form, the first argument is a callable which takes zero arguments. The second argument is a *sentinel value*. The return value from `iter()` in this case is an iterator which produces values by repeatedly calling the callable argument. This iterator terminates when the value produced by the callable is equal to the sentinel.

Using extended `iter` with normal functions

This form of `iter` is not seen very often, and it may not be immediately clear why it's useful. But one relatively common use for it is to create infinite sequences out of normal functions.

In this example, we'll create an iterator over an infinite sequence of timestamps:

```
>>> import datetime
>>> i = iter(datetime.datetime.now, None)
>>> next(i)
datetime.datetime(2013, 11, 8, 13, 20, 37, 720158)
>>> next(i)
datetime.datetime(2013, 11, 8, 13, 20, 39, 296055)
>>> next(i)
datetime.datetime(2013, 11, 8, 13, 20, 40, 734203)
```

Here's how this works. `datetime.now()` takes zero arguments, and it returns a `datetime` object for the current time. Since it takes no arguments, it's suitable as the first argument to the extended `iter` form. For the sentinel argument to `iter` we've chosen `None`. Since `None` will never be returned by `datetime.now`, the iterator will never terminate, producing timestamps for as long as we choose to call it.

Reading files with extended `iter`

Another example of the extended `iter` form shows how it can be used to read from a file until a specific line is read. In this case, we read from a file until a line containing just the string "END" is read:

```
with open("ending_file.txt", 'rt') as f:
    for line in iter(lambda: f.readline().strip(), "END"):
        print(line)
```

If `ending_file.txt` contains this text:

```
You should
see this
text.
END
But not
this text.
```

then the output from our loop will be the first three lines from the file:

```
You should
see this
text.
```

Case study: an iterable and iterator for streamed sensor data

The concepts in this chapter are a bit high-level and perhaps a bit abstract, so let's take a look at a small example that shows how these techniques might be useful in the real world.

One situation where you might need to create an iterable object in Python is when reading data from a sensor. Often sensors produce a stream of data or can simply provide a value whenever queried, and it would be nice to be able to access these values in a loop. We're going to write a simple class which mimics a sensor and provides a stream of data.

Since we don't all have simple access to a physical sensor for this example, we're going to simulate the sensor data with random values within a range. Here's our sensor class:

```
import random

class Sensor:
    def __iter__(self):
        return self

    def __next__(self):
        return random.random()
```

That’s incredibly simple, but you can see that it meets the criteria for iteration. It implements `__iter__` which returns an iterator which, in this case, is the same object. Since the `Sensor` also implements `__next__`, it works equally well as an iterator. `__next__` simply returns a random number, but you could imagine more complex code that actually read real values from a sensor.

Let’s combine this with our timestamp generator and create a small system for logging sensor data every second for ten seconds:

```
import datetime
import itertools
import random
import time


class Sensor:
    def __iter__(self):
        return self

    def __next__(self):
        return random.random()

sensor = Sensor()
timestamps = iter(datetime.datetime.now, None)

for stamp, value in itertools.islice(zip(timestamps, sensor), 10):
    print(stamp, value)
    time.sleep(1)
```

This gives great results:

```
2013-11-08 13:37:43.439541 0.9099758630610068
2013-11-08 13:37:44.439454 0.9372711463526507
2013-11-08 13:37:45.440571 0.7290460647892831
2013-11-08 13:37:46.441032 0.9157167055401962
2013-11-08 13:37:47.442279 0.4931189807187356
2013-11-08 13:37:48.443690 0.6387447072526389
2013-11-08 13:37:49.444061 0.7902468804546714
2013-11-08 13:37:50.445296 0.00751052511934891
2013-11-08 13:37:51.446641 0.26767572449284105
2013-11-08 13:37:52.447714 0.07697202636210276
```

Viola! Your very own streaming, iterable sensor in Python! If you want to really take this to the next level you could modify `Sensor.__next__` to read, for example, your CPU's temperature sensor instead of returning a random value. How you do this depends on your system, but it would be a great exercise if you want to really make a working end-to-end sensor program in Python.

Summary

Hopefully this chapter has given you greater insight into Python's iteration system and given you greater confidence to develop with it. Let's review what we've covered:

- Comprehension
  - Comprehensions can process more than on input series
  - Multiple input series in comprehensions work like nested for-loops
  - Comprehensions can also have multiple if-clauses interspersed with the for-clauses
  - Later clauses in a comprehension can reference variables bound in earlier clauses
  - Comprehension can also appear in the result expression of a comprehension, resulting in nested result series
- Python provides a number of functional-style tools for working with iterators
- `map`
  - `map()` calls a function for each element in its input series
  - `map()` returns an iterable object, not a fully-evaluated collection
  - `map()` results are lazily evaluated, meaning that you must access them to force their calculation
  - `map()` results are typically evaluated through the use of iteration constructs such as for-loops

  - You must provide as many input series to `map()` as the callable argument has parameters
  - `map()` takes one element from each input series for each output element it produces
  - `map()` stops producing output when its shortest input series is exhausted
  - `map()` can be used to implement the same behavior as comprehensions in some cases
- `filter`
  - `filter()` selects values from an input series which match a specified criteria
  - `filter()` passes each element in its input series to the function argument
  - `filter()` returns an iterable over the input elements for which the function argument is 'truthy', skipping those for which the result is 'falsey'.
  - Like `map()`, `filter()` produces its output lazily
  - If you pass `None` as the first argument to `filter()`, it yields the input values which evaluate to `True` in a boolean context
- `functools.reduce`
  - `reduce()` cumulatively applies a function to the elements of an input series
  - `reduce()` calls the input function with two arguments: the accumulated result so far, and the next element in the series
  - `reduce()` is a generalization of summation
  - `reduce()` returns the accumulated result after all of the input has been processed
  - If you pass an empty series to `reduce()` it will raise a `TypeError`
  - `reduce()` accepts an optional initial value argument
  - This initial value is conceptually added to the front of the input series
  - The initial value is returned if the input series is empty
- The `map()` and `reduce()` functions in Python are related to the ideas in the map-reduce algorithm
- Iterables and iterators
  - Python's `next()` function calls `__next__()` on its argument
  - Iterators in Python must support the `__next__()` method
  - `__next__()` should return the next item in the series, or raise `StopIteration` if it is exhausted
  - Python's `iter()` function calls `__iter__()` on its argument
  - Iterable objects in Python must support the `__iter__()` method
  - `__iter__()` should return an iterator for the iterable object
  - Objects with a `__getitem__()` method that accepts consecutive integer indices starting at zero are also iterables

  - Iterables implemented via `__getitem__()` must raise `IndexError` when they are exhausted
  - The extended form of `iter()` accepts a zero-argument callable and a sentinel value
  - Extended `iter()` repeatedly calls the callable argument until it returns the sentinel value
  - The values produced by extended `iter()` are those returned from the callable
  - One use case for extended `iter()` is to iterate using simple functions
  - The value returned by extended `iter()` is both an iterable and an iterator
  - Protocol conforming iterators must also be iterable.

Chapter 8 - Inheritance and Subtype Polymorphism

In chapter 8 of "The Python Apprentice"[63] we covered the basics of inheritance in Python. That chapter was limited, however, to single-inheritance and the fundamental concepts of class hierarchies. In this chapter we'll take a deeper look at Python's support for inheritance — including multiple-inheritance — and the underlying mechanisms by which Python dispatches method calls. We'll also look at techniques for runtime type checking for those times when it's necessary.

Single inheritance

Before we look at multiple inheritance, let's do a quick review of single-inheritance. The syntax of single inheritance is part of the class declaration, with the base-class put in parentheses after the class name:

```
class SubClass(BaseClass):
    # ...
```

What this means is that `SubClass` will have all of the methods of `BaseClass`, and `SubClass` will be able to override these methods with new definitions if it wants to. In other words, `SubClass` can do everything that `BaseClass` can do, and it can optionally modify or specialize that behavior.

In general, a subclass initializer will want to call its base-class initializer to make sure that the full object is initialized. Remember, though, that if a subclass doesn't override the initializer, then the base-class initializer is called when an instance of the subclass is created.

Basic inheritance example

To see the basics of inheritance in action let's define a very simple base-class called `Base`:

[63] https://leanpub.com/python-apprentice

```
class Base:
    def __init__(self):
        print('Base initializer')

    def f(self):
        print('Base.f()')
```

If we create an instance of Base we see that its initializer is called:

```
>>> b = Base()
Base initializer
```

Similarly, calling b.f() invokes Base.f:

```
>>> b.f()
Base.f()
```

There should be nothing surprising here.

Let's now define a subclass of Base called Sub:

```
class Sub(Base):
    pass
```

This subclass doesn't add any functionality to Base at all.

Because we haven't defined an initializer for Sub, we can see that it inherits Base's initializer:

```
>>> s = Sub()
Base initializer
```

And likewise Sub also inherits Base.f():

```
>>> s.f()
Base.f()
```

Sub can override Base.f() by providing its own definition of the method:

```
class Sub(Base):
    def f(self):
        print('Sub.f()')
```

Now if we create an instance of `Sub` we see that the `Base` initializer is still being called, but `Sub`'s definition of `f` is now used:

```
>>> s = Sub()
Base initializer
>>> s.f()
Sub.f()
```

Subclass initializers

Finally, let's give `Sub` it's own initializer:

```
class Sub(Base):
    def __init__(self):
        print('Sub initializer')

    def f(self):
        print('Sub.f()')
```

Now when we create a new instance of `Sub`, we only see `Sub`'s initializer being called:

```
>>> s = Sub()
Sub initializer
```

If you're used to languages like C++ and Java, you might have expected Python to also call `Base`'s initializer when creating a `Sub` instance, but this isn't how Python behaves. Rather, Python treats the `__init__()` method like any other method, and it doesn't automatically call base-class initializers for subclasses that define initializers. If you define an initializer in a subclass and still want to call the initializer of a base-class — and this is often important to do — then you need to call it explicitly using the `super()` function.

Let's finish this small example by doing just that:

```
class Sub(Base):
    def __init__(self):
        super().__init__()
        print('Sub initializer')

    def f(self):
        print('Sub.f()')
```

Now when we construct an instance of `Sub`, we see both the `Sub` and `Base` initializers being called:

```
>>> s = Sub()
Base initializer
Sub initializer
```

The fact that Python doesn't provide special support for calling initializers is important to understand, and if it seems strange to you, don't worry: it soon becomes second-nature.

This small example should have been review for you. If you need to refresh your memory of basic inheritance in Python, you can refer to chapter 8 of “The Python Apprentice”[64] which covers it in some detail.

Realistic inheritance example: SortedList

As a more concrete and practical example of how to use inheritance in Python, we'll first define our own simplified list class called `SimpleList`:

```
class SimpleList:
    def __init__(self, items):
        self._items = list(items)

    def add(self, item):
        self._items.append(item)

    def __getitem__(self, index):
        return self._items[index]

    def sort(self):
```

[64] https://leanpub.com/python-apprentice

```
        self._items.sort()

    def __len__(self):
        return len(self._items)

    def __repr__(self):
        return "SimpleList({!r})".format(self._items)
```

`SimpleList` uses a standard list internally, and it provides a smaller, more limited API for interacting with the list data. We'll use `SimpleList` as the basis for the rest of our exploration of inheritance in Python.

A sorted subclass

Next let's create a subclass of `SimpleList` which keeps the list contents sorted. We'll call this class `SortedList` and it looks like this:

```
class SortedList(SimpleList):
    def __init__(self, items=()):
        super().__init__(items)
        self.sort()

    def add(self, item):
        super().add(item)
        self.sort()

    def __repr__(self):
        return "SortedList({!r})".format(list(self))
```

You can see that the `class` declaration includes the class name followed by `SimpleList` in parentheses, so `SimpleList` is the base-class of `SortedList`. The initializer for `SortedList` takes an optional argument which is an iterable series for initializing the list's contents. The initializer calls `SimpleList`'s initializer and then immediately uses `SimpleList.sort()` to sort the contents.

`SortedList` also overrides the `add` method on `SimpleList` to ensure that the list always remains sorted.

One aspect of this example that you'll notice are the calls to `super`. We'll be covering `super()` in more detail later in this chapter, but for now it can be understood to mean "call a method on my base-class.” So, for example, calling `super().add(x)` in `SortedList.add()` means to call `SimpleList.add(x)` with the same `self` argument, or, in other words, to use the base-class implementation of `add`.

If we go to our REPL, we can see that `SortedList` works as expected:

```
>>> sl = SortedList([4, 3, 78, 1])
>>> sl
SortedList([1, 3, 4, 78])
>>> len(sl)
4
>>> sl.add(-42)
>>> sl
SortedList([-42, 1, 3, 4, 78])
>>> sl.add(7)
>>> sl
SortedList([-42, 1, 3, 4, 7, 78])
```

Type inspection

Single inheritance in Python is relatively simple and should be conceptually familiar to anyone who’s worked with almost any other object oriented language. Multiple inheritance in Python is not much more complex and, as we’ll eventually see, both single and multiple inheritance ultimately rely on a single underlying model. Before we look at multiple inheritance, though, we need to lay some ground work for the examples we’ll be using.

`isinstance()`

Along with the `SortedList` class we defined earlier, we’re going to define another class, `IntList`, which only allows integers in the list. This list subclass will prevent the insertion of non-integer elements.

To do this, `IntList` will need to check the types of the items that are inserted, and the tool it will use for this is the built-in `isinstance()` function which takes an object as its first argument and a `type` as its second. It then determines if the object is of the specified `type`, returning `True` if it is and `False` otherwise.

For example, here we see `isinstance()` applied to a number of builtin types:

```
>>> isinstance(3, int)
True
>>> isinstance('hello!', str)
True
>>> isinstance(4.567, bytes)
False
```

Checking the inheritance graph

Instead of just checking for an exact type match, `isinstance()` will also return `True` if the object is a *subclass* of the second argument. For example, here we can see that a `SortedList` is an instance of `SortedList` as well as of `SimpleList`:

```
>>> sl = SortedList()
>>> isinstance(sl, SortedList)
True
>>> isinstance(sl, SimpleList)
True
```

Checking multiple types at once

A final twist to `isinstance()` is that it can accept a tuple of types for its second argument. This is equivalent to asking if the first argument is an instance of *any* of the types in the tuple. For example, this call returns `True` because `x` is an instance of `list`.

```
>>> x = []
>>> isinstance(x, (float, dict, list))
True
```

Implementing `IntList`

Now that we know how to use `isinstance()` we can define our `IntList` class like this:

```
class IntList(SimpleList):
    def __init__(self, items=()):
        for x in items:
            self._validate(x)
        super().__init__(items)

    @staticmethod
    def _validate(x):
        if not isinstance(x, int):
            raise TypeError('IntList only supports integer values.')

    def add(self, item):
        self._validate(item)
        super().add(item)

    def __repr__(self):
        return "IntList({!r})".format(list(self))
```

You'll immediately notice that `IntList` is structurally similar to `SortedList`. It provides its own initializer and, like `SortedList`, overrides the `add()` method to perform extra checks. In this case, `IntList` calls its `_validate()` method on every item that goes into the list. `_validate()` uses `isinstance()` to check the type of the candidates, and if a candidate is not an instance of `int`, `_validate()` raises a `TypeError`.

Let's see how that looks in the REPL:

```
>>> il = IntList([1, 2, 3 ,4])
>>> il.add(19)
>>> il
IntList([1, 2, 3, 4, 19])
>>> il.add('5')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "sorted_int_list.py", line 52, in add
    self._validate(item)
  File "sorted_int_list.py", line 49, in _validate
    raise TypeError('IntList only supports integer values.')
TypeError: IntList only supports integer values.
```

issubclass()

We've seen how `isinstance()` can be used to do type checks in Python. Checks like that are uncommon in Python, and while some people consider them to be a sign of poor design, sometimes they're simply the easiest way to solve a problem.

There is another built-in function related to `isinstance()` which can be used for type checking. This function, `issubclass()`, operates on types only, rather than operating on instances of types. As its name implies, `issubclass()` is used to determine if one class is a subclass of another. `issubclass()` takes two arguments, both of which need to be type objects, and it returns `True` if the first argument is a direct or indirect subclass of the second.

Here we can see that both `IntList` and `SortedList` are subclasses of `SimpleList`, as you would expect:

```
>>> issubclass(IntList, SimpleList)
True
>>> issubclass(SortedList, SimpleList)
True
```

On the other hand, `SortedList` is not a subclass of `IntList`:

```
>>> issubclass(SortedList, IntList)
False
```

Checking the inheritance graph

We can also use a simple example to verify that `issubclass` looks at the entire inheritance graph, not just direct parents:

```
>>> class MyInt(int): pass
...
>>> class MyVerySpecialInt(MyInt): pass
...
>>> issubclass(MyVerySpecialInt, int)
True
```

These classes are obviously pretty silly, but they do illustrate that `issubclass` recognizes that `MyVerySpecialInt` is indeed a subclass of `int` even though it only directly inherits from `MyInt`.

Multiple inheritance

Now that we've defined two interesting subclasses of `SimpleList` we're ready to look at multiple inheritance in Python. Multiple inheritance simply means defining classes with more than one direct base-class. This feature is not universal among object oriented languages. C++ supports multiple inheritance, for example, while Java does not. Multiple inheritance can lead to certain complex situations — for example, deciding what to do when more than one base class defines a particular method — but as we'll see Python has a relatively simple and understandable system for handling such cases.

The syntax for defining a class with multiple inheritance is very similar to single inheritance. To declare multiple base-classes, simply use a comma separated list of classes in the parentheses after the class name:

```
class SubClass(Base1, Base2, Base3):
    # ...
```

A class can have as many base classes as you want.

Just like single inheritance, a class with multiple base-classes supports all of the methods of its bases. As long there is no overlap in the method names of the base classes, it's always easy to see how method calls are resolved: find the base class with the matching method name, and that's which method gets called. If there is method name duplication across base classes, Python has a well-defined "method resolution order" for determining which is used. We'll look at method resolution order in more detail shortly.

Defining `SortedIntList`

Let's jump right in and define our own class using multiple inheritance. So far in this module we've defined `SortedList` and `IntList`, both of which inherit from our `SimpleList`. Now we're going to define a class which inherits from both of these classes and, thus, has the properties of both. Here's our `SortedIntList`:

```
class SortedIntList(IntList, SortedList):
    def __repr__(self):
        return "SortedIntList({!r})".format(list(self))
```

It doesn't look like much, does it? We've simply defined a new class and given it two base-classes. In fact, the only new implementation in the class is there to give it a proper string representation[65].

If we go to the REPL we can see that it works as we expect. The initializer sorts the input sequence:

```
>>> sil = SortedIntList([42, 23, 2])
>>> sil
SortedIntList([2, 23, 42])
```

However, it rejects non-integer values:

```
>>> SortedIntList([3, 2, '1'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "sorted_int_list.py", line 43, in __init__
    for x in items: self._validate(x)
  File "sorted_int_list.py", line 49, in _validate
    raise TypeError('IntList only supports integer values.')
TypeError: IntList only supports integer values.
```

Likewise, the `add` method maintains both the sorting and type constraints defined by the base-classes:

```
>>> sil.add(-1234)
>>> sil
SortedIntList([-1234, 2, 23, 42])
>>> sil.add('the smallest uninteresting number')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "sorted_int_list.py", line 45, in add
    self._validate(item)
  File "sorted_int_list.py", line 42, in _validate
    raise TypeError('IntList only supports integer values.')
TypeError: IntList only supports integer values.
```

[65] Using techniques of *introspection* we introduce in chapter 12, the methods can figure out the name of the class of the object on which they have been invoked. This means that even the `__repr__()` override can be removed, leaving us with an empty class block, save for a syntactically necessary `pass` statement.

You should spend some time playing with `SortedIntList` to convince yourself that it works as expected.

It may not be immediately apparent how all of this works, though. After all, both `IntList` and `SortedList` define `add()`. How does Python know which `add()` to call? More importantly, since both the sorting and type constraints are being enforced by `SortedIntList`, how does Python seem to know to call *both* of them? The answers to these questions have to do with the method resolution order we mentioned earlier, along with the details of how `super()` really works. We'll get to all of that very soon.

Automatic initializer calls

Before that, there are a few more details about multiple inheritance that we need to cover. First, if a class uses multiple inheritance but defines no initializer, only the initializer of the first base-class is automatically called when an instance of the class is created.

Consider this simple example:

```
class Base1:
    def __init__(self):
        print('Base1.__init__')

class Base2:
    def __init__(self):
        print('Base2.__init__')

class Sub(Base1, Base2):
    pass
```

If we now create an instance of `Sub` we see that only the initializer for `Base1` is called:

```
>>> s = Sub()
Base1.__init__
```

Through the use of `super()` we could design these classes such that both the `Base1` and `Base2` initializers are called automatically, and we'll see how to do that soon.

`type.__bases__`

Another useful thing to know when thinking about inheritance is the `__bases__` member of class objects. `__bases__` is simply a tuple of a class's base-classes:

```
SortedIntList.__bases__
(<class '__main__.IntList'>, <class '__main__.SortedList'>)
```

`SortedIntList` inherits from both `IntList` and `SortedList`, and those show up in the `__bases__` member of the `SortedIntList` class object. The entries in `__bases__` are in the same order as they were declared in the class definition.

You don’t have to use multiple inheritance to populate `__bases__`, as you can see if you look at `__bases__` for our `IntList` class:

```
>>> IntList.__bases__
(<class '__main__.SimpleList'>,)
```

MRO: Method Resolution Order

We’re finally ready to talk about this notion of “method resolution order” that we’ve mentioned several time now. In Python, the method resolution order — or simply MRO — of a class is the ordering of a class’s inheritance graph used to determine which implementation to use when a method is invoked. When you invoke a method on an object which has one or more base-classes, the actual code that gets run may be defined on:

- the class itself
- one of its direct base-classes
- a base-class of a base-class
- any other member of the class’s inheritance graph.

The MRO of a class determines the *order* in which the inheritance graph is searched to find the correct implementation of the method.

That’s a lot take in, but MRO is actually very simple. We’ll look at some examples that will make things more clear. First, though, we need to look at where a class’s MRO is stored.

The `__mro__` attribute

The method resolution order for a class is stored on a special member named `__mro__`. Here you can see the MRO for `SortedIntList`:

```
>>> SortedIntList.__mro__
(<class '__main__.SortedIntList'>,
 <class '__main__.IntList'>,
 <class '__main__.SortedList'>,
 <class '__main__.SimpleList'>,
 <class 'object'>)
```

The `__mro__` attribute is a tuple of classes defining the method resolution order.[66]

How is MRO used?

How is the MRO used to dispatch method calls? The answer is that when you call a method on an object in Python, Python looks at the MRO for that object's type. For each entry in the MRO, starting at the front and working in order to the back, Python checks if that class has the requested method. As soon as Python finds a class with a matching method, it uses that method and the search stops.

Let's see a simple example. First we'll define a few classes with a diamond inheritance graph:

```
class A:
    def func(self):
        return 'A.func'

class B(A):
    def func(self):
        return 'B.func'

class C(A):
    def func(self):
        return 'C.func'

class D(B, C):
    pass
```

Here the various `func` methods simply report which class they come from.

If we look at `D`'s MRO, we see that Python will check `D` first, then `B`, then `C`, followed by `A`, and finally `object` when resolving calls to objects of type `D`:

[66] You may also discover a method called `mro()`. This is called once, when a class is first created, to populate the `__mro__` attribute. You shouldn't call it yourself, preferring the `__mro__` attribute. We cover this further in *The Python Master* when we discuss *metaclasses*.

```
>>> D.__mro__
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>,
 <class '__main__.A'>, <class 'object'>)
```

> By the way, `object` is the ultimate base-class of every class in Python, and we'll discuss that more later in this chapter.

Based on that MRO, what should we expect if we call `func()` on an instance of `D`?:

```
>>> d = D()
>>> d.func()
'B.func'
```

Because `B` was the first class in `D`'s MRO with the method `func()`, Python called `B.func()`. If `C` had been earlier in the MRO than `B`, then `C.func()` would have been called.

We can see this by changing the order of `B` and `C` in the definition of `D`:

```
class D(C, B):
    pass
```

After this change, the new MRO for `D` puts `C` before `B`:

```
>>> D.__mro__
(<class '__main__.D'>, <class '__main__.C'>, <class '__main__.B'>,
 <class '__main__.A'>, <class 'object'>)
```

And indeed calling `func()` on a new instance of `D` results in a call to `C`'s implementation:

```
>>> d = D()
>>> d.func()
'C.func'
```

That's all that there really is to it. MRO is an ordering of a class's inheritance graph that Python calculates for you. When Python resolves a method call, it simply walks along that ordering until a class supports the requested method.

The MRO for `SortedIntList`

Let's see the MRO for our `SortedIntList` class:

```
>>> SortedIntList.__mro__
(<class '__main__.SortedIntList'>,
 <class '__main__.IntList'>,
 <class '__main__.SortedList'>,
 <class '__main__.SimpleList'>,
 <class 'object'>)
```

As you might have expected, the MRO is `SortedIntList` followed by `IntList`, followed by `SortedList`, with `SimpleList` and `object` bringing up the rear. So calls to `add` on a `SortedIntList`, for example, will resolve to `IntList.add` since `IntList` is the first class in the MRO with an `add` method.

This raises a very interesting question, however. When we wrote `IntList`, it never had any connection to the `SortedList` class. Yet, our `SortedIntList`, as we've seen, is properly maintaining both the sorting constraint and the type constraint of both `SortedList` and `IntList`. If `add` resolves to `IntList.add`, and if `IntList` is using `super()` to call its base-class implementation, how is `SortedList` being invoked to maintain the sorting? The answer to *that* mystery has to do with how `super()` actually works.

How is MRO calculated?

Before we move on to looking at `super()` in detail, you might be curious about how Python actually calculates the MRO for a class. The short answer is that Python uses an algorithm known as C3 linearization[67] for determining MRO. We won't go into the details of C3, except to mention a few important qualities of the MRO that it produces:

1. A C3 MRO ensures that subclasses come before their base-classes.
2. C3 ensures that the base-class order as defined in a class definition is also preserved.
3. C3 preserves the first two qualities independent of where in an inheritance graph you calculate the MRO. In other words, the MROs for all classes in a graph agree with respect to relative class order.

[67] https://en.wikipedia.org/wiki/C3_linearization

Inconsistent MROs

One outcome of the C3 algorithm is that not all inheritance declarations are allowed in Python. That is, some base-class declarations will violate C3 and Python will refuse to compile them.

Consider this simple example in the REPL:

```
>>> class A: pass
...
>>> class B(A): pass
...
>>> class C(A): pass
...
>>> class D(B, A, C): pass
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Cannot create a consistent method resolution
order (MRO) for bases A, C
```

Here, since B and C both inherit from A, B and C must both come before A in any MRO. This follows from the first quality that C3 preserves. However, since D's base-class declaration puts A before C, and since C3 also guarantees that base-declaration order is preserved, C3 can not produce a consistent MRO. That is, it can't put A both before and after C.

Understanding C3 is not critical — or really even necessary — for using Python, but it is an interesting topic for those curious about language design. If you want to learn more about it, you can find plenty of information on the web.

`super()`

Finally we have enough background to understand `super()`. So far we've seen `super()` used to access methods on base-classes, for example in `SortedList.add()` where we used `super()` to call `SimpleList.add()` before sorting the contents:

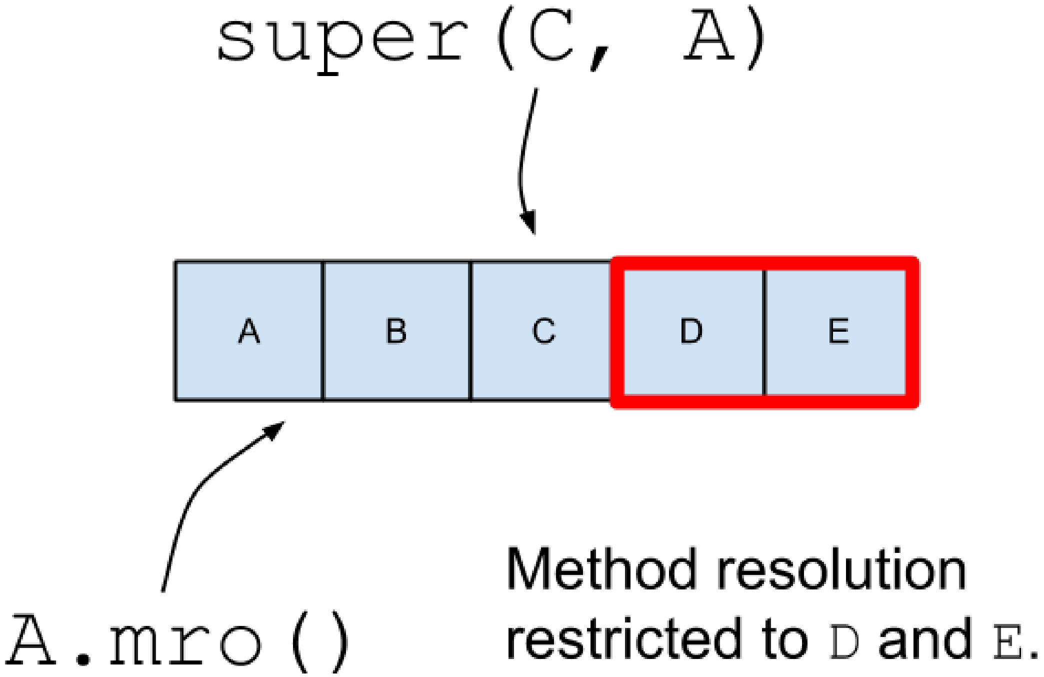
```
class SortedList(SimpleList):
  def add(self, item):
      super().add(item)
      self.sort()
```

From this example you might conclude that `super()` somehow returns the base-class of a method's class, and that you can then invoke methods on the "base-class part" of an object. This is only partly true.

It's hard to sum up what `super()` does in a single sentence, but here's an attempt:

> Given a method resolution order and a class C in that MRO, `super()` gives you an object which resolves methods using only the part of the MRO which comes *after* C.

In other words, `super()` doesn't work with the base-classes of a class, but instead it works with the MRO of the type of the object on which a method was originally invoked. The distinction is subtle but very important, and as we'll see it resolves the mystery of how `SortedIntList` works.



How super limits method resolution

Bound and unbound proxies

First let's look at the details of calling `super()`, which can be called in several forms. All of them return a so-called *super proxy* object. You can call any method on a super proxy, and it will route that call to the correct method implementation if one exists.

There are two high-level types of super proxies, *bound* and *unbound.* Bound proxies, as the name suggests, are bound to instances or class objects. On the other hand, unbound proxies aren't connected to any instance, so don't do any method dispatch themselves. Unbound proxies are primarily an implementation detail for other Python features. Some prominent Python developers consider them a bit of a wart on the language, and really they are beyond the scope of this book, so we won't discuss them in any more detail.

Bound proxies, on the other hand, are an important part of mastering Python, so we'll discuss them in detail. From now on, when we mention *proxies* or *super proxies*, understand that we mean *bound* proxies.

class-bound proxies

How do we use `super()` to create proxies? As we mentioned previously, bound proxies can be bound to either classes or instances of classes. We will call these *class-bound* and *instance-bound* proxies, respectively.

To create a class-bound proxy, you use this form:

```
super(base-class, derived-class)
```

Here, both arguments are class objects. The second class must be a subclass of (or the same class as) the first argument.

When you invoke a method on the proxy, here's what happens:

1. Python finds the MRO for `derived-class`
2. It then finds `base-class` in that MRO
3. It takes everything *after* `base-class` in the MRO and finds the first class in that sequence with a method name matching the request.

Class-bound proxies and `SortedIntList`

Let's see that in action with our `SortedIntList`. First, let's look at the MRO for `SortedIntList`:

```
>>> SortedIntList.__mro__
(<class '__main__.SortedIntList'>,
 <class '__main__.IntList'>,
 <class '__main__.SortedList'>,
 <class '__main__.SimpleList'>,
 <class 'object'>)
```

It's `SortedIntList`, `IntList`, `SortedList`, `SimpleList`, and `object`. Let's see what do we get if we call `super()` with the arguments `SortedList` and `SortedIntList`:

```
>>> super(SortedList, SortedIntList)
<super: <class 'SortedList'>, <SortedIntList object>>
```

That gives us a proxy bound to the arguments we'd expect. But what if we use that proxy to resolve a method like `add()`?

Applying the algorithm described above, Python first finds the MRO for the second argument, `SortedIntList`. It then takes everything in the MRO after the first argument, `SortedList`, giving us an MRO containing just `SimpleList` and `object`. It then finds the first class in that MRO with an `add()` method, which of course is `SimpleList`.

Let's see if we're right:

```
>>> super(SortedList, SortedIntList).add
<function SimpleList.add at 0x10f1c7b00>
```

There you have it! `super()` returned a proxy which, when asked for an `add()` method, returned `add()` from the `SimpleList` class.

Class-bound proxies and methods

Now that we have a method, we should be able to call it, right?:

```
>>> super(SortedList, SortedIntList).add(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: add() missing 1 required positional argument: 'item'
```

This failed because our proxy is bound to a class, not an instance, so we can't invoke it. If we used the proxy to look up a staticmethod or classmethod, however, we could invoke it directly.

To illustrate this, let's use a class-bound super proxy to call the `_validate()` staticmethod on `IntList`:

```
>>> super(SortedIntList, SortedIntList)._validate(5)
>>> super(SortedIntList, SortedIntList)._validate('hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "sorted_int_list.py", line 49, in _validate
    raise TypeError('IntList only supports integer values.')
TypeError: IntList only supports integer values.
```

This is probably not the kind of code you'd write in practice, but it does show how `super()` works when bound to class objects.

> Note that Python will raise an exception if the second argument is not a subclass of the first:
>
> ```
> >>> super(int, IntList)
> Traceback (most recent call last):
>   File "<stdin>", line 1, in <module>
> TypeError: super(type, obj): obj must be an instance or subtype of type
> ```

Instance-bound proxies

Instance-bound super proxies work very much like class-bound proxies, but instead of binding to a class object they bind to an instance. To create an instance-bound proxy, call `super()` like this:

```
super(class, instance-of-class)
```

Here, the first argument must be a class object, and the second argument must be an instance of that class or any class derived from it.

The behavior of the super proxy in this case is like this:

1. Python finds the MRO for the type of the second argument.
2. Then Python finds the location of the first argument to `super()` in that MRO; remember that the instance must be derived from the class, so the class must be in the MRO.

3. Finally, Python takes everything in the MRO *after* the class and uses that as the MRO for resolving methods.

Importantly, since the proxy is bound to an instance, we can call the methods after they've been found. Let's try that with our `SortedIntList` example:

```
>>> from sorted_list import *
>>> SortedIntList.__mro__
(<class 'sorted_list.SortedIntList'>,
 <class 'sorted_list.IntList'>,
 <class 'sorted_list.SortedList'>,
 <class 'sorted_list.SimpleList'>,
 <class 'object'>)
>>>
>>> sil = SortedIntList([5, 15, 10])
>>> sil
SortedIntList([5, 10, 15])
>>>
>>> super(SortedList, sil)
<super: <class 'SortedList'>, <SortedIntList object>>
>>>
```

The proxy is bound to a `SortedIntList` and will start method resolution from `SortedIntList`'s MRO at the entry *after* `SortedList`. As you'll recall, the entry in `SortedIntList`'s MRO after `SortedList` is `SimpleList`, so this super proxy will be directly using `SimpleList` methods. Interestingly, this bypasses the constraints imposed by `SortedList` and `IntList`!:

```
>>> super(SortedList, sil).add(6)
>>> sil
SortedIntList([3, 4, 5, 6])
>>>
>>> super(SortedList, sil).add('I am not a number! I am a free man!')
>>> sil
SortedIntList([3, 4, 5, 6, 'I am not a number! I am a free man!'])
```

Not only is our `SortedIntList` not sorted any more, but it also contains references to classic British TV[68]. Clearly you need to use `super()` with care or you can end up breaking your designs.

[68] https://en.wikipedia.org/wiki/Number_Six_(The_Prisoner)

The no-argument versions

We now know how `super()` works for creating class- and instance-bound proxy objects, and we've seen both in action. The examples we've just shown all pass arguments to `super()`, while the examples we saw earlier in this chapter — for example, in the implementation of `SortedIntList` — didn't use any arguments at all.

It turns out that you can call `super()` in a method with no arguments, and Python will sort out the arguments for you.

If you're in an instance method (that is, a method which takes an instance as its first argument) and you call `super()` without arguments, that's the same as calling `super()` with the method's class as the first argument and `self` as the second. In the simple case of single inheritance, then, this is equivalent to looking for a method on the base-class.

If you call `super()` without arguments in a class method, Python sets the arguments for you so that it's equivalent to calling `super()` with the method's class as the first argument and the classmethods first argument (that is, the "class" argument) as the second. Again, in the typical case, this is equivalent to calling the base-class's method.

More generally, in both cases — for instance methods and class methods — calling `super()` with no arguments puts the method's class as the first argument to `super()` and the first argument to the method itself as the second.

Resolving the `SortedIntList` mystery

So let's do a quick review of what `super()` does. Given a class and an MRO that contains that class, `super()` takes everything in the MRO *after* the class and uses that as a new MRO for resolving methods. This is all bundled up into proxy objects which are returned from the `super()` call.

Given that, let's see if we can resolve the apparent mystery of how `SortedIntList` works. Remember that `SortedList` and `IntList` were developed without referencing one another, yet when they're combined in a subclass, both of their constraints are still properly maintained.

The key to how this works is that both `SortedList` and `IntList` use `super()` to defer to their base-class. But as we now know, `super()` doesn't just let us access base-classes, but rather it lets us access the complete method resolution order for a class.

So, when `SortedList` and `IntList` are both used as bases for `SortedIntList`, the MRO for `SortedIntList` contains both of them. A call to `add` on a `SortedIntList` resolves to a call to `IntList.add()` which itself calls `super()`. The `super()` call in `IntList.add()` uses the full MRO for `SortedIntList`, meaning that rather than resolving to `SimpleList.add()` as we initially expected, it actually resolves to `SortedList.add()`. This is how `SortedIntList` maintains two constraints without having to manually combine them.

This is a fairly deep result, and if you understand how `SortedIntList` works, then you have a good grasp on `super()` and method resolution order in Python. If you're still unsure about these concepts, review this chapter and experiment on your own until you do.

`object`

Finally, let's finish this module by looking at the core of Python's object model, the class called — fittingly enough — `object`. You saw object earlier in this module when we looked at the MROs for various classes.

For example, it show up in the MRO for `IntList`:

```
>>> IntList.__mro__
(<class '__main__.IntList'>,
 <class 'SimpleList'>,
 <class 'object'>)
```

It also appears in the MRO for `SortedIntList`:

```
>>> SortedIntList.__mro__
(<class '__main__.SortedIntList'>,
 <class '__main__.IntList'>,
 <class '__main__.SortedList'>,
 <class '__main__.SimpleList'>,
 <class 'object'>)
```

Indeed, it's also in the MRO for more "primitive" things like `list` and `int`:

```
>>> list.__mro__
(<class 'list'>, <class 'object'>)
>>> int.__mro__
(<class 'int'>, <class 'object'>)
```

The fact is, `object` is the ultimate base-class for every class in Python. At the root of every inheritance graph you'll find `object`, and that's why it shows up in every MRO.

If you define a class with no base-class, you actually get `object` as the base automatically.

You can see this by looking at the `__bases__` member of a simple example class like this:

```
>>> class NoBaseClass: pass
...
>>> NoBaseClass.__bases__
(<class 'object'>,)
```

What does `object` actually do? We won't get into the details of everything it does (or of how it does it) in this book,[69] but it's good to have some idea of the role that `object` plays.

First, let's see what attributes `object` has:

```
>>> dir(object)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
 '__init__', '__le__', '__lt__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__']
```

All of these are special attributes or functions, so clearly `object` is doing significant work to tie into Python's implementation details. For example, the `__eq__`, `__ge__`, `__gt__`, `__le__`, `__lt__`, and `__ne__` methods are all hooks into Python's comparison operators. The `__str__` and `__repr__` methods, of course, tie into the `str()` and `repr()` functions, and `object` provides the default implementations of those methods.

More importantly, `object` provides the mechanisms for basic attribute lookup and management. Methods like `__getattribute__`, `__setattr__`, and `__delattr__` form the foundation on which Python objects expose attribute to callers.

[69] We cover some more aspects of `object` in "The Python Master".

For the most part you don't need to worry about `object` or understand how it works. But it is useful to know that it exists and that it's part of every class in Python. As you become an expert Python user you may eventually find that you need to know the details of `object`, so keep it in the back of your mind as you progress.

Inheritance for Implementation Sharing

One of the aspects of Python that differentiates it from nominally typed languages like C++ and Java is that, for the most part, the specific type name of an object does not determine if it can be used in a given context. Rather, Python uses duck-typing where an object's fitness for a use is only determined at the time it's actually used, and exceptions are raised when an object doesn't have the necessary attributes to fulfill a request.

Functions are defined without specifying type names on their arguments, and you can pass objects of any type to any function. Likewise, you can try to call any method you want on any object; Python won't complain until runtime.

One important result of this dynamic type system has to do with how inheritance is used in Python versus how it's used in nominally typed languages. With static nominal typing, such as in Java, if you want to pass an object to a function you need to make sure that the object is of the type expected by the function.

As a result, you very often end up creating specific interfaces and base-classes for specific uses, and the need to satisfy the type system becomes a significant — and sometimes the most significant — element in your development process.

In Python, since there is no *static* type system to satisfy, inheritance isn't generally needed for the purposes of bestowing an object with a particular type. Rather, inheritance in Python is best used as a way to share implementation. That is, inheritance in Python is a convenient way to reuse code much more than it is a way to construct type hierarchies.

Summary

Inheritance, method resolution, and the proper use of `super()` can be confusing at first, so hopefully this chapter showed you that there's really a clear, consistent, and ultimately quite simple system underlying Python's inheritance and object model.

Here are the topics we covered:

- Inheritance

  - Specify single inheritance by putting a base-class in parentheses after defining a class's name
  - Subclasses have all of the methods of their base-class
  - It's often best to explicitly call a base-class initializer from a subclass's initializer
  - If a class with a single base-class doesn't define an initializer, the base-class's initializer will be called automatically on construction
- Type introspection
  - `isinstance()` takes an object as its first argument and a type as its second
  - `isinstance()` determines if its first argument is an instance of the second argument, or any subclass of the second argument
  - `isinstance()` can accept a tuple of types as its second argument, in which it returns true if the first argument is of any of those types
  - Checking for specific types is rare in Python and is sometimes regarded as bad design
  - `isinstance()` determines if its first argument is a direct or indirect subclass of, or the same type as, the second argument
- Multiple inheritance
  - Multiple inheritance means having more than one direct base-class
  - You declare multiple base-classes with a comma-separated list of class names in parentheses after a class's name in a class definition
  - A class can have as many base-classes as you want
  - Python uses a well-defined "method resolution order" to resolve methods at runtime
  - If a multiply-inheriting class defines no initializer, Python will automatically call the initializer of its first base-class on construction
  - `__bases__` is a tuple of types on a class object which defines the base-classes for the class
  - `__bases__` is in the same order as in the class definition
  - `__bases__` is populated for both single and multiple inheritance
- Method resolution order
  - Method resolution order defines the order in which Python will search an inheritance graph for methods
  - MRO is short for Method Resolution Order
  - MRO is stored as a tuple of types in the `__mro__` attribute of a class
  - To resolve a method, Python uses the first entry in a class's MRO which has the requested method
  - MRO is dependent on base-class declaration order
  - MRO is calculated by Python using the C3 algorithm

  - MRO honors base-class ordering from class definitions
  - MRO puts subclasses before base-classes
  - The relative order of classes in an MRO is consistent across all classes
  - It is possible to specify an inconsistent base-class ordering, in which case Python will raise a `TypeError` when the class definition is reached
- `super()`
  - `super()` operates by using the elements in an MRO that come after some specified type
  - `super()` returns a proxy object which forwards calls to the correct objects
  - There are two distinct types of `super()` proxies, bound and unbound
  - Unbound `super()` proxies are primarily used for implementing other Python features
  - Bound proxies can be bound to either class objects or instances
  - Calling `super()` with a base-class and derived-class argument returns a proxy bound to a class
  - Calling `super()` with a class and an instance of that class returns a proxy bound to an instance
  - A `super` proxy takes the MRO of its second argument (or the type of its second argument), finds the first argument in that MRO, and uses everything after it in the MRO for method resolution
  - Since class-bound proxies aren't bound to an instance, you can't directly call instance methods that they resolve for you
  - However, classmethods resolved by class-bound proxies can be called directly
  - Python will raise a `TypeError` if the second argument is not a subclass or instance of the first argument
  - Inappropriate use of `super()` can violate some design constraints
  - Calling `super()` with no arguments inside an instance method produces an instance-bound proxy
  - Calling `super()` with no arguments inside a classmethod produces a class-bound proxy
  - In both cases, the no-argument form of `super()` is the same as calling `super()` with the method's class as the first argument and the method's first argument as the second
  - Since `super()` works on MROs and not just a class's base-classes, classes can be designed to cooperate without a priori knowledge of one another
- `object`
  - The class `object` is at the core of Python's object model
  - `object` is the ultimate base-class for all other classes in Python

  - If you don't specify a base-class for a class, Python automatically uses `object` as the base
  - Because `object` is in every class's inheritance graph, it shows up in every MRO.
  - `object` provides hooks for Python's comparison operators
  - `object` provides default `__repr__()` and `__str__()` implementations
  - `object` implements the core attribute lookup and management functionality in Python
- Inheritance in Python is best used as a way to share implementation

Chapter 9 - Implementing Collections with Protocols

In chapter 5 of *The Python Apprentice*[70] we showed how the different built-in collections can be categorised according to which protocols they support. In this chapter we'll build on that and demonstrate how to go beyond the built-in collection types of `list`, `tuple`, `dict` and `set` by creating your own collections. We'll build a new and fully functioning collection type of our own design. This will require us to implement a series of different protocols, each of which represents a different capability of collections. The protocols we will cover are:

- The *container* protocol, which allows us to test for item membership in a collection.
- The *sized* protocol, which allows us to determine the number of items in a collection.
- The *iterable* protocol which yield the items in the collection one by one. We encountered the iterable protocol previously in chapter 7. Here we'll utilise that knowledge directly.
- The *sequence* protocol which supports random read access to collections.
- The *set* protocol which support various set operations such as set-intersection.

There are many further protocols for different collections which we won't cover here, but we will give you a good grounding in *how* to implement protocols so you can build on the foundation we provide.

Collection protocols

Each of the collection protocols is founded on the idea of implementing a specific set of methods — usually special "dunder" methods. We'll work through implementing five of these protocols in the course of building a `SortedSet` collection which has similar behaviour to a regular set but which stores its items in sorted order. It will also support the *sequence* protocol for random access by integer index.

[70] https://leanpub.com/python-apprentice

We'll follow a simple test driven development — or TDD —[71] approach to developing the collection, although we'll abbreviate a few steps to avoid distracting from our larger goals. We briefly covered unit testing in Python in chapter 10 of *The Python Apprentice*[72].

We'll be starting with a completely fresh project in which to build our new collection. We'll only be using two Python source files, one for tests and one for the collection itself, so you can use any Python development environment you'd like for this work.

Test first

Our first test suite, in `test_sorted_set.py`, looks like this:

```
# test_sorted_set.py

import unittest

from sorted_set import SortedSet


class TestConstruction(unittest.TestCase):

    def test_empty(self):
        s = SortedSet([])

    def test_from_sequence(self):
        s = SortedSet([7, 8, 3, 1])

    def test_with_duplicates(self):
        s = SortedSet([8, 8, 8])

    def test_from_iterable(self):
        def gen6842():
            yield 6
            yield 8
            yield 4
            yield 2
        g = gen6842()
        s = SortedSet(g)
```

[71] https://en.wikipedia.org/wiki/Test-driven_development
[72] https://leanpub.com/python-apprentice

```
if __name__ == '__main__':
    unittest.main()
```

The goal here is to ensure that we can successfully construct a `SortedSet` instance in a way that is typical for Python collections. We don't implement any particular protocol here, but it is sensible to ensure that our collections can be constructed from existing iterable sequences of various configurations, including being empty. We don't want to rely on the argument passed to the constructor being anything more sophisticated than an *iterable*, and using a generator-function to produce a generator object is as good a way as any to test for that.

Any attempt to run this falls at the first hurdle with a failure to import `SortedSet`. The tests in here exercise the initializer, checking that it exists and can be called with these arguments without causing an exception to be raised:

```
$ python test_sorted_test.py
EEEEE
======================================================================
ERROR: test_default_empty (__main__.TestConstruction)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_sorted_test.py", line 9, in test_empty
    s = SortedSet()
NameError: global name 'SortedSet' is not defined
. . .
```

The initializer

Let's go ahead and implement enough of `SortedSet` in `sorted_set.py` to pass these basic tests:

```
# sorted_set.py

class SortedSet:

    def __init__(self, items):
        self._items = sorted(items)
```

To get things going, we just need a `SortedSet` class which we can instantiate with existing collections. We need to be able to create our sorted set from sequences and iterables, and have it ignore duplicates. We've skipped a few TDD cycles here and made a design decision about implementing `SortedSet` around a regular `list` object. We create this list using the `sorted()` built-in function which always returns a `list` irrespective of which type of iterable object it is passed.

Now we can run our tests by directly running the test module:

```
$ python test_sorted_set.py
....
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

Great! The tests are passing, so we can move on.

Default initialization

We'd also like to be able to construct an empty `SortedSet` with no arguments. Although that isn't mandated by any of the collection protocols, it would make our collection consistent with the existing Python collections, which will eliminate a potential surprise to users of our class. Here's the very simple test:

```
# test_sorted_set.py

class TestConstruction(unittest.TestCase):
    # ...
    def test_default_empty(self):
        s = SortedSet()
```

This fails because our existing initializer implementation expects exactly one argument.

```
$ python test_sorted_set.py
E....
======================================================================
ERROR: test_default_empty (__main__.TestConstruction)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_sorted_set.py", line 29, in test_default_empty
    s = SortedSet()
TypeError: __init__() missing 1 required positional argument: 'items'

----------------------------------------------------------------------
Ran 5 tests in 0.001s

FAILED (errors=1)
```

We can fix this by updating the implementation of `__init__()`:

```
# sorted_set.py

class SortedSet:

    def __init__(self, items=None):
        self._items = sorted(items) if items is not None else []
```

We deliberately use `None` as the default argument, rather than an empty `list`, to avoid inadvertently mutating the default argument object which is only created once when the method is first defined.

The *container* protocol

The first protocol we will implement is the *container* protocol. The container protocol is the most fundamental of the collection protocols and simply allows us to determine whether a particular item is present in the collection. The container protocol is what underpins the the `in` and `not in` infix operators. Let's put together some tests for that:

```
# test_sorted_set.py

class TestContainerProtocol(unittest.TestCase):

    def setUp(self):
        self.s = SortedSet([6, 7, 3, 9])

    def test_positive_contained(self):
        self.assertTrue(6 in self.s)

    def test_negative_contained(self):
        self.assertFalse(2 in self.s)

    def test_positive_not_contained(self):
        self.assertTrue(5 not in self.s)

    def test_negative_not_contained(self):
        self.assertFalse(9 not in self.s)
```

Here we use the `setUp()` method to create a `SortedSet` test fixture. We then use four test methods to assert the four combinations of membership and non-membership, success and failure. These four tests are grouped into the `TestContainerProtocol` class, which is a `unittest.TestCase`, so we can easily execute this group of tests alone.

Of course, running the tests results in four failures. The error messages are worthy of inspection because they indicate that we *could* fix the problem by making `SortedSet` *iterable* rather than making it support the *container* protocol:

```
$ python test_sorted_set.py
.....EEEE
======================================================================
ERROR: test_negative_contained (__main__.TestContainerProtocol)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_sorted_set.py", line 41, in test_negative_contained
    self.assertFalse(2 in self.s)
TypeError: argument of type 'SortedSet' is not iterable

. . .
```

While this is true[73], we won't take that approach here. We will, however, implement the *iterable* protocol in due course.

To get our tests passing we'll implement the more restrictive *container* protocol, which is implemented by the special `__contains__()` method, which accepts a single argument which is the item for which to test for, and returns a boolean. Internally, our `__contains__()` implementation will just delegate to the membership test on the enclosed `list` object:

```
# sorted_set.py

class SortedSet:
    # ...
    def __contains__(self, item):
        return item in self._items
```

This gets our tests passing again.

If you're tempted to have `SortedSet.__contains__(self, item)` delegate directly to `self._items.__contains__(item)`, like in the following code fragment, you should resist the urge:

```
# sorted_set.py

class SortedSet:
    # ...
    def __contains__(self, item):
        return self._items.__contains__(item)  # Don't do this!
```

While this will work, it's generally poor form to invoke special methods directly, as this can bypass various optimisations and standard behaviours implemented in the global functions or operators which delegate to them.

The *sized* protocol

The next degree of sophistication above containment is being able to determine how many items are in a collection, which is handled by the the *sized* protocol. The built-in `len()` function delegates to this protocol, and must always returns a non-negative integer. Let's put some tests together for that:

[73] The machinery of the `in` and `not in` operators contains a fallback to exhaustive search through the iterable series of items.

```python
# test_sorted_set.py

class TestSizedProtocol(unittest.TestCase):

    def test_empty(self):
        s = SortedSet()
        self.assertEqual(len(s), 0)

    def test_one(self):
        s = SortedSet([42])
        self.assertEqual(len(s), 1)

    def test_ten(self):
        s = SortedSet(range(10))
        self.assertEqual(len(s), 10)

    def test_with_duplicates(self):
        s = SortedSet([5, 5, 5])
        self.assertEqual(len(s), 1)
```

Going on the old adage that computing science only has three interesting numbers – zero, one, and N – we test those three cases, using 10 in place of N. Our fourth test ensures that duplicate items passed to the constructor are only counted once — important behaviour for a set.

On running the tests, you'll get four failures of this form:

```
$ python test_sorted_set.py
.........EEEE
======================================================================
ERROR: test_empty (__main__.TestSizedProtocol)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_sorted_set.py", line 54, in test_empty
    self.assertEqual(len(s), 0)
TypeError: object of type 'SortedSet' has no len()

. . .
```

To get the tests passing, we'll need to implement the special `__len__()` method to which the `len()` built-in delegates:

```
# sorted_set.py

class SortedSet:
    # ...
    def __len__(self):
        return len(self._items)
```

Again we delegate to `len()` on the underlying `list`. This improves matters, but our fourth *sized* protocol test is still failing:

```
$ python test_sorted_set.py
............F
======================================================================
FAIL: test_with_duplicates (__main__.TestSizedProtocol)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_sorted_set.py", line 66, in test_with_duplicates
    self.assertEqual(len(s), 1)
AssertionError: 3 != 1

----------------------------------------------------------------------
Ran 13 tests in 0.002s

FAILED (failures=1)
```

Strictly, this test covers initializer behaviour rather than conformance to the *sized* protocol, but the most important thing is that we have this test somewhere in our test suite. To fix it we need to eliminate duplicate items in the series passed to the constructor. The easiest way to do that is to construct a regular `set` with these items[74]. We'll then use that `set` to construct the sorted list:

[74] It is common to use a temporary `set` in this way to eliminate duplicates. Although it may seem wasteful to construct a collection just for this purpose, the underlying implementation of `set` in C will be much faster than anything you can write in Python.

```
# sorted_set.py

class SortedSet:
    # ...
    def __init__(self, items=None):
        self._items = sorted(set(items)) if items is not None else []
```

This has the added advantage that the `set()` constructor is a good model for what we would like our constructor to support. By delegating to it directly, we can be sure we be sure that the `SortedSet` constructor is compatible with the `set()` constructor.

With this change in place, all of our tests our passing again, so it's time to move on.

The *iterable* protocol

We saw how to implement *iterable* in chapter 7. We simply need to provide the special `__iter__()` method which must return an iterator over the series of items. We'll write a couple of tests for that, one which calls the `iter()` built-in and one which uses a for-loop:

```
# test_sorted_set.py

class TestIterableProtocol(unittest.TestCase):

    def setUp(self):
        self.s = SortedSet([7, 2, 1, 1, 9])

    def test_iter(self):
        i = iter(self.s)
        self.assertEqual(next(i), 1)
        self.assertEqual(next(i), 2)
        self.assertEqual(next(i), 7)
        self.assertEqual(next(i), 9)
        with self.assertRaises(StopIteration):
            next(i)

    def test_for_loop(self):
        index = 0
        expected = [1, 2, 7, 9]
        for item in self.s:
            self.assertEqual(item, expected[index])
            index += 1
```

The `test_iter()` method obtains an iterator using the `iter()` built-in. It asserts that each value is the expected one, and that the iterator terminates the series normally by raising a `StopIteration` exception.

> Notice the use of the `assertRaises()` method of `TestCase`. This returns a context manager, so can be used in conjunction with a `with` statement. The code inside the `with`-block is expected expected to raise the specified exception

The somewhat clunky (and frankly unPythonic) formulation of the `test_for_loop()` method is so that we can have the for-loop *directly* iterate over our `SortedSet`. More elegant solutions, which would use for example `zip()`[75] to combine the expected and actual values, or `enumerate()` to generate the index, could be performing a quite different test on our class.

Both tests will fail, because we haven't yet implemented iteration.

Our simple implementation of `__iter__()` will delegate to the underlying `list`:

```
# sorted_set.py

class SortedSet:
    # ...
    def __iter__(self):
        return iter(self._items)
```

Most implementations of `__iter__()` will either delegate in this way to some underlying collection, or be implemented as a generator function. Remember, generator functions return generator objects, which are by definition iterators. This means that generator functions fulfil the requirements of `__iter__()`. Were we to do that in this case, `__iter__()` might look like this:

```
def __iter__(self):
    for item in self._items:
        yield item
```

[75] https://docs.python.org/3/library/functions.html#zip

In Python versions 3.3 or later, we can eliminate the loop by using the `yield from` syntax[76] for delegating to an iterable:

PEP 380 - Syntax for Delegating to a Subgenerator[77]

```
def __iter__(self):
    yield from self._items
```

You can use whichever form you prefer, although the first form using the list iterator is likely to be faster, and is perhaps less obtuse.

With 15 tests passing, and our *container*, *sized* and *iterable* protocols in place, we can now proceed to implement the *sequence* protocol.

The *sequence* protocol

The *sequence* protocol requires that — in addition to being an *iterable*, *sized*, and *container* — the collection supports element access with square brackets, slicing with square brackets, and production of a reverse iterator when passed to the `reversed()` built-in function. Furthermore, the collection should implement `index()` and `count()`, *and*, if it needs to support the informally defined *extended sequence protocol* it should support concatenation and repetition through the addition and multiplication infix operators.

Implementing a conforming sequence is a tall order! Let's chip away at it using our test-driven approach, and along the way we'll introduce some tools to reduce the magnitude of the task facing us.

First, the indexing and slicing. Both indexing and slicing using postfix square brackets and are supported by the `__getitem__()` special method which we briefly encountered in chapter 8. Our `__getitem__()` will provide for regular forward indexing and reverse indexing with negative integers. Including tests for the various edge cases our new test case looks like this:

[76] The `yield from` syntax was introduced in

[77] https://www.python.org/dev/peps/pep-0380/

```
# test_sorted_set.py

class TestSequenceProtocol(unittest.TestCase):

    def setUp(self):
        self.s = SortedSet([1, 4, 9, 13, 15])

    def test_index_zero(self):
        self.assertEqual(self.s[0], 1)

    def test_index_four(self):
        self.assertEqual(self.s[4], 15)

    def test_index_one_beyond_the_end(self):
        with self.assertRaises(IndexError):
            self.s[5]

    def test_index_minus_one(self):
        self.assertEqual(self.s[-1], 15)

    def test_index_minus_five(self):
        self.assertEqual(self.s[-5], 1)

    def test_index_one_before_the_beginning(self):
        with self.assertRaises(IndexError):
            self.s[-6]
```

All of these tests fail.

To get these tests passing, we need to implement the special `__getitem__()` method. This is easy enough since we can delegate to the list indexing operator:

```
# sorted_set.py

class SortedSet:
    # ...
    def __getitem__(self, index):
        return self._items[index]
```

Now let's add further tests for slicing to the `TestSequenceProtocol` test case:

```
# test_sorted_set.py

class TestSequenceProtocol(unittest.TestCase):
    # ...
    def test_slice_from_start(self):
        self.assertEqual(self.s[:3], SortedSet([1, 4, 9]))

    def test_slice_to_end(self):
        self.assertEqual(self.s[3:], SortedSet([13, 15]))

    def test_slice_empty(self):
        self.assertEqual(self.s[10:], SortedSet())

    def test_slice_arbitrary(self):
        self.assertEqual(self.s[2:4], SortedSet([9, 13]))

    def test_slice_full(self):
        self.assertEqual(self.s[:], self.s)
```

Notice how writing the tests first drives us to the design decision: slicing a `SortedSet` should return a `SortedSet`. There's a good chance that if we'd been doing test-after, rather than test-first, we'd have ended up with our slices returning `list`s instead. In fact, when we run the tests, we can see that that is exactly what happens: our tests are failing because the existing implementation, which delegates to `list`, returns `list` slices:

```
$ python test_sorted_test.py
.................FFFFF....
======================================================================
FAIL: test_slice_arbitrary (__main__.TestSequenceProtocol)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_sorted_test.py", line 125, in test_slice_arbitrary
    self.assertEqual(self.s[2:4], SortedSet([9, 13]))
AssertionError: [9, 13] != <sorted_set.SortedSet object at 0x107e27c50>
. . .
```

To remedy that, we'll need a slightly more sophisticated version of `__getitem__()` which detects whether it is being called with an index or a slice and acts accordingly. But what does the `index` argument of our `__getitem__()` method contain when it is called with slice syntax? Let's temporarily instrument our method to find out. We'll print the index value and the type of the index value:

```
# sorted_set.py

class SortedSet:
    # ...
    def __getitem__(self, index):
        print(index)
        print(type(index))
        return self._items[index]
```

Now run the tests again. This will produce, in the case of the `test_slice_arbitrary()` function, output like this:

```
slice(2, 4, None)
<class 'slice'>
```

This indicates that a `slice` object is passed, and it's configured with the arguments to the slice.[78] In our examples the `step` attribute for extended slicing is not used, so defaults to `None`.

It's sufficient for our purposes to detect when a `slice` object has been passed to `__getitem__()`. We can then wrap up the `list` we get back from delegating to the `list slice` in a new `SortedSet`:

```
# sorted_set.py

class SortedSet:
    # ...
    def __getitem__(self, index):
        result = self._items[index]
        return SortedSet(result) if isinstance(index, slice) else result
```

We do this by calling the list index operator as normal. We then conditionally wrap the result in a `SortedSet` on the basis of a type test on the `index` argument performed by the built-in `isinstance` function.

Let's run the tests again!

[78] We won't cover slice objects in detail here, except to point out that they can be manually constructed by calling the slice constructor which accepts integer `start`, `stop` and `step` arguments. These arguments can also be retrieved through attributes of the same name. You can read more about slice objects in the Python documentation

```
$ python test_sorted_test.py
.................FFFFF....
======================================================================
FAIL: test_slice_arbitrary (__main__.TestSequenceProtocol)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_sorted_test.py", line 125, in test_slice_arbitrary
    self.assertEqual(self.s[2:4], SortedSet([9, 13]))
AssertionError: <sorted_set.SortedSet object at 0x10bd02c18> != <sorted_set.SortedSet\
 object at 0x10bd02be0>
. . .
```

Oh dear! The tests are still failing, but now for a different reason:

A detour into `repr()`

It's difficult to tell exactly what's gone wrong here. The string representations of our `SortedSet` aren't very informative. We know how to fix that, though, by implementing the `__repr__()` special method.[79] This should return a string helpful to developers like us. We won't abandon TDD though, so first a couple of new tests in a new test case:

```
# test_sorted_set.py
. . .
class TestReprProtocol(unittest.TestCase):

    def test_repr_empty(self):
        s = SortedSet()
        self.assertEqual(repr(s), "SortedSet()")

    def test_repr_one(self):
        s = SortedSet([42, 40, 19])
        self.assertEqual(repr(s), "SortedSet([19, 40, 42])")
```

This takes us from five to seven failing tests. Implementing `__repr__()` will take us back down to five:

[79] See chapter 5 for all of the details on string representations.

```
# sorted_set.py

class SortedSet:
    # ...
    def __repr__(self):
      return "SortedSet({})".format(repr(self._items) if self._items else '')
```

We follow the lead of the built-in collections and render an argumentless constructor call into the string if the collection is empty. If there are items in the collection, we delegate to the `list` `repr()` to render the argument. Notice also that we use implicit conversion of the `self._items` list to `bool` in the conditional; if the collection is empty this expression evaluates to `False` in this boolean context.

Now when we run our tests, not only do the two `repr()` tests pass, but we get much more information about what's going wrong with the other tests:

```
$ python test_sorted_test.py
...................FFFFF....
======================================================================
FAIL: test_slice_arbitrary (__main__.TestSequenceProtocol)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_sorted_test.py", line 125, in test_slice_arbitrary
    self.assertEqual(self.s[2:4], SortedSet([9, 13]))
AssertionError: SortedSet([9, 13]) != SortedSet([9, 13])

======================================================================
FAIL: test_slice_empty (__main__.TestSequenceProtocol)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_sorted_test.py", line 122, in test_slice_empty
    self.assertEqual(self.s[10:], SortedSet())
AssertionError: SortedSet() != SortedSet()
. . .
```

In every case our actual sets appear to be equal to the expected sets, but in fact they are not! What's going on?

A detour into equality

The problem here is that by default, Python equality comparisons — which are inherited from the ultimate base-class, `object` — are for reference equality, rather than value equality or equivalence. So when we make a comparison between two `SortedSet` instances we are really asking, "Are these two SortedSet instances the same object?"

```
>>> from sorted_set import SortedSet
>>> SortedSet([1, 2, 3]) == SortedSet([1, 2, 3])
False
```

In other words, the equality operator is returning exactly the same result as the `is` operator which tests equality of identity:

```
>>> SortedSet([1, 2, 3]) is SortedSet([1, 2, 3])
False
```

In fact, this is the default behaviour for all Python objects unless the equality operator has been specialized for the objects in question. The reason we can get a different result with `list` equality is that the `list` type overrides the default equality implementation:

```
>>> [1, 2, 3] == [1, 2, 3]
True
```

To get the same behaviour for our `SortedSet` class, we should do the same. Continuing with our test-driven methodology, we first need a few more tests:

```
# test_sorted_set.py
. . .
class TestEqualityProtocol(unittest.TestCase):

    def test_positive_equal(self):
        self.assertTrue(SortedSet([4, 5, 6]) == SortedSet([6, 5, 4]))

    def test_negative_equal(self):
        self.assertFalse(SortedSet([4, 5, 6]) == SortedSet([1, 2, 3]))

    def test_type_mismatch(self):
        self.assertFalse(SortedSet([4, 5, 6]) == [4, 5, 6])

    def test_identical(self):
        s = SortedSet([10, 11, 12])
        self.assertTrue(s == s)
```

In these tests we've deliberately used `assertTrue()` and `assertFalse()` rather than `assertEqual()` and `assertNotEqual()`. This leaves the equality operator in plain sight, and it ensures that the tests are couched in terms of the equality operator only, not the inequality operator. When you run these tests, you'll see that the last three of these tests already pass, although for the wrong reasons.

Let's now implement a specialized equality for `SortedSet` by overriding the `__eq__()` special method. Our implementation is simple. It justs delegates to the same operator with the enclosed lists of the left-hand-side operand (in this case `self`) and right-hand- side operand (abbreviated as `rhs`):

```python
# sorted_set.py

class SortedSet:
    # ...
    def __eq__(self, rhs):
        return self._items == rhs._items
```

With this in place, let's run our tests again:

```
$ python test_sorted_test.py
............E...................
======================================================================
ERROR: test_type_mismatch (__main__.TestEqualityProtocol)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_sorted_test.py", line 151, in test_type_mismatch
    self.assertFalse(SortedSet([4, 5, 6]) == [4, 5, 6])
  File "/Users/abingham/sandbox/coll/sorted_set.py", line 23, in __eq__
    return self._items == rhs._items
AttributeError: 'list' object has no attribute '_items'

----------------------------------------------------------------------
Ran 32 tests in 0.002s

FAILED (errors=1)
```

Now our first equality test passes, but our type-mismatch test fails. We need to refine our implementation to perform a type check. We'll follow the Python convention of returning the special built-in singleton value `NotImplemented` if the types don't match:

```
# sorted_set.py

class SortedSet:
    # ...
    def __eq__(self, rhs):
        if not isinstance(rhs, SortedSet):
            return NotImplemented
        return self._items == rhs._items
```

> Notice that we *return* the `NotImplemented` object rather than *raising* a `NotImplementedError`. This is something of a curiosity in the Python language, and the runtime will use it to retry the comparison once with the arguments reversed, potentially giving a different implementation of `__eq__()` on another object chance to respond. You can read more about it at https://docs.python.org/3/reference/datamodel.html?highlight=**eq**#object.**eq**

With this change in place, all of our tests are passing again:

```
$ python test_sorted_test.py
................................
----------------------------------------------------------------------
Ran 32 tests in 0.002s

OK
```

Implementing inequality

Before returning to continue our implementation the *sequence* protocol, we should also ensure that value *inequality* works as expected. Here are the inequality tests:

```
# test_sorted_set.py

class TestInequalityProtocol(unittest.TestCase):

    def test_positive_unequal(self):
        self.assertTrue(SortedSet([4, 5, 6]) != SortedSet([1, 2, 3]))

    def test_negative_unequal(self):
        self.assertFalse(SortedSet([4, 5, 6]) != SortedSet([6, 5, 4]))

    def test_type_mismatch(self):
        self.assertTrue(SortedSet([1, 2, 3]) != [1, 2, 3])

    def test_identical(self):
        s = SortedSet([10, 11, 12])
        self.assertFalse(s != s)
```

These pass without any further changes to the implementation, demonstrating that Python will implement inequality by negating the equality operator[80]. That said, it is possible to override the `__ne__()` special method if inequality needs its own implementation.

After that extended detour into `repr()`, equality, and inequality, it's time to get back to implementing the sequence protocol. It's worth noting that our new collection is more useful and robust than it likely would have been had we not been following TDD. It's entirely possible we would not have remembered to specialize equality without being driven to do it by the failing tests.

Back to *sequence*

So far our collection is an *iterable*, *sized*, *container* which implements `__getitem__()` from the *sequence* protocol. But it still lacks support for reverse iterators as well as `index()` and `count()`.

The `reversed()` built-in function should return an iterator which yields the collection items in reverse order. Here are our tests for it, which we append to the `TestSequenceProtocol` test case:

[80] This is the case for Python 3, but not for Python 2 where you must walk the extra mile to implement `__ne__()` manually.

```
# test_sorted_set.py

class TestSequenceProtocol(unittest.TestCase):
    # ...
    def test_reversed(self):
        s = SortedSet([1, 3, 5, 7])
        r = reversed(s)
        self.assertEqual(next(r), 7)
        self.assertEqual(next(r), 5)
        self.assertEqual(next(r), 3)
        self.assertEqual(next(r), 1)
        with self.assertRaises(StopIteration):
            next(r)
```

Running it, we can see that we don't need to do any extra work. The test passes! By default, the implementation of `reversed()` will check for the presence of the `__reversed__()` special method and delegate to that. However, if `__reversed__()` has not been implemented but both `__getitem__()` and `__len__()` *are* supported, `reversed()` with produce an iterator that internally walks back through the sequence by decrementing an index. That's fine for our purposes, so we won't go to the trouble of overriding `__reversed__()`. If we did want or need to implement `__reversed__()` it would be straightforward to implement it as a generator function. This is also a great example of why you shouldn't call special protocol methods directly, which could cause you to miss out on some of these niceties.

Next on our list are `index()` and `count()`. Let's get some tests in place. First, the `index()` method, which should return the index of the first matching item in the collection, or raise a `ValueError`:

```
# test_sorted_set.py

class TestSequenceProtocol(unittest.TestCase):
    # ...
    def test_index_positive(self):
        s = SortedSet([1, 5, 8, 9])
        self.assertEqual(s.index(8), 2)

    def test_index_negative(self):
        s = SortedSet([1, 5, 8, 9])
        with self.assertRaises(ValueError):
            s.index(15)
```

Both of these tests fail because we haven't yet implemented `index()` itself, and being a regular method there are no opportunities for fallback mechanisms in the Python implementation to kick in. Clearly though, it would be possible in principle to implement `index()` in terms of methods we have already in place, such as `__getitem__()`.

Fortunately, such default implementations are available in the Python Standard Library in the base classes of the `collections.abc` module.[81] This module contains many classes which can be inherited by — or *mixed in* to — classes, taking some of the legwork out of collection protocol implementation.

In our case we want to use the `collections.abc.Sequence`[82] class which, if we provide implementations of `__getitem__` and `__len__` (as we have), will provide a whole raft of mixin methods, including `index()`. We make our `SortedSet` a subclass of `Sequence` like this:

```
# sorted_set.py

from collections.abc import Sequence


class SortedSet(Sequence):

    # ...
```

Now our `index()` unit tests pass!

We'll round off our implementation of the *sequence* protocol by ensuring that `count()` is also correctly implemented. Recall that `count()` returns the number of times a specified item occurs in a list. Here are our tests:

[81] `abc` is an acronym for Abstract Base Class. We cover these in detail in chapter 8 of *The Python Master*. For now, the mechanism is unimportant; it's sufficient to know that the base class we're using implements useful behavior which we can inherit.

[82] https://docs.python.org/3/library/collections.abc.html#collections.abc.Sequence

```
# test_sorted_set.py

class TestSequenceProtocol(unittest.TestCase):
    # ...
    def test_count_zero(self):
        s = SortedSet([1, 5, 7, 9])
        self.assertEqual(s.count(11), 0)

    def test_count_one(self):
        s = SortedSet([1, 5, 7, 9])
        self.assertEqual(s.count(7), 1)
```

Thanks to our inheritance of the `Sequence` abstract base class, this works immediately.

Improving the implementation of `count()`

Remember, however, that the default implementations inherited from the base-class have no knowledge of the implementation details or relevant characteristics of the data in our concrete class. As a result, these default implementations may not be optimally efficient. For example, the inherited `index()` implementation cannot exploit the fact that our list is sorted. Likewise the inherited `count()` implementation cannot exploit the fact that our collection never contains more than one item with a particular value; it will always inspect every element, even when that's unnecessary.

Fortunately there's nothing to stop us overriding the inherited implementations with more efficient versions. Let's look more closely at the performance of the `count()` method we have inherited by performing some simple experiments at the REPL. First we'll create a sorted set from 2000 randomly selected integers in the range 0 to 999:

```
>>> from sorted_set import SortedSet
>>> from random import randrange
>>> s = SortedSet(randrange(1000) for _ in range(2000))
```

We use a generator expression to evaluate the `random.randrange()` function two-thousand times. In our case, this resulted in the following `SortedSet`:

```
>>> s
SortedSet([0, 1, 2, 3, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 32, 33, 34, 35,
36, 37, 38, 40, 41, 42, 43, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55,
...
975, 976, 977, 978, 979, 980, 981, 983, 984, 986, 987, 988, 989, 990,
991, 993, 994, 995, 996, 997, 998, 999])
```

The set contains 865 elements:

```
>>> len(s)
865
```

As you can see, some of the values, such as four and six, are missing.

Now let's use the `count()` method to count the number of occurrences of each number in the range 0 to 999 inclusive:

```
>>> [s.count(i) for i in range(1000)]
[1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1,
1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1,
...
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0,
1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1,
1, 1, 0, 1, 1, 1, 1, 1, 1, 1]
```

This gives us an overview of which numbers are missing from the `SortedSet`. That all seemed quick enough, but now let's evaluate that list comprehension one hundred times. We'll use the `timeit` function from the `timeit` module to measure how long it takes. We introduced `timeit` module back in chapter 2:

```
>>> from timeit import timeit
>>> timeit(setup='from __main__ import s',
...        stmt='[s.count(i) for i in range(1000)]', number=100)
4.517851034004707
```

That took around four and half seconds.

We know that the `count()` operation on a `SortedSet` only ever returns zero or one. We also know that the `list` inside our set is always sorted, so we should be able to perform a binary search for the element in a time proportional to $\log n$ rather than n (where n is the number of elements in the set, and $\log n$ will always be much smaller than n).

We won't need to write the binary search ourselves, because there's a robust implementation in the Python standard library in the `bisect` module[83]. Here's our override of the `count()` implementation:

```
# sorted_set.py

from bisect import bisect_left

class SortedSet:
    # ...
    def count(self, item):
        index = bisect_left(self._items, item)
        found = (index != len(self._items)) and (self._items[index] == item)
        return int(found)
```

A quick check will show that our tests still pass. This method works by using the `bisect_left()` function to determine at which index `item` would need to be inserted into `self._items` in order to maintain sorted order. We then check that `index` is within the bounds of the list and whether the element at that index is equivalent to the item we're searching for. We assign this boolean value to the variable `found`. Finally we convert the `bool` to an `int` which results in zero and one for `False` and `True` respectively.

Let's run a comparable test (this time the set is slightly smaller owing to the inherent randomness in the process we are using to create it):

[83] https://docs.python.org/3/library/bisect.html

```
>>> from sorted_set import SortedSet
>>> from random import randrange
>>> s = SortedSet(randrange(1000) for _ in range(2000))
>>> len(s)
855
>>> from timeit import timeit
>>> timeit(setup='from __main__ import s',
...        stmt='[s.count(1) for i in range(1000)]', number=100)
0.18291257397504523
```

We see that this implementation is about 25 times faster with this dataset. We would expect progressively larger improvements with larger collections.

Refactoring common code

It may have occurred to you that our `found` variable in the new implementation of `count()` represents exactly the same result as would be obtained from the `__contains__()` method we wrote when implementing the *container* protocol. Let's refactor by extracting our efficient search implementation into `__contains__()` and using the membership test within `count()`. The `__contains__()` method becomes:

```
# sorted_set.py

class SortedSet:
    # ...
    def __contains__(self, item):
        index = bisect_left(self._items, item)
        return (index != len(self._items)) and self._items[index] == item
```

We replace the $\mathcal{O}(n)$ complexity list membership test with a superior $\mathcal{O}(\log n)$ test[84]. The `count()` method is now reduced to simply:

[84] $\mathcal{O}(\log n)$ is an example of *Big O Notation*. It is used to classify algorithms according to how their running time grows in relation to the input data size n. For an $\mathcal{O}(n)$ algorithm, the running time is directly proportional to n, so doubling the input size will double the run time. For an $\mathcal{O}(\log n)$ algorithm the running time is proportional to the logarithm of the size of the input data. For example, as n rises by a factor to ten from 100 to 1000, $\log_{10} n$ goes only from from 2 to 3, an increase in running time of only one-half. A few minutes playing with the `log()` and `log10()` functions in the Python `math` module should convince you that the base of the logarithm doesn't matter.

```
# sorted_set.py

class SortedSet:
    # ...
    def count(self, item):
        return int(item in self._items)
```

Oh, and all the tests still pass.

Improving the implementation of `index()`

The inherited `index()` implementation is also not very efficient. Knowing nothing about the nature of of `SortedSet`, it has no choice but to perform a relatively inefficient linear search. Let's fix that too, again using the `bisect` module. We already have tests in place, so we can proceed directly to overriding the method:

```
# sorted_set.py

class SortedSet:
    # ...
    def index(self, item):
        index = bisect_left(self._items, item)
        if (index != len(self._items)) and self._items[index] == item:
          return index
        raise ValueError("{} not found".format(repr(item)))
```

If you're offended by the repetition between `index()` and what we already have in `__contains__()` you could implement `__contains__()` in terms of `index()` like this:

```
# sorted_set.py

class SortedSet:
    # ...
    def __contains__(self, item):
        try:
            self.index(item)
            return True
        except ValueError:
            return False
```

This strikes us as a perhaps a little obtuse, however.

Do we still need to use `collections.abc.Sequence`?

At this point we've overridden all of the methods inherited from `collections.abc.Sequence` except for `__reversed__()`. We don't need to override that because there's already a fall back in the `reversed()` implementation to an algorithm that uses `__getitem__()` and `__len__()`. So is there still any value in inheriting from `Sequence`? There are none that are exercised by our tests — you can verify this by removing the inheritance relationship and noting that all of our tests still pass.

Nevertheless, one of the advantages of inheriting from the classes in `collections.abc` is that it becomes easy to determine whether a class implements a particular collection protocol. Here are some protocol tests using the built-in `issubclass()` function:

```
>>> from collections.abc import *
>>> issubclass(list, Sequence)
True
>>> issubclass(list, Sized)
True
>>> issubclass(dict, Mapping)
True
>>> issubclass(dict, Sized)
True
>>> issubclass(dict, Iterable)
True
```

One obvious question — to paraphrase the words of PEP 3119[85], where the collection abstract base classes are laid out — is whether this approach represents a retreat from "duck typing". However, the PEP states that is not required, only encouraged, to inherit from the appropriate ABCs when implementing collections. We'll take heed of that encouragement, and add some tests to cover inheritance from the `Container`, `Sized`, `Iterable` and `Sequence` ABCs:

[85] https://www.python.org/dev/peps/pep-3119/

```
# test_sorted_set.py

from collections.abc import Container, Iterable, Sequence, Sized


class TestContainerProtocol(unittest.TestCase):
    # ...
    def test_protocol(self):
        self.assertTrue(issubclass(SortedSet, Container))


class TestSizedProtocol(unittest.TestCase):
    # ...
    def test_protocol(self):
        self.assertTrue(issubclass(SortedSet, Sized))


class TestIterableProtocol(unittest.TestCase):
    # ...
    def test_protocol(self):
        self.assertTrue(issubclass(SortedSet, Iterable))


class TestSequenceProtocol(unittest.TestCase):
    # ...
    def test_protocol(self):
        self.assertTrue(issubclass(SortedSet, Sequence))
```

When you run these tests you may be surprised — perhaps even very surprised — that only the last of the four tests fails! We've encountered a powerful and yet little known feature of the `issubclass()` function in relation to the Abstract Base Class system. This feature allows it to take advantage of duck-typing without explicit inheritance. A full explanation is an advanced topic, beyond the scope of *this* book, but is covered in chapter 8 of *The Python Master*.

By reintroducing inheritance from `Sequence`, which inherits all of the other relevant protocols, we can get all of our tests passing again.

Implementing concatenation and repetition

Concatenation and repetition of sequences can be performed with the add and multiply operators. We cover this topic in *The Python Apprentice* with respect to strings, lists and tuples.

Neither of these two operators are enforced or supplied by the Sequence abstract base class, because they are part of the informally described *extended sequence protocol*, to which many Python sequence types such as list and str conform, but to which others, such as range do not.

It's not entirely obvious that SortedSet *should* support concatenation or repetition in the usual sense, since doing so the normal way would not be possible whilst maintaining the uniqueness and ordering invariants. We'll sidestep this issue by implementing concatenation as a set-union operator which results in a set containing all elements from both operands. Here are the tests:

```
# test_sorted_set.py

class TestSequenceProtocol(unittest.TestCase):
    # ...
    def test_concatenate_disjoint(self):
        s = SortedSet([1, 2, 3])
        t = SortedSet([4, 5, 6])
        self.assertEqual(s + t, SortedSet([1, 2, 3, 4, 5, 6]))

    def test_concatenate_equal(self):
        s = SortedSet([2, 4, 6])
        self.assertEqual(s + s, s)

    def test_concatenate_intersecting(self):
        s = SortedSet([1, 2, 3])
        t = SortedSet([3, 4, 5])
        self.assertEqual(s + t, SortedSet([1, 2, 3, 4, 5]))
```

To get these tests to pass, we need to implement support for the infix plus operator, which is done via the special method `__add__()`:

```
# sorted_set.py

from itertools import chain

class SortedSet(Sequence):
    # ...
    def __add__(self, rhs):
        return SortedSet(chain(self._items, rhs._items))
```

Rather than simply concatenating the enclosed lists of the two operands, which would could result in a large temporary intermediate list object, we use `itertools.chain()`[86]. This requires an additional import at the top of the module. `chain` streams all values from one operand and then the other into the `SortedSet` constructor. This simple and elegant implementation works as expected.

Finally, we should implement repetition, which for lists works like repeated concatenation to the original list. This will have no effect for `SortedSet`, so all we need to do is return a new object which is equivalent to the existing one, unless the multiplicand is less than one, in which case we should return an empty collection. Here are the unit tests:

```
# test_sorted_set.py

class TestSequenceProtocol(unittest.TestCase):
    # ...
    def test_repetition_zero_right(self):
        s = SortedSet([4, 5, 6])
        self.assertEqual(s * 0, SortedSet())

    def test_repetition_nonzero_right(self):
        s = SortedSet([4, 5, 6])
        self.assertEqual(s * 100, s)
```

To get these tests to pass, we need to implement the infix multiplication operator. This delegates to the `__mul__()` special method for cases where our class is on the left-hand side:

[86] https://docs.python.org/3/library/itertools.html#itertools.chain

```
# sorted_set.py

class SortedSet(Sequence):
    # ...
    def __mul__(self, rhs):
        return self if rhs > 0 else SortedSet()
```

Here we return `self` or an empty set depending on the value of the right-hand-side. Note that the only reason we can return simply `self` is because our `SortedSet` objects are immutable. If they were to be made mutable, it would be necessary to return a *copy* of the `self` object here. This could be achieved either by simply passing `self` to the `SortedSet()` constructor, or perhaps by implementing a more efficient `copy()` method.

Note that the `__mul__()` method is not invoked if we reverse the operands:

```
# test_sorted_set.py

class TestSequenceProtocol(unittest.TestCase):
    # ...
    def test_repetition_zero_left(self):
        s = SortedSet([4, 5, 6])
        self.assertEqual(0 * s, SortedSet())

    def test_repetition_nonzero_left(self):
        s = SortedSet([4, 5, 6])
        self.assertEqual(100 * s, s)
```

For cases where our class is on the right-hand-side, the `__rmul__()` operator must be implemented as well:

```
# sorted_set.py

class SortedSet(Sequence):
    # ...
    def __rmul__(self, lhs):
        return self * lhs
```

Since the order of the operands doesn't matter for repetition, `__rmul__()` simply delegates to `__mul__()`.

At this point we're doing very well! We have a `SortedSet` implementation that implements the *container*, *sized*, *iterable* and *sequence* protocols very comprehensively, robustly, and efficiently in around 50 lines of code.

The *set* protocol

Since we are implementing a set which maintains its elements in sorted order, it seems entirely reasonable that we should support the *set* protocol. Referring to the `collections.abc` documentation[87] we can see that there is an abstract base class called `Set`[88] with abstract methods `__contains__()`, `__iter__()` and `__len()__`. This brings us a bevy of special methods which implement various set operators including all the relational operators:

Special method	Infix Operator	Method on `set`	Meaning
`__le__()`	`<=`	`issubset()`	subset
`__lt__()`	`<`		proper subset
`__eq__()`	`==`		equal
`__ne__()`	`!=`		not equal
`__gt__()`	`>`		proper superset
`__ge__()`	`>=`	`issuperset()`	superset

These will allow the comparison of `SortedSet` objects in terms of subset and superset relationships. The abstract base-class only provides the special methods for the infix operator support. If we want the named methods equivalents — like the built-in `set` type — we're going to need to implement them ourselves, although of course we can define them in terms of the operators.

One important difference between the operator and method versions of these operations is that while the operators require that the operands both be of the same type, the method versions will accept any iterable series as an argument.

Tests for the *set* protocol

The fact that we have already overridden `__eq__()` is probably a good thing. Our version is likely to be more efficient that the default implementation inherited from the base-class.

Let's create unit tests for these infix relational operators and their named method equivalents. We won't review each in detail now because there are quite a few, but all are included with the example code associated with this book:

[87] https://docs.python.org/3/library/collections.abc.html

[88] https://docs.python.org/3/library/collections.abc.html#collections.abc.Set

```
# test_sorted_test.py

class TestRelationalSetProtocol(unittest.TestCase):

    def test_lt_positive(self):
        s = SortedSet({1, 2})
        t = SortedSet({1, 2, 3})
        self.assertTrue(s < t)

    def test_lt_negative(self):
        s = SortedSet({1, 2, 3})
        t = SortedSet({1, 2, 3})
        self.assertFalse(s < t)

    def test_le_lt_positive(self):
        s = SortedSet({1, 2})
        t = SortedSet({1, 2, 3})
        self.assertTrue(s <= t)

    def test_le_eq_positive(self):
        s = SortedSet({1, 2, 3})
        t = SortedSet({1, 2, 3})
        self.assertTrue(s <= t)

    def test_le_negative(self):
        s = SortedSet({1, 2, 3})
        t = SortedSet({1, 2})
        self.assertFalse(s <= t)

    def test_gt_positive(self):
        s = SortedSet({1, 2, 3})
        t = SortedSet({1, 2})
        self.assertTrue(s > t)

    def test_gt_negative(self):
        s = SortedSet({1, 2})
        t = SortedSet({1, 2, 3})
        self.assertFalse(s > t)

    def test_ge_gt_positive(self):
        s = SortedSet({1, 2, 3})
        t = SortedSet({1, 2})
        self.assertTrue(s > t)
    def test_ge_eq_positive(self):
        s = SortedSet({1, 2, 3})
        t = SortedSet({1, 2, 3})
        self.assertTrue(s >= t)

    def test_ge_negative(self):
        s = SortedSet({1, 2})
        t = SortedSet({1, 2, 3})
        self.assertFalse(s >= t)


class TestSetRelationalMethods(unittest.TestCase):

    def test_issubset_proper_positive(self):
        s = SortedSet({1, 2})
        t = [1, 2, 3]
        self.assertTrue(s.issubset(t))

    def test_issubset_positive(self):
        s = SortedSet({1, 2, 3})
        t = [1, 2, 3]
        self.assertTrue(s.issubset(t))

    def test_issubset_negative(self):
        s = SortedSet({1, 2, 3})
        t = [1, 2]
        self.assertFalse(s.issubset(t))

    def test_issuperset_proper_positive(self):
        s = SortedSet({1, 2, 3})
        t = [1, 2]
        self.assertTrue(s.issuperset(t))

    def test_issuperset_positive(self):
        s = SortedSet({1, 2, 3})
        t = [1, 2, 3]
        self.assertTrue(s.issuperset(t))

    def test_issuperset_negative(self):
        s = SortedSet({1, 2})
        t = [1, 2, 3]
        self.assertFalse(s.issuperset(t))
```

Predictably, all 16 of these new tests fail. We won't fix them just yet though — we'll plough onwards through the other necessary `Set` operators.

The `Set` base-class also provides mixins for the special methods corresponding to the bitwise operators. We covered the the bitwise-and and bitwise-or operators in the context of processing binary data files in chapter 9 of *The Python Apprentice*[89]. In the context of the *set* protocol, however, they are implemented to perform set-intersection and set-union operations respectively. Furthermore, the infix exclusive-or operator is configured to produce the symmetric-difference of two sets. We covered the long-hand methods for the set algebra operations `intersection()`, `union()` and `symmetric_difference()` in the chapter 5 of *The Python Apprentice*[90], although we did not cover these infix operator alternatives. The `Set` protocol also defines the subtraction operator to perform the set-difference operation, and the `set` class provides a corresponding `difference()` method:

Special method	Infix operator	Method on `set` class
`__and__()`	`&`	`intersection()`
`__or__()`	`|`	`union()`
`__xor__()`	`^`	`symmetric_difference()`
`__sub__()`	`-`	`difference()`

As with the relational operators, the `Set` abstract base-class provides the special methods which underlie the infix operators. However, it does *not* implement the equivalent public methods used by the `set` built-in class. If we want those, we're going to have to implement them ourselves. As before the operator versions expect operands of the same type, whereas the named methods accept any iterable series as the argument. Here are the tests for the operators and their corresponding named methods:

```python
# test_sorted_set.py

class TestOperationsSetProtocol(unittest.TestCase):

    def test_intersection(self):
        s = SortedSet({1, 2, 3})
        t = SortedSet({2, 3, 4})
        self.assertEqual(s & t, SortedSet({2, 3}))

    def test_union(self):
        s = SortedSet({1, 2, 3})
```

[89] https://leanpub.com/python-apprentice
[90] https://leanpub.com/python-apprentice

```
        t = SortedSet({2, 3, 4})
        self.assertEqual(s | t, SortedSet({1, 2, 3, 4}))

    def test_symmetric_difference(self):
        s = SortedSet({1, 2, 3})
        t = SortedSet({2, 3, 4})
        self.assertEqual(s ^ t, SortedSet({1, 4}))

    def test_difference(self):
        s = SortedSet({1, 2, 3})
        t = SortedSet({2, 3, 4})
        self.assertEqual(s - t, SortedSet({1}))


class TestSetOperationsMethods(unittest.TestCase):

    def test_intersection(self):
        s = SortedSet({1, 2, 3})
        t = [2, 3, 4]
        self.assertEqual(s.intersection(t), SortedSet({2, 3}))

    def test_union(self):
        s = SortedSet({1, 2, 3})
        t = [2, 3, 4]
        self.assertEqual(s.union(t), SortedSet({1, 2, 3, 4}))

    def test_symmetric_difference(self):
        s = SortedSet({1, 2, 3})
        t = [2, 3, 4]
        self.assertEqual(s.symmetric_difference(t), SortedSet({1, 4}))

    def test_difference(self):
        s = SortedSet({1, 2, 3})
        t = [2, 3, 4]
        self.assertEqual(s.difference(t), SortedSet({1}))
```

One last mixin method contributed by the `Set` abstract base class is the `isdisjoint()` method which tests whether two sets have elements in common. We'll round off our tests for the `Set` protocol by adding two new tests for that to the `TestSetRelationalSetProtocol` test case:

```
# test_sorted_set.py

class TestSetRelationalSetProtocol(unittest.TestCase):
    # ...
    def test_isdisjoint_positive(self):
        s = SortedSet({1, 2, 3})
        t = [4, 5, 6]
        self.assertTrue(s.isdisjoint(t))

    def test_isdisjoint_negative(self):
        s = SortedSet({1, 2, 3})
        t = [3, 4, 5]
        self.assertFalse(s.isdisjoint(t))
```

At this point we have a total of 79 unit tests of which 26 are failing. Let's make some code changes to get them passing.

The obvious first step is to have `SortedSet` inherit from the `Set` abstract base class. We'll use multiple inheritance for this so we can still inherit from `Sequence`, which also requires the additional import from `collections.abc`:

```
# sorted_set.py

from bisect import bisect_left
from collections.abc import Sequence, Set
from itertools import chain


class SortedSet(Sequence, Set):

    # ...
```

That simple change gets 16 out of 26 failing tests to pass, so now we are down to the 10 tests which cover the named methods not inherited from the `Set` abstract base class. Let's implement those in terms of the inherited operators, remembering to support any iterable series as the argument. We achieve this by constructing a `SortedSet` from each iterable series before applying the operator version:

```
# sorted_set.py

class SortedSet(Sequence, Set):
    # ...
    def issubset(self, iterable):
        return self <= SortedSet(iterable)

    def issuperset(self, iterable):
        return self >= SortedSet(iterable)

    def intersection(self, iterable):
        return self & SortedSet(iterable)

    def union(self, iterable):
        return self | SortedSet(iterable)

    def symmetric_difference(self, iterable):
        return self ^ SortedSet(iterable)

    def difference(self, iterable):
        return self - SortedSet(iterable)
```

Our `SortedSet` class is now functionally complete. We'll round off by asserting that we've implemented the `Set` protocol:

```
# test_sorted_set.py

from collections.abc import Container, Iterable, Sequence, Set, Sized


class TestSetProtocol(unittest.TestCase):

    def test_protocol(self):
        self.assertTrue(issubclass(SortedSet, Set))
```

You'll notice that we've implemented an immutable set — once constructed it's values cannot be modified. For many applications this is sufficient and indeed a good thing; immutable object are generally easier to reason about.

An exercise for the reader

If you need a mutable sorted set, it would be straightforward to inherit from the `MutableSet` abstract base-class[91] instead of the `Set` abstract base-class. You would need to implement the two additional required abstract methods we don't already have: `add()` and `discard()`. You may also want to follow the lead of the built-in `set` class and implement named methods such as `update()` and `symmetric_difference_update()` for completeness — this would allow `SortedSet` to be used as a drop-in replacement for `set()` in nearly all cases.

Finally, you might want to follow the lead of the other collections and provide a `copy()` method. Be aware that some of the methods we've implemented already rely on assumptions which don't hold for mutable sets. For example, under certain conditions `__mul__()` returns `self`. For a mutable set we'd want to return a copy of `self` instead. We'll leave making a mutable version of `SortedSet` as the proverbial exercise for the reader, which will also provide valuable experience in use of the `bisect` module.

Summary

In this chapter we covered how to implement your own collection classes which conform to standard Python protocols. In particular we covered these topics:

- Collection protocols
  - Showed how to implement the *container* protocol which supports the `in` and `not in` membership test operators by implementing the `__contains__()` special method.
  - Showed how to implement the *sized* protocol which allows the `len()` built-in function to be used, by implementing the `__len__()` special method.
  - Revisited the *iterable* protocol and its implementation using the `__iter__()` special method.
  - Introduced the *sequence* protocol which requires that a wide variety of operators, methods and behaviours be supported.
- Implementing protocols
  - Showed how to implement the `__getitem__()` special method to support both indexing and slicing.
  - Briefly touched on `slice` objects.
  - Implemented `__repr__()` for our class to aid testing.

[91] https://docs.python.org/3/library/collections.abc.html#collections.abc.MutableSet

  - Discovered that we needed to implement value equality testing via the `__eq__()` special method.
  - Found the value inequality works automatically in terms of negated equality.
  - Understood that, although support obtaining a reverse iterator from a collection via the `reversed()` built-in is a requirement for the *sequence* protocol, it wasn't necessary to implement the `__reversed__()` special method because the built-in function falls back to using `__getitem__()` and `__len__()`.
- Abstract base-classes and `collections.abc`
  - Introduced the `collections.abc` module, which contains base classes to define, and assist with implementing, the various collection protocols.
  - Used the `collections.abc.Sequence` abstract base class to provide functioning implementations many of the required methods and operators.
  - Showed how to test for protocol implementation using the `issubclass()` function in combination with the abstract base classes of the `collections.abc` module.
  - Implemented the *set* protocol using the same techniques.
- Other topics
  - Demonstrated how test-driven development approach led us to discovering some important requirements we may otherwise have forgotten.
  - Decreased the algorithmic complexity of our implementation from $\mathcal{O}(n)$ to $\mathcal{O}(\log n)$ by using the standard library `bisect` module to perform binary searches.
  - Introduced the reinterpretation of the bitwise operators in a set context, where they perform set relational operations.

Chapter 10 - Errors and Exceptions in Depth

In Chapter 6 of *The Python Apprentice*[92] we introduced exceptions and exception techniques. There we covered the basic concepts of exceptions and the specifics of how to handle them in Python. We also gave advice on when and how to deploy exception raising and exception handling in your code. This chapter builds directly on the foundation we established in *The Python Apprentice*, and here we seek to deepen our understanding of some of the tools Python provide for sophisticated error handling and reporting.

Exception dangers

We'll start by reminding ourselves of the basic exception handling and raising constructs, and we'll highlight a practice to avoid.

Consider this simple program which uses `random.randrange()` to choose a number between 0 and 99 inclusive. We repeatedly ask the user to guess the number, breaking out of the loop if they get the answer right:

```
#!/usr/bin/env python3

from random import randrange


def main():
    number = randrange(100)
    while True:
        guess = int(input("? "))
        if guess == number:
            print("You win!")
            break

if __name__ == '__main__':
    main()
```

[92] https://leanpub.com/python-apprentice

Let's give it a whirl:

```
$ python3 handler.py
? 10
? 37
? 53
? 22
? ^CTraceback (most recent call last):
  File "handler.py", line 15, in <module>
    main()
  File "handler.py", line 9, in main
    guess = int(input("? "))
KeyboardInterrupt
```

This is quite a boring game, so if we're not successful after a handful of attempts, we can press `Ctrl-C` to exit the game. Let's have another go:

```
$ python3 handler.py
? 45
? 21
? 99
? seven
Traceback (most recent call last):
  File "handler.py", line 15, in <module>
    main()
  File "handler.py", line 9, in main
    guess = int(input("? "))
ValueError: invalid literal for int() with base 10: 'seven'
```

This time we caused the program to fail with invalid input. We use the `int()` constructor to convert the string returned by the `input()` function to an integer. When we use the word "seven" rather than the digit 7, that conversion raises an exception which is unhandled, and so the program exits. Let's fix our program, by incorporating an exception handler around the problem statement:

```
#!/usr/bin/env python3

from random import randrange


def main():
    number = randrange(100)
    while True:
        try:
           guess = int(input("? "))
        except:
            continue
        if guess == number:
            print("You win!")
            break

if __name__ == '__main__':
    main()
```

In the exception handler we use a `continue` statement to proceed immediately with the next iteration of the innermost loop — the while loop in this case.

Let's try it:

```
? 10
? 17
? seven
? 9
?
```

When you're bored, press Ctrl-C to exit the program:

```
? ^C? ^C? ^C? ^C? ^C?
```

Oh dear! By not specifying an exception class for the `except` statement, we handled *all* exceptions, including the `KeyboardInterrupt` exception which is raised when we press `Ctrl-C`. Catching all exceptions is, in general, a very bad idea. Practically everything that can go wrong in a Python program manifests as an exception.

To see this vividly, try replacing `guess = int(input("? "))` with `guess = foo(input("? "))`. When run, this program will go into an infinite loop of repeatedly raising and handling the `NameError` that is raised by the unknown `foo()` function.

The solution here is to catch the specific exception that we're interested in, namely `ValueError`:

```
def main():
    number = randrange(100)
    while True:
        try:
           guess = foo(input("? "))
        except ValueError:
            continue
        if guess == number:
            print("You win!")
            break
```

With that change in place, the `NameError` propagates out and terminates the program:

```
Traceback (most recent call last):
  File "examples/exceptions/handler.py", line 18, in <module>
    main()
  File "examples/exceptions/handler.py", line 10, in main
    guess = foo(input("? "))
NameError: global name 'foo' is not defined
```

If you revert to using the `int()` function again, our program now works as intended:

```
$ python3 handler.py
? 10
? 15
? seven
? 3
? ^CTraceback (most recent call last):
  File "handler.py", line 18, in <module>
    main()
  File "handler.py", line 10, in main
    guess = int(input("? "))
KeyboardInterrupt
```

In summary, you should almost always avoid omitting the exception class from an except statement, since handling all exceptions in the way is seldom required and is usually a mistake.[93]

[93] Some people call such `except` clauses "Pokemon excepts" because they "catch them all".

Exceptions hierarchies

The built-in exception classes, of which there are many, are arranged into a class hierarchy using inheritance[94]. This is significant because when you specify an exception class in an `except` statement, any class which is a subclass of the specified class will be caught, in addition to the specified class itself. Let's look at two built-in exception types with which we are already familiar: `IndexError` and `KeyError`.

An `IndexError` is raised whenever we attempt an out-of-range lookup into a sequence type, such as a `list`:

```
>>> s = [1, 4, 6]
>>> s[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

A `KeyError` is raised when we lookup a missing key in a mapping type such as a dictionary:

```
>>> d = dict(a=65, b=66, c=67)
>>> d['x']
>>> d['x']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'x'
```

Let's investigate the inheritance hierarchy of these two exception types. From Chapter 8[95] we know we can retrieve transitive base classes from a class object using the `__mro__`-attribute which contains the method resolution order of class object as tuple. Let's try it on `IndexError`:

```
>>> IndexError.__mro__
(<class 'IndexError'>, <class 'LookupError'>, <class 'Exception'>,
 <class 'BaseException'>, <class 'object'>)
```

[94] https://docs.python.org/3/library/exceptions.html#exception-hierarchy

[95] chapter-8

This shows the full exception class hierarchy from `object` (the root of all class hierarchies) down through `BaseException` (the root of all exceptions), a class called `Exception` (which we'll return to shortly) to `LookupError`, and finally `IndexError`.

Now we'll try the same exercise for `KeyError`:

```
>>> KeyError.__mro__
(<class 'KeyError'>, <class 'LookupError'>, <class 'Exception'>,
 <class 'BaseException'>, <class 'object'>)
```

We can see that `KeyError` is also an immediate subclass of `LookupError`, and so `IndexError` and `KeyError` must be siblings in the class hierarchy.

What this means in practice is that we can catch both `IndexError` and `KeyError` exceptions by catching `LookupError`. Here's a short program which raises and handles one `IndexError` and one `KeyError`:

```
def lookups():
    s = [1, 4, 6]
    try:
        item = s[5]
    except IndexError:
        print("Handled IndexError")

    d = dict(a=65, b=66, c=67)
    try:
        value = d['x']
    except KeyError:
        print("Handled KeyError")

if __name__ == '__main__':
    lookups()
```

It behaves as expected when run:

```
Handled IndexError
Handled KeyError
```

Now we'll modify the program to handle `LookupError`s instead of the more specific exception classes:

```
def lookups():
    s = [1, 4, 6]
    try:
        item = s[5]
    except LookupError:
        print("Handled IndexError")

    d = dict(a=65, b=66, c=67)
    try:
        value = d['x']
    except LookupError:
        print("Handled KeyError")

if __name__ == '__main__':
    lookups()
```

This behaves identically:

```
Handled IndexError
Handled KeyError
```

Let's look at the full hierarchy for built in exceptions. We've met many of these already:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
```

```
+-- MemoryError
+-- NameError
|    +-- UnboundLocalError
+-- OSError
|    +-- BlockingIOError
|    +-- ChildProcessError
|    +-- ConnectionError
|    |    +-- BrokenPipeError
|    |    +-- ConnectionAbortedError
|    |    +-- ConnectionRefusedError
|    |    +-- ConnectionResetError
|    +-- FileExistsError
|    +-- FileNotFoundError
|    +-- InterruptedError
|    +-- IsADirectoryError
|    +-- NotADirectoryError
|    +-- PermissionError
|    +-- ProcessLookupError
|    +-- TimeoutError
+-- ReferenceError
+-- RuntimeError
|    +-- NotImplementedError
+-- SyntaxError
|    +-- IndentationError
|         +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
|    +-- UnicodeError
|         +-- UnicodeDecodeError
|         +-- UnicodeEncodeError
|         +-- UnicodeTranslateError
+-- Warning
     +-- DeprecationWarning
     +-- PendingDeprecationWarning
     +-- RuntimeWarning
     +-- SyntaxWarning
     +-- UserWarning
     +-- FutureWarning
     +-- ImportWarning
     +-- UnicodeWarning
     +-- BytesWarning
     +-- ResourceWarning
```

Over the history of Python there have been several changes to the exception hierarchy, so it's worthwhile checking the details for the exact interpreter versions you'll be using if your code needs to be portable. The hierarchy we're showing here is for Python 3.3. You'll see that `BaseException` is at the root of the hierarchy. The only exception which derives from `BaseException` other than the `Exception` class we'll come to shortly, are the so-called system-exiting exceptions, most notably `SystemExit` and `KeyboardInterrupt`. We've already witnessed the untoward effects of intercepting and swallowing `KeyboardInterrupt` exceptions. Likewise, inadvertent handling of `SystemExit` — which is raised by the `sys.exit()` function when a process is programmatically terminated — causes similar problems.

The majority of exceptions derive from `Exception`, and so if you want to catch all exceptions aside from the system-exiting exceptions, you might be tempted to catch this. Note, however, that a whole host of exception types typically associated with programming mistakes, such as `SyntaxError`, `IndentationError`, `TabError`, `NameError`, `UnboundLocalError`, `AssertionError` and `ImportError` are also subclasses of `Exception`, so handling `Exception` has the potential to hide serious problems.

In general, we encourage you to handle as specific exceptions as possible. At the same time, `OSError` is particular is useful for detecting that *something* has gone wrong with a filesystem operation without needing to worry about the details: a missing file indicated by `FileNotFoundError`, permissions problem indicated by `PermissionError`, or something else. Even though we may catch a general type of error, the exception object we receive retains its original type and any exception payload, so we're not passing up on opportunities for detailed error reporting based on exception payloads.

Exception payloads

Most exception objects carry a simple payload which contains diagnostic information about what caused the exception. The majority of the built in exception types accept a simple string in the constructor call. The exception type you will raise most frequently is probably `ValueError` which is often used for argument validation guard clauses near the beginning of functions. Consider this function for determining the median value of an iterable series:

```
def median(iterable):
    """Obtain the central value of a series.

    Sorts the iterable and returns the middle value if there is an even
    number of elements, or the arithmetic mean of the middle two elements
    if there is an even number of elements.

    Args:
        iterable: A series of orderable items.

    Returns:
        The median value.
    """
    items = sorted(iterable)
    median_index = (len(items) - 1) // 2
    if len(items) % 2 != 0:
        return items[median_index]
    return (items[median_index] + items[median_index + 1]) / 2.0
```

Let's try this on a few series:

```
>>> median([5, 2, 1, 4, 3])
3
>>> median([5, 2, 1, 4, 3, 6])
3.5
```

So far, so good. But look what happens when we supply an empty list:

```
>>> median([])
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "examples/exceptions/payload.py", line 18, in median
    return (items[median_index] + items[median_index + 1]) / 2.0
IndexError: list index out of range
```

We get an `IndexError` containing a message payload displayed in the stacktrace "list index out of range". This is all very well, since we can't define the concept of 'median' for an empty series. But we're leaking an implementation detail of our function here, namely that internally we're using a sequence lookup to perform the computation.

Let's add a guard clause which checks that the supplied series is non-empty:

```
def median(iterable):
    """Obtain the central value of a series.

    Sorts the iterable and returns the middle value if there is an even
    number of elements, or the arithmetic mean of the middle two elements
    if there is an even number of elements.

    Args:
        iterable: A series of orderable items.

    Returns:
        The median value.
    """
    items = sorted(iterable)
    if len(items) == 0:
        raise ValueError("median() arg is an empty series")

    median_index = (len(items) - 1) // 2
    if len(items) % 2 != 0:
        return items[median_index]
    return (items[median_index] + items[median_index + 1]) / 2
```

Now we get a more relevant error message in our stacktrace:

```
>>> median([])
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "examples/exceptions/payload.py", line 16, in median
    raise ValueError("median() arg is an empty sequence")
ValueError: median() arg is an empty series
```

Most usually, exception payloads are strings and are passed as a single argument to the exception constructor. The string should contain as helpful a message as possible.

Accessing payloads through `args`

We can programmatically retrieve the message too, using the `args` exception attribute. Here, we add a function to exercise our `median()` function with faulty input, catch the `ValueError`, and print the payload stored in its `args` attribute:

```
def main():
    try:
        median([])
    except ValueError as e:
        print("Payload", e.args)

if __name__ == '__main__':
    main()
```

When run, notice that `args` is a single element tuple containing the message that was passed to the constructor:

```
Payload: ('median() arg is an empty sequence',)
```

Another way to retrieve the payload in string form is to convert the exception object to a string using the `str()` or `repr()` functions.

Although you might infer that multiple arguments can be passed to the exception constructor, and that these will be available in the `args` tuple — and you would be right — you should only pass a single string argument to exception constructors. PEP 352[96] is quite clear on the matter:

> "No restriction is placed upon what may be passed in for `args` for backwards-compatibility reasons. In practice, though, only a single string argument should be used."

This means that you should only expect the `args` attribute to contain a single string value, which in any case you could retrieve by converting the exception object to a string rather than retrieving `args[0]`.

That said, specific exception classes may provide additional specific named attributes which contain further information about the cause; `UnicodeError` is one such example; it has five additional named attributes:

[96] https://www.python.org/dev/peps/pep-0352/

```
>>> try:
...     b'\x81'.decode('utf-8')
... except UnicodeError as e:
...     print(e)
...     print('encoding:', e.encoding)
...     print('reason:', e.reason)
...     print('object:', e.object)
...     print('start:', e.start)
...     print('end:', e.end)
...
'utf-8' codec can't decode byte 0x81 in position 0: invalid start byte
encoding: utf-8
reason: invalid start byte
object: b'\x81'
start: 0
end: 1
```

User-defined exceptions

When your needs aren't adequately met by of the built-in exceptions, you can define your own. Consider this function, which uses Heron's formula[97] to compute the area of a triangle given the length of three sides:

```
import math


def triangle_area(a, b, c):
    p = (a + b + c) / 2
    a = math.sqrt(p * (p - a) * (p - b) * (p - c))
    return a
```

This works well for side lengths that represent legitimate triangles:

```
>>> from heron import triangle_area
>>> triangle_area(3, 4, 5)
6.0
```

But if no such triangle with these side lengths exists, we get a `ValueError` from an attempt to find a real square root of a negative number:

[97] https://en.wikipedia.org/wiki/Heron%27s_formula

```
>>> triangle_area(3, 4, 10)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "examples/exceptions/heron.py", line 6, in triangle_area
    a = math.sqrt(p * (p - a) * (p - b) * (p - c))
ValueError: math domain error
```

Rather than the obscure math domain error message, we'd prefer to raise a more specific exception here which can carry more useful information in its payload. A good start is to define our own exception class `TriangleError`.

A basic exception implementation

When doing this you should subclass `Exception` rather than `BaseException`. If you just want a distinct exception type with basic facilities which can be raised and handled separately from other exception types, the most basic definition can suffice:

```
class TriangleError(Exception):
    pass
```

This is a fully functioning exception since it inherits complete implementations of `__init__()`, `__str__()` and `__repr__()`. Let's modify our function to identify illegal triangles:

```
import math


class TriangleError(Exception):
    pass


def triangle_area(a, b, c):
    sides = sorted((a, b, c))
    if sides[2] > sides[0] + sides[1]:
        raise TriangleError("Illegal triangle")

    p = (a + b + c) / 2
    a = math.sqrt(p * (p - a) * (p - b) * (p - c))
    return a
```

This works as expected:

```
>>> from heron import triangle_area
>>> triangle_area(3, 4, 10)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "examples/exceptions/heron.py", line 11, in triangle_area
    raise TriangleError("Illegal triangle")
heron.TriangleError: Illegal triangle
```

Enriching the payload

Now let's modify our exception to accept more data about the putative triangle:

```
import math


class TriangleError(Exception):

    def __init__(self, text, sides):
        super().__init__(text)
        self._sides = tuple(sides)

    @property
    def sides(self):
        return self._sides

    def __str__(self):
        return "'{}' for sides {}".format(self.args[0], self._sides)

    def __repr__(self):
        return "TriangleError({!r}, {!r}".format(self.args[0], self._sides)
```

Our exception now overrides `__init__()` and provides a constructor which accepts a message and collection of side lengths. The message is forwarded to the base-class constructor for storage, and the side lengths are stored in an instance attribute in the derived class.

We store the side lengths in a tuple to prevent modification, and we provide a read-only attribute to access them. We also override the `__str__()` and `__repr__()` methods, using the `args` attribute from the base-class to retrieve our message string.

We must also remember to modify the constructor call for the exception:

```
def triangle_area(a, b, c):
    sides = sorted((a, b, c))
    if sides[2] > sides[0] + sides[1]:
        raise TriangleError("Illegal triangle", sides)

    p = (a + b + c) / 2
    a = math.sqrt(p * (p - a) * (p - b) * (p - c))
    return a
```

Now when we feed an illegal triangle into the function, we get a better error report:

```
>>> triangle_area(3, 4, 10)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "examples/exceptions/heron.py", line 25, in triangle_area
    raise TriangleError("Illegal triangle", sides)
heron.TriangleError: 'Illegal triangle' for sides (3, 4, 10)
```

Moreover, with an appropriate handler in place, we can get access to the side lengths which caused the problem:

```
>>> from heron import TriangleError
>>> try:
...     triangle_area(3, 4, 10)
... except TriangleError as e:
...     print(e.sides)
...
(3, 4, 10)
```

Exception chaining

Exception chaining allows us to associate one exception with another, and it has two main use cases. The first case is when, during processing of one exception, another exception occurs, usually in a way incidental to the first exception. The second case is when we wish to deliberately handle an exception by translating it into a different exception type. In both cases there are good reasons for wanting to keep a reference to the original exception: it can avoid unnecessary duplication of information and it can improve diagnostic messages. Let's look at each of these two cases in turn.

Implicit chaining

The first case is called implicit chaining and occurs when one exception occurs whilst another is being processed. The Python runtime machinery associates the original exception with the new exception by setting the special `__context__` attribute of the most recent exception. Let's demonstrate this by adding a `main()` function which contains two bugs to our triangle area program. The first bug is that we try to evaluate the area of a non-triangle with sides 3, 4 and 10. The second bug is that in the process of handling the resulting `TriangleException` we cause an `io.UnsupportedOperation` exception by trying to print to the read-only `stdin` stream instead of the `stderr` stream we intended:

```
import sys


def main():
    try:
        a = triangle_area(3, 4, 10)
        print(a)
    except TriangleError as e:
        print(e, file=sys.stdin)


if __name__ == '__main__':
    main()
```

Here's the stack trace we get when we run the program:

```
Traceback (most recent call last):
  File "examples/exceptions/heron.py", line 34, in main
    a = triangle_area(3, 4, 10)
  File "examples/exceptions/heron.py", line 25, in triangle_area
    raise TriangleError("Illegal triangle", sides)
__main__.TriangleError: 'Illegal triangle' for sides (3, 4, 10)

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "examples/exceptions/heron.py", line 41, in <module>
    main()
  File "examples/exceptions/heron.py", line 37, in main
```

```
    print(e, file=sys.stdin)
io.UnsupportedOperation: not writable
```

See how, although the `TriangleError` was handled by our `except` block, it is still reported in the trace with the message "During handling of the above exception, another exception occurred". Python is able to give such a detailed report because the `TriangleError` has been attached to the `__context__` attribute of the `UnsupportedOperation` exception object. We'll temporarily add some code to demonstrate this:

```
def main():
    try:
        a = triangle_area(3, 4, 10)
        print(a)
    except TriangleError as e:
        try:
            print(e, file=sys.stdin)
        except io.UnsupportedOperation as f:
            print(e)
            print(f)
            print(f.__context__ is e)
```

We've added another exception handler around our faulty print statement and we print `e`, `f`, and the result of evaluting whether `f.__context__` *is* `e`:

```
TriangleError(Illegal triangle, (3, 4, 10)
UnsupportedOperation('not writable',)
True
```

And indeed it is. Because we don't need to do anything differently to associate these exceptions, this chaining is called *implicit*.

Explicit chaining

The converse of *implicit* chaining is *explicit* chaining. This is when we deliberately associate an existing exception instance with a new exception object at the point the latter is raised. This is done in the process of translating an exception from one type to another. Consider the following simple module:

```
import math


def inclination(dx, dy):
    return math.degrees(math.atan(dy / dx))
```

The inclination function returns the slope in degrees given the horizontal and vertical distance components of a line. This works fine for most slopes:

```
>>> from chaining import inclination
>>> inclination(3, 5)
59.03624346792648
```

But it fails with `ZeroDivisionError` when the horizontal component `dx` is zero:

```
>>> inclination(0, 5)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "examples/exceptions/chaining.py", line 5, in inclination
    return math.degrees(math.atan(dy / dx))
ZeroDivisionError: division by zero
```

Now let's modify the code by introducing a new exception type, `InclinationError`, and an exception handler for the `ZeroDivisionError` that swallows the active exception and raises a new one — in essence translating one exception to another:

```
import math


class InclinationError(Exception):
    pass


def inclination(dx, dy):
    try:
        return math.degrees(math.atan(dy / dx))
    except ZeroDivisionError as e:
        raise InclinationError("Slope cannot be vertical") from e
```

We have included the syntax for *explicit* exception chaining here with the `from e` suffix when we create the exception. This associates the new exception object with the original exception `e`. However, unlike the *implicit* chaining which associates the chained exception through the `__context__` attribute, *explicit* chaining associates the chained exception through the `__cause__` attribute.

Let see what happens when we trigger the exception:

```
>>> from chaining import *
>>> inclination(0, 5)
Traceback (most recent call last):
  File "examples/chaining.py", line 10, in inclination
    return math.degrees(math.atan(dy / dx))
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "examples/chaining.py", line 12, in inclination
    raise InclinationError("Slope cannot be vertical") from e
chaining.InclinationError: Slope cannot be vertical
```

Like implicitly chained exceptions, the default stack trace report includes the chained exception. If you want to examine the chained exception directly you can use the `__cause__` attribute of the most recent exception:

```
>>> try:
...     inclination(0, 5)
... except InclinationError as e:
...     print(e)
...     print(e.__cause__)
...
Slope cannot be vertical
division by zero
```

Whenever you translate exceptions in this way — typically at module boundaries where you should be hiding implementation details by raising exceptions your clients can reasonably expect — consider whether to explicitly chain root cause exceptions to improve diagnostics and aid debugging.

Tracebacks

We've mentioned many times that "everything in Python is an object", and this even extends to tracebacks — those records of the function call stack which are printed by the interpreter when an exception is unhandled and the program exits.

In Python 3, each exception object has a `__traceback__` attribute which contains a reference to the `traceback` object associated with that exception. Let's add a `main()` function to our `chaining.py` example to play with the traceback object:

```
def main():
    try:
        inclination(0, 5)
    except InclinationError as e:
        print(e.__traceback__)

if __name__ == '__main__':
    main()
    print("Finished")
```

This gives:

```
<traceback object at 0x102162320>
Finished
```

Shortly, it will become apparent why we've decided to explicitly print "Finished" before the program exits normally.

The `traceback` module

To do anything useful with the traceback object use the Python Standard Library `traceback` module. It contains functions for interrogating `traceback` objects. To display a traceback, we can use the `print_tb()` function:

```
$ python3 chaining.py
<traceback object at 0x1006e5908>
  File "chaining.py", line 18, in main
    inclination(0, 5)
  File "chaining.py", line 13, in inclination
    raise InclinationError("Slope cannot be vertical") from e
Finished
```

See how the program continues running after we've printed the traceback; the exception is being handled and the program has exited normally. The ability to get hold of `traceback` objects in this way is invaluable for logging diagnostic output. If you need to render the `traceback` object into a string, rather than printing it directly, you can use the `format_tb()` function instead of `print_tb()`.

> One word of caution here about keeping references to the traceback object: always render the output you need from a traceback object within the dynamic scope of the `except` block. That is, you shouldn't store a `traceback` — or indeed exception — object for later use. This is because the `traceback` object contains references to all the stack frame objects which comprise the call stack, and each stack frame contains references to all of its local variables. As such, the size of the transitive closure of objects reachable from the `traceback` object can be very large, and if you maintain that reference these objects will not be garbage collected. Prefer to render tracebacks into another form for even short-term storage in memory.

Assertions

The Python language includes an `assert` statement the purpose of which is to help you prevent bugs creeping into your code, and, when they do, to help you find them more quickly. The form of the `assert` statement is:

```
assert condition [, message]
```

`condition` is a boolean expression and `message` is an optional string for an error message. If `condition` is `False`, an `AssertionError` exception is raised, causing the program to terminate. If `message` is supplied, it is used as the exception payload. Here's an example:

```
>>> assert False, "The condition was false."
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: The condition was false
```

The purpose of the assertion statement is to give you a convenient means for monitoring program *invariants*, which are conditions which should **always** be true for your program. If for some reason an assertion fails, it will always point to a programming error: either some other part of the program is wrong, or, at the very least, the assertion statement itself is incorrect.

If the assertion condition is `True`, the statement has no effect:

```
>>> assert 5 > 2, "You are in a defective universe!"
>>>
```

Assertions are best used to document any assumptions your code makes, such as a name being bound to an object other than `None`, or a `list` being sorted at a particular point in the program. There are many good, and some very bad, places to use assertions in your programs.

Internal invariants

Often, you will see comments in code which document an assumption, particularly in conjunction with else-blocks like this:

```
def modulus_three(n):
    r = n % 3
    if r == 0:
        print("Multiple of 3")
    elif r == 1:
        print("Remainder 1")
    else:  # r == 2
        print("Remainder 2")
```

Comments such as this are much better reformulated as assertions, which can be check for truth at runtime:

```
def modulus_three(n):
    r = n % 3
    if r == 0:
        print("Multiple of 3")
    elif r == 1:
        print("Remainder 1")
    else:
        assert r == 2, "Remainder is not 2"
        print("Remainder 2")
```

It may seem like you're paying the cost on an unneeded comparison here, but in practice we find the overhead of most assertions is small compared to the huge benefits they bring in helping us build correct programs. The benefit of such assertions become particularly apparent when people use clone-and-modify programming, where new code is based on existing code that has been adjusted — correctly or not — to suit a new purpose. Here, somebody has cloned and modified `modulus_three()` into a new function `modulus_four()`:

```
def modulus_four(n):
    r = n % 4
    if r == 0:
        print("Multiple of 4")
    elif r == 1:
        print("Remainder 1")
    else:
        assert r == 2, "Remainder is not 2"
        print("Remainder 2")
```

Can you see the mistake? For some inputs the assertion is violated:

```
>>> modulus_four(4)
Multiple of 4
>>> modulus_four(3)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "examples/exceptions/modulus.py", line 20, in modulus_four
    assert r == 2, "Remainder is not 2"
AssertionError: Remainder is not 2
```

The assertion allows us to identify the problems and correct the program:

```python
def modulus_four(n):
    r = n % 4
    if r == 0:
        print("Multiple of 4")
    elif r == 1:
        print("Remainder 1")
    elif r == 2:
        print("Remainder 2")
    else:
        assert r == 3, "Remainder is not 3"
        print("Remainder 3")
```

An alternative formulation of this construct might be:

```python
def modulus_four(n):
    r = n % 4
    if r == 0:
        print("Multiple of 4")
    elif r == 1:
        print("Remainder 1")
    elif r == 2:
        print("Remainder 2")
    elif r == 3:
        print("Remainder 3")
    else:
        assert False, "This should never happen"
```

This would be perfectly acceptable, and perhaps even preferable, because the symmetry of the other cases makes it easier to spot blunders. Notice that the assertion is not used to validate the arguments to the function, only to detect if the implementation of the is incorrect. For function argument validation you should prefer to raise `ValueError`.

Class Invariants

Recall the `SortedSet` implementation we developed in Chapter 9[98] which used a sorted list of items as the internal representation. All methods in that class assume that the list is indeed sorted and remains that way. It would be wise to encode this class invariant as assertions at the beginning of every method, especially if you perform the exercise of making a *mutable* version of this collection type.

[98] chapter-9

For example, the `index()` method of that class assumes the items are already sorted because it uses a binary search. And the `count()` method depends on there being no duplicates because it performs a simple membership test. We can assert on these assumptions being `True` with a helper method:

```
class SortedSet(Sequence, Set):

  # ...

  def _is_unique_and_sorted(self):
      return all(self[i] < self[i + 1] for i in range(len(self) - 1))

  def index(self, item):
      assert self._is_unique_and_sorted()
      index = bisect_left(self._items, item)
      if (index != len(self._items)) and self._items[index] == item:
          return index
      raise ValueError("{} not found".format(repr(item)))

  def count(self, item):
      assert self._is_unique_and_sorted()
      return int(item in self._items)
```

Performance

The precondition assertions we perform to test our sorted set assumptions are relatively expensive. In the case that the class is working as expected, each test walks through the whole collection. Worse from a performance standpoint is that sometimes the test is performed more than once. See that `count()` is implemented in terms of `__contains__()`, and `__contains__()` is in turn implemented in terms of `index()`, so calling `count()` affirms our assumption twice. This is not necessarily a bad thing, but it can be detrimental to performance. Beyond trying to keep assertions both effective and cheap to evaluate, we have the option to run Python programs will all assertions disabled, using the `-O` flag on the command line:

```
$ python3 -m timeit -n 1 -s "from random import randrange; from sorted_set import Sor\
tedSet; s = SortedSet(randrange(1000) for _ in range(2000))"          "[s.count(i) fo\
r i in range(1000)]"
1 loops, best of 3: 2.79 sec per loop

$ python3 -O -m timeit -n 1 -s "from random import randrange; from sorted_set import \
SortedSet; s = SortedSet(randrange(1000) for _ in range(2000))"        "[s.count(i) f\
or i in range(1000)]"
1 loops, best of 3: 2.94 msec per loop
```

Notice that the result with `-O` is in *milli* seconds, so it roughly 1000 times faster than the version where assertions are enabled.

We encourage you to use this option only if performance concerns demand it. Running with assertions enabled in production can be a fabulous way of flushing out problems in your code. The majority of Python code is run will assertions *enabled* for this reason.

Avoiding side effects

Because `assert` statements can be removed from a program with a simple command line flag, it is crucial that the presence or absence of assertions does not affect the correctness of the program. For this reason you must avoid assertion conditions which have side effects. For example, never do something like:

```
assert my_list.pop(item)
```

Preconditions, postconditions and assertions

Assertions can be used to enforce function post-conditions. That is, they can assure us that a function is returning what we think it is returning. Consider this function, which we've placed in a module `wrapper.py`, which is used to wrap strings of text at a specified line length:

```
# wrapper.py

def wrap(text, line_length):
    """Wrap a string to a specified line length.

    Args:
        text: The string to wrap.
        line_length: The line length in characters.

    Returns:
        A wrapped string.
    """
    words = text.split()
    lines_of_words = []
    current_line_length = line_length
    for word in words:
        if current_line_length + len(word) > line_length:
            lines_of_words.append([])  # new line
            current_line_length = 0
        lines_of_words[-1].append(word)
        current_line_length += len(word)
    lines = [' '.join(line_of_words) for line_of_words in lines_of_words]
    return '\n'.join(lines)
```

The function is fairly complex, uses lots of mutable state, and — in the words of Turing-award recipient Sir Tony Hoare[99] — this is not code in which we could claim there are "obviously no deficiencies", although it may be true that there are "no obvious deficiencies"! [100]

Let's define some text:

[99] https://en.wikipedia.org/wiki/Tony_Hoare

[100] From his Turing Award Lecture: "I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

```
wealth_of_nations = "The annual labour of every nation is the fund which or" \
"iginally supplies it with all the necessaries and conveniencies of life wh" \
"ich it annually consumes, and which consist always either in the immediate" \
" produce of that labour, or in what is purchased with that produce from ot" \
"her nations. According, therefore, as this produce, or what is purchased w" \
"ith it, bears a greater or smaller proportion to the number of those who a" \
"re to consume it, the nation will be better or worse supplied with all the" \
" necessaries and conveniencies for which it has occasion."
```

Let's then give our function a whirl:

```
>>> from wrapper import *
>>> wrap(wealth_of_nations, 25)
'The annual labour of every\nnation is the fund which\noriginally supp
lies it with\nall the necessaries and\nconveniencies of life which\nit
 annually consumes, and\nwhich consist always either\nin the immediate
 produce of\nthat labour, or in what is\npurchased with that produce\n
from other nations.\nAccording, therefore, as\nthis produce, or what i
s\npurchased with it, bears a\ngreater or smaller\nproportion to the n
umber of\nthose who are to consume it,\nthe nation will be better or\n
worse supplied with all the\nnecessaries and\nconveniencies for which
it\nhas occasion.'
```

Well, it appears to have returned a single string with embedded line endings. Let's print that string to get the result rendered in a more helpful way:

```
>>> print(_)
The annual labour of every
nation is the fund which
originally supplies it with
all the necessaries and
conveniencies of life which
it annually consumes, and
which consist always either
in the immediate produce of
that labour, or in what is
purchased with that produce
from other nations.
According, therefore, as
this produce, or what is
purchased with it, bears a
```

```
greater or smaller
proportion to the number of
those who are to consume it,
the nation will be better or
worse supplied with all the
necessaries and
conveniencies for which it
has occasion.
```

Everything appears to work, but without counting the length of each line, it’s difficult to be sure. And in any case, it’s hard not to get distracted by Adam Smith’s spelling of “conveniences”.

Adding assertions for conditions

Let’s make sure our conditions are met by adding an assertion prior to the return of the final result. Our assertion takes the string we’re about to return, splits it back up into lines using `str.splitlines()`, and checks that the length of each line is less than or equal to the specified line length. We use the built-in `all()` function to check this is true for all lines:

```
def wrap(text, line_length):
    """Wrap a string to a specified line length.

    Args:
        text: The string to wrap.
        line_length: The line length in characters.

    Returns:
        A wrapped string.
    """
    words = text.split()
    lines_of_words = []
    current_line_length = line_length
    for word in words:
        if current_line_length + len(word) > line_length:
            lines_of_words.append([])  # new line
            current_line_length = 0
        lines_of_words[-1].append(word)
        current_line_length += len(word)
    lines = [' '.join(line_of_words) for line_of_words in lines_of_words]
    result = '\n'.join(lines)
    assert all(len(line) <= line_length for line in result.splitlines())
    return result
```

Neat!

Let's reload the code, and try again:

```
>>> from wrapper import *
>>> wrap(wealth_of_nations, 25)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "examples/exceptions/wrapper.py", line 24, in wrap
    assert all(len(line) <= line_length for line in result.splitlines())
AssertionError
```

Ouch! Our assertion failed, so our function isn't working as expected.

Fixing defects

The problem is that each time we add a word, we account for the length of the word in `current_line_length`, but we don't account for the space that will follow it when we join all the words back together using the `' '.join(line_of_words)` expression later in the program.

The fix is to account for that length when we increase the value of `current_line_length`. The simplest approach is just to do this:

```
current_line_length += len(word) + 1
```

But that introduces a so-called 'magic number' into our code. Why is that one there? Only a careful reading of the code by future maintainers will reveal why. Better, we think, to be explicit about where the one comes from: it's the length of the space following the word. So we'll do this:

```
current_line_length += len(word) + len(' ')
```

Rerunning with the fix in place, we can at least be sure that our function now meets its most basic requirement:

```
>>> from wrapper import *
>>> wrap(wealth_of_nations, 25)
'The annual labour of\nevery nation is the fund\nwhich originally supp
lies\nit with all the\nnecessaries and\nconveniencies of life\nwhich i
t annually\nconsumes, and which\nconsist always either in\nthe immedia
te produce of\nthat labour, or in what\nis purchased with that\nproduc
e from other\nnations. According,\ntherefore, as this\nproduce, or wha
t is\npurchased with it, bears\na greater or smaller\nproportion to th
e number\nof those who are to\nconsume it, the nation\nwill be better
or worse\nsupplied with all the\nnecessaries and\nconveniencies for wh
ich\nit has occasion.'
```

Programmer errors versus client errors

Our `wrap()` function isn't fully robust yet, though. What happens when we pass a negative line length:

```
>>> wrap(wealth_of_nations, -25)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "examples/exceptions/wrapper.py", line 24, in wrap
    assert all(len(line) <= line_length for line in result.splitlines())
AssertionError
```

In this case we also get an assertion error — from the same assertion — because the algorithm has no way to build lines of negative length. In a way, this is good: We found the problem. In other ways, this is bad because the assertion is intended to check that the lines are too long, not that `line_length` is too short. This conceptual mismatch would have been more apparent had we added a helpful message to the assertion statement:

```
assert all(len(line) <= line_length for line in result.splitlines()), "Line too long"
```

But a deeper conceptual problem exists here, which is that assertions are intended to detect *our* mistakes as implementers of this function, not the *clients* mistakes as callers of the function. The client is clearly at fault here, for passing a negative number to our function, and we should tell them so. If they were to see this assertion failure they would assume, with reasonable justification, that they had uncovered a defect in our code, rather than a problem with their code.

For this reason, it's *inappropriate* to use assertions to validate arguments provided by code beyond our immediate control. In other words, don't be tempted to use assertions as validation guards like this:

```python
def wrap(text, line_length):
    """Wrap a string to a specified line length.

    Args:
        text: The string to wrap.
        line_length: The line length in characters.

    Returns:
        A wrapped string.
    """
    assert line_length > 0, "line_length must be positive"
    words = text.split()
    lines_of_words = []
    current_line_length = line_length
    for word in words:
        if current_line_length + len(word) > line_length:
            lines_of_words.append([])  # new line
            current_line_length = 0
        lines_of_words[-1].append(word)
        current_line_length += len(word) + len(' ')
    lines = [' '.join(line_of_words) for line_of_words in lines_of_words]
    result = '\n'.join(lines)
    assert all(len(line) <= line_length for line in result.splitlines()), "Line too l\
ong"
    return result
```

Instead, prefer to raise a specific exception and document it appropriately:

```
def wrap(text, line_length):
    """Wrap a string to a specified line length.

    Args:
        text: The string to wrap.
        line_length: The line length in characters.

    Returns:
        A wrapped string.

    Raises:
        ValueError: If line_length is not positive.
    """
    if line_length < 1:
        raise ValueError("line_length {} is not positive".format(line_length))

    words = text.split()
    lines_of_words = []
    current_line_length = line_length
    for word in words:
        if current_line_length + len(word) > line_length:
            lines_of_words.append([])  # new line
            current_line_length = 0
        lines_of_words[-1].append(word)
        current_line_length += len(word) + len(' ')
    lines = [' '.join(line_of_words) for line_of_words in lines_of_words]
    result = '\n'.join(lines)
    assert all(len(line) <= line_length for line in result.splitlines()), "Line too l\
ong"
    return result
```

So now we get a reasonable — and predictable — error:

```
>>> from wrapper import *
>>> wrap(wealth_of_nations, -25)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "examples/exceptions/wrapper.py", line 17, in wrap
    raise ValueError("line_length {} is not positive".format(line_length))
ValueError: line_length -25 is not positive
```

Much better!

Reasonable relaxation of constraints

There's still an interesting case our code doesn't handle though. What should we do if the text contains a word longer than our line length?:

```
>>> from wrapper import *
>>> wrap('The next train to Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoc\
h is at 16:32', 25)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "examples/exceptions/wrapper.py", line 30, in wrap
    assert all(len(line) <= line_length for line in result.splitlines()), "Line too l\
ong"
AssertionError: Line too long
```

First we have to decide what is reasonable in this case, and we have a few options. Either we:

- Weaken our requirement and produce overly long lines
- Reject text containing words longer than the line length
- Split words over multiple lines

In our case, we'll take the easy way out of rejecting overly long text by raising an exception.

```
def wrap(text, line_length):
    """Wrap a string to a specified line length.

    Args:
        text: The string to wrap.
        line_length: The line length in characters.

    Returns:
        A wrapped string.

    Raises:
        ValueError: If line_length is not positive.
    """
    if line_length < 1:
        raise ValueError("line_length {} is not positive".format(line_length))

    words = text.split()
    if (max(map(len, words))) > line_length:
        raise ValueError("line_length must be at least as long as the longest word")

    lines_of_words = []
    current_line_length = line_length
    for word in words:
        if current_line_length + len(word) > line_length:
            lines_of_words.append([])  # new line
            current_line_length = 0
        lines_of_words[-1].append(word)
        current_line_length += len(word) + len(' ')
    lines = [' '.join(line_of_words) for line_of_words in lines_of_words]
    result = '\n'.join(lines)
    assert all(len(line) <= line_length for line in result.splitlines()), "Line too l\
ong"
    return result
```

It's good practice to polish your Python skills by trying to modify this function to use one of the other strategies for overly long words that we mentioned, although we recommend avoiding getting bogged down in the hyphenation rules for Welsh![101]

Summary

In this chapter we covered a lot of information related to exceptions and assertions:

- Exceptions
  - Demonstrated the dangers of handling all exceptions, especially so called system-exit exceptions such as `KeyboardInterrupt`.
  - Looked at the standard built-in exception hierarchy
  - Showed how catching base class exceptions can be used to catch multiple related exception types.
  - Informed you that you should almost never catch the `BaseException` or `Exception` types since they have subclasses which are almost always programming errors, such as `IndentationError`.
  - Investigated exception payloads, and how to use them effectively.
  - Showed how to define your own exception classes, by inheriting from `Exception`.

[101] Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch is the longest place name in Europe.

  - Showed that at it's most basic an `Exception` subclass need only contain a `pass` statement.
  - Demonstrated how to implement rich attributes on custom exceptions.
  - Illustrated exception chaining for both:
    - *Implicitly* chained exceptions, which set the `__context__` attribute on successor exception objects.
    - *Explicitly* chained exceptions, which set the `__cause__` attribute on successor exception objects.
- Tracebacks
  - Showed how to extract traceback objects from the `__traceback__` attribute of exceptions.
  - Counselled you to render tracebacks to strings rather than keeping references to them, to avoid space-leak problems with large object graphs reachable from the traceback instance.
  - Showed some key features of the standard library `traceback` module.
- Looked at assertions, including not only how to use them, but *when* to use them.

Chapter 11 - Defining Context Managers

In Chapter 9 of *The Python Apprentice*[102] we took our first look at Python's *with-blocks*. We only looked at how to use them with `close()`-able objects, such as files, but in fact with-blocks are a powerful tool for managing all sorts of resources and object lifetimes.

In this chapter we'll be focusing on *context managers*, those objects designed to be used as the argument to a *with-statement*. We'll also look at some advanced syntax for with-statements, as well as some tools in the standard library that help simplify context manager development.

> We differentiate between the terms *with-block* and *with-statement*. A with-statment is a statement of the form `with EXPR as NAME:` while a with-block is a with-statement plus an associated code block. This distinction is important because the with-statement is where Python directly interacts with context managers, while the entire with-block defines the lifetime of the context manager.

What is a context manager?

To begin with, what exactly is a *context manager*? A context manager is an object that is designed to be used in a with-statement. Remember that a with-statement has the following syntax:

```
with EXPR [as NAME]:
```

When a with-statement is executed, the `EXPR` part of the statement — that is, the expression following the `with` keyword — evaluates to a value. This value must be a context manager, and the underlying mechanics of the with-statement use this value in specific ways to implement the semantics of the with-statement.

[102] https://leanpub.com/python-apprentice

All of that is true and important, but perhaps it doesn't get to the heart of what a context manager is. Conceptually a context manager implements two methods which are used by the with-statement. The first method is called before the with-statement's code-block begins, and the second method is called after execution of the code in the with-block finishes, whether by naturally running to completion, returning early from the enclosing function, or by the raising of an exception which is allowed to propagate out of the with-block.

In other words, *a context manager represents code that runs before and after the with-statement's code-block*. You can think of these operations as set-up and tear-down, construction and destruction, resource allocation and deallocation, or any number of other ways.

In general — and for reasons that will become clear soon — we'll refer to these methods as *enter* and *exit*. The important point is that both the enter and exit methods are called every time the with-statement is executed, no matter how the code-block terminates.

So this brings us to perhaps the most useful statement of what a context manager is: It is a way to ensure that resources are properly and automatically managed around the code that uses these resources. The *enter* method of a context-manager ensures that the object is ready for use in the with-block, and the *exit* method ensures that the object is properly closed, shut down, or cleaned up, however execution of the block ends.

Files as context managers

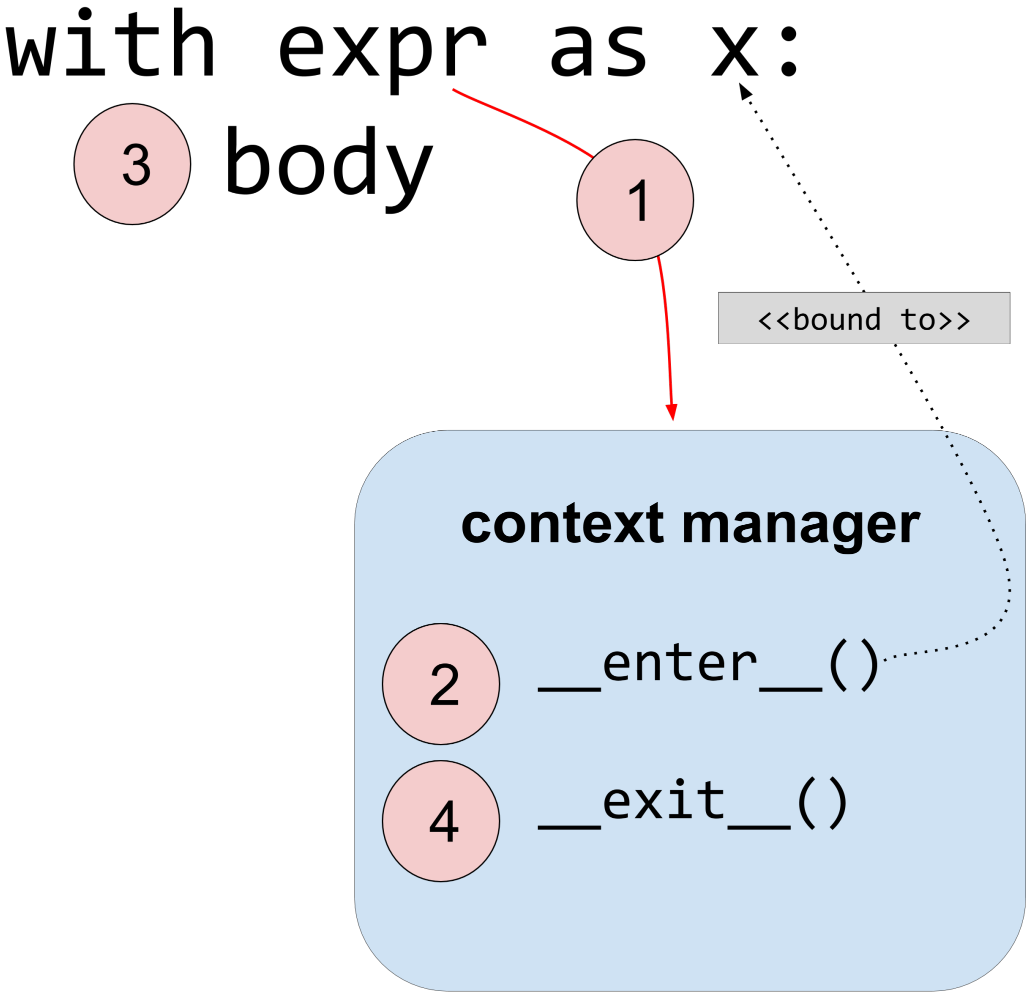
Since you've almost certainly used with-statements at some point, you've almost certainly used a context-manager whether you knew it or not. Almost every Python developer will have used files in a with-statement like this:

```
with open('important_data.txt', 'wt') as f:
    f.write('The secret password is 12345')
```

The benefit of using files in a with-statement is that they are automatically closed at the end of the with-block. This works because file objects are context managers. That is, files have methods which are called by the with-statement before the block is entered, and after the block exits. The *exit* method for a file (the code executed after the with-block exits) does the work of closing the file, and this is how files work with with-statements to ensure proper resource management.

The context manager protocol

Now that we understand what a context-manager is supposed to do, we can take a look at how to implement them. For an object to be a context-manager, it needs to support the *context-manager protocol* which consists of two methods, `__enter__()` and `__exit__()`. We'll look at the details of both of these methods, but first let's get a good idea of how these methods are used by a with-statement.



Context managers: step by step

Here’s what’s going on in this picture:

1. A with-statement evaluates its expression (the code immediately following the `with` keyword). The expression must evaluate to a context-manager. That is, the expression must produce an object which supports both the `__enter__()` and `__exit__()` methods.
2. The with-statement calls `__enter__()` on that context-manager with no arguments.

If `__enter__()` raises an exception, execution never enters the with-block, and the with-statement is done.

Assuming that `__enter__()` executes successfully, it can return a value. If the with-statement includes an as-clause, the return value of `__enter__()` is bound to the name in the as-clause; otherwise, this return value is discarded.

> It's important to really recognize what's going on here. A naïve interpretation of a with-statement might lead you to think that the result of the with-statement's expression is bound to the as-variable, while in fact it is the *return value of the context-manager's `__enter__()` that is bound to the as-variable.* In many cases, in fact, `__enter__()` simply returns the context-manager itself, which is the `self` argument to the `__enter__()` method, but this is not always the case, so it's important to keep in mind what's really happening.

3. The with-statement executes the body of the with-block. The with-block can terminate in one of two fundamental ways: Either with an exception or by running off the end of the block, which we call normal termination.
4. No matter how the with-block is terminated, Python then executes the context-manager's `__exit__()` method. If the block exits normally, `__exit__()` is called with no extra information. If, on the other hand, the block exits exceptionally, then the exception information is passed to `__exit__()`. This means that `__exit__()` can perform different actions based on how the with-block terminates, and this can be a very powerful tool.

A first example

With that in mind, let's make our first simple context manager. As with many things in Python, it's much easier to demonstrate context managers by example rather than by words alone.

Start by creating the simplest possible context manager like this:

```
class LoggingContextManager:
    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        return
```

As you can see, this implements both `__enter__()` and `__exit__()`, but neither one really does anything.

Let's try that out in a with-statement:

```
>>> with LoggingContextManager() as x:
...     print(x)
...
<__main__.LoggingContextManager object at 0x102b9b8d0>
```

Python didn't complain about anything, so it appears that we've created a valid — if somewhat useless — context manager. In the with-statement `x` is bound to our `LoggingContextManager`, which is what we'd expect, since we return `self` from `__enter__()`.

To demonstrate that the as-variable is actually bound to the return value of `__enter__()` rather than just the result of the expression, let's change `__enter__()` to return something else:

```
class LoggingContextManager:
    def __enter__(self):
        return "You're in a with-block!"

    def __exit__(self, exc_type, exc_val, exc_tb):
        return
```

Now if we use this context manager we'll see that the value of `x` is the string returned by `__enter__()`:

```
>>> with LoggingContextManager() as x:
...     print(x)
...
You're in a with-block!
```

We seem to know what we're doing! Let's now finish our context manager by having it log some text when we enter and exit it:

```
class LoggingContextManager:
    def __enter__(self):
        print('LoggingContextManager.__enter__()')
        return "You're in a with-block!"

    def __exit__(self, exc_type, exc_val, exc_tb):
        print('LoggingContextManager.__exit__({}, {}, {})'.format(
              exc_type, exc_val, exc_tb))
        return
```

Notice how our `__exit__()` method prints information about the exception information it receives. We'll look at these arguments in more detail later, but for now just know that these arguments are all `None` if the block exits normally, and that they contain other information if they block exits exceptionally.

Let's test out our fully armed and operational context manager:

```
>>> with LoggingContextManager() as x:
...     print(x)
...
LoggingContextManager.__enter__()
You're in a with-block!
LoggingContextManager.__exit__(None, None, None)
```

Exactly as we'd expect, we see the output from the with-block sandwiched between the strings printed by the context-manager's enter and exit functions.

To cap this example off, let's raise an exception from the with-block and see that, indeed, our `__exit__()` method is still called, this time with exception information:

```
>>> with LoggingContextManager() as x:
...     raise ValueError("Something has gone wrong!")
...     print(x)
...
LoggingContextManager.__enter__()
LoggingContextManager.__exit__(<class 'ValueError'>, Something has gone wrong!, <trac\
eback object at 0x102b9add0>)
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: Something has gone wrong!
```

We see the output from our `__enter__()` method, and we also see that the output from `__exit__()` indicates that an exception occurred.

You'll notice that the exception information is printed a second time as well, after our `__exit__()` method. This happens because we're letting the exception propagate out of the with-statement, so the REPL is printing its own information about the exception. There are ways to prevent exception propagation out of with-statements, and we'll look at those in a bit.

`__enter__()`

Now that you've created a complete context-manager yourself, there's not much left to say about the `__enter__()` method. It's called on the context-manager just before entering the with-block, and its return value is bound to the as-variable of the with-statement.

`__enter__()` is allowed to return anything it wants, including `None`, and the with-statement itself doesn't ever access or use this value. It is very common, however, for context-managers to simply return themselves from `__enter__()`.

This is the case, for example, with Python's `file` class. Consider code like this:

```
with open('important_data.txt', 'r') as f:
    data = f.read()
```

Here we know that `open()` returns a file object[103], and we also expect `f` to be bound to that file object, so clearly file's `__enter__()` method must be returning the file object itself.

We can verify this with a simple experiment in the REPL:

[103] https://docs.python.org/3/library/functions.html#open

```
>>> f = open('a_file', 'w')
>>> with f as g:
...     print(f is g)
...
True
```

We first open a file without using a with-statement, binding the file to the name `f`. We then use `f` as a context-manager in a with-statement, binding the result of `__enter__()` to the name `g`, and we see that `f` and `g` are the same object.

`__exit__()`

The `__exit__()` method is substantially more complex than `__enter__()` because it performs several roles in the execution of a with-statement. It's first and primary role, of course, is that it is executed after then with-block terminates, so it is responsible for cleaning up whatever resources the context-manager controls.

It's second role is to properly handle the case where the with-block exits with an exception. To handle this case, `__exit__()` accepts three arguments: The type of the exception that was raised, the value of the exception (that is, the actual exception instance), and the traceback associated with the exception. When a with-block exits without an exception, all three of these arguments are set to `None`, but when it exits exceptionally these arguments are bound to the exception which terminated the block.

In many cases a context-manager needs to perform different `__exit__()` code depending on whether an exception was raised or not, so it's typical for `__exit__()` to first check the exception information before doing anything else. A common idiom is to simply check the `type` argument to detect an exceptional exit.

Let's update `LoggingContextManager.__exit__()` to behave differently when an exception is raised:

```
class LoggingContextManager:
    . . .
    def __exit__(self, exc_type, exc_val, exc_tb):
        if exc_type is None:
            print('LoggingContextManager.__exit__: '
                  'normal exit detected')
        else:
            print('LoggingContextManager.__exit__: '
                  'Exception detected! '
                  'type={}, value={}, traceback={}'.format(
                      exc_type, exc_val, exc_tb))
```

The revised implementation of `__exit__()` first checks whether `type` is `None`. If it is, then this means that no exception was raised and a simple message is printed.

If `type` is not `None`, however, `__exit__()` prints a longer message that includes the exception information. Let's test this out in the REPL:

```
>>> with LoggingContextManager():
...     pass
...
LoggingContextManager.__enter__()
LoggingContextManager.__exit__: normal exit detected
```

Here we see that normal exits are handled properly. Now let's raise an exception from the with-block:

```
>>> with LoggingContextManager():
...     raise ValueError('Core meltdown imminent!')
...
LoggingContextManager.__enter__()
LoggingContextManager.__exit__: Exception detected! type=<class 'ValueError'>, value=\
Core meltdown imminent!, traceback=<traceback object at 0x102ba6128>
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: Core meltdown imminent!
```

We now see that `LoggingContextManager` properly detects exceptions.

Controlling exception propagation

There's a final role that `__exit__()` plays in with-statements that has to do with how exceptions are propagated out of them. By default, when an exception is raised from a with-block, the context-manager's `__exit__()` is executed and, afterward, the original exception is re-raised. We've seen this in our examples when the REPL prints exception information after our context-manager is done.

To really drive this point home, let's try another small experiment in the REPL:

```
>>> try:
...     with LoggingContextManager():
...             raise ValueError('The system is down!')
... except ValueError:
...     print('*** ValueError detected ***')
...
LoggingContextManager.__enter__()
LoggingContextManager.__exit__: Exception detected! type=<class 'ValueError'>, value=\
The system is down!, traceback=<traceback object at 0x102ba60e0>
*** ValueError detected ***
```

Here we use a with-statement inside a try-block and catch the `ValueError` which is propagated out of the with-statement.

So how does `__exit__()` control the propagation of exceptions out of with-statements?

It has to do with the return value of the `__exit__()` method. If `__exit__()` returns a value which evaluates to `False` in a boolean context, then any exception that came out of the with-block will be re-raised after the `__exit__()` call.

> Essentially the with-statement is asking `__exit__()` "should I suppress the exception?" If `__exit__()` returns `False`, then the with-statement re-raises the exception accordingly. If `__exit__()` returns `True`, then the with-statement will exit normally, that is, without raising the exception.

With that in mind, remember from earlier that the default behavior of a with-statement is to propagate exceptions. What this really means is that, if your context-manager's `__exit__()` function doesn't return anything, then exceptions would be propagated. This is because, as you'll recall, a method that doesn't explicitly return a value will implicitly return `None`.[^function-return-none] Of course, `None` evaluates to `False`, so an `__exit__()` which doesn't explicitly return anything is instructing the calling with-statement to allow exceptions to propagate.

Raising exceptions from `__exit__()`

Another important point to understand is that `__exit__()` should not explicitly re-raise the exception that it receives. If `__exit__()` wants to ensure that the exception from the with-block is re-raised, it should simply return `False` and let the with-statement re-raise it.

Furthermore, `__exit__()` should only explicitly raise exceptions when it actually fails itself, that is, when something goes wrong in the `__exit__()` method. The with-statement machinery will interpret exceptions from `__exit__()` as a failure of `__exit__()`, not as a propagation of the original exception.

The expansion of a with-statement

Before we leave this section on creating context managers, let's take a second and look at exactly what Python is doing in with-statements. The with-statement was defined in PEP343[104], and in that PEP the authors provide an explicit expansion of the with-statement into Python code that doesn't use the with-statement.

It looks like this:

```
mgr = (EXPR)
exit = type(mgr).__exit__  # Not calling it yet
value = type(mgr).__enter__(mgr)
exc = True
try:
    try:
        VAR = value  # Only if "as VAR" is present
        BLOCK
    except:
        # The exceptional case is handled here
        exc = False
        if not exit(mgr, *sys.exc_info()):
            raise
        # The exception is swallowed if exit() returns true
```

[104] https://www.python.org/dev/peps/pep-0343/

```
finally:
    # The normal and non-local-goto cases are handled here
    if exc:
        exit(mgr, None, None, None)
```

We won't spend any time going over this expansion, but if you want to really understand how context-managers are used by with-statements, then it's worth taking some time on your own to really understand this code. It may seems like a lot to ingest at first, but it's really not saying anything more than what we've already covered in this chapter, and seeing it written so explicitly might help clear up any lingering questions you have.

And in any case, it's just interesting to see how these kinds of language features are developed!

`contextlib.contextmanager`

Now that we've covered the details of how to create context-managers, let's spend some time looking at a tool that helps simplify context-manager development. The `contextlib` package[105] is part of Python's standard library. In it's own words, it "provides utilities for common tasks involving the `with` statement." In this book we're just going to look at one part of `contextlib`, the `contextmanager` decorator. There are several other interesting parts to `contextlib`, however, and it's well worth your time to investigate them.

The `contextmanager` decorator is, as its name suggests, a decorator that you can use to create new context managers. The concept behind `contextmanager` is simple. You define a generator function — that is, a function which uses `yield` instead of `return` — and decorate it with the `contextmanager` decorator to create a context-manager factory. This factory is nothing more than a callable object which returns context-managers, making it suitable for use in a with-statement.

Let's see how this looks in code since that will make everything much more clear:

[105] https://docs.python.org/3/library/contextlib.html

```
@contextlib.contextmanager
def my_context_manager():
    # <ENTER>
    try:
        yield [value]
        # <NORMAL EXIT>
    except:
        # <EXCEPTIONAL EXIT>
        raise
```

Here we see the `contextmanager` decorator applied to a generator called `my_context_manager`.

You can now use `my_context_manager` just like any other context-manager:

```
with my_context_manager() as x:
    # . . .
```

Here's how it functions. First the generator function is executed up to its yield statement. Everything before the `yield` is equivalent to the `__enter__()` method on a normal context-manager. Next, the `yield` statement supplies the value which will be bound to any name given in the as-clause of the with-statement. In other words the `yield` is like the return value from `__enter__()` in a normal context-manager definition.

Once `yield` is called, control leaves the context-manager function and goes to the with-block. If the with-block terminates normally, then execution flow returns to the context-manager function immediately after the `yield` statement. In our code snippet, this is the section marked `<NORMAL EXIT>`. If, on the other hand, the with-block raises an exception, then that exception is re-raised from the `yield` statement in the context-manager. In our code snippet, this means that execution would go to the `except` block and into the section labeled `<EXCEPTIONAL EXIT>`.

In other words, the `contextmanager` decorator allows you define context-managers control flow in a single generator function via the `yield` statement rather than breaking it up across two methods. Since generators remember their state between calls to `yield`, you don't need to define a new class to create a stateful context-manager.

Let's see if we can rewrite our old `LoggingContextManager` using `contextmanager`:

```
import contextlib
import sys


@contextlib.contextmanager
def logging_context_manager():
    print('logging_context_manager: enter')
    try:
        yield "You're in a with-block!"
        print('logging_context_manager: normal exit')
    except Exception:
        print('logging_context_manager: exceptional exit',
              sys.exc_info())
```

That certainly looks simpler than the original version, so let's see if it works:

```
>>> with logging_context_manager() as x:
...     print(x)
...
logging_context_manager: enter
You're in a with-block!
logging_context_manager: normal exit
```

That looks good for the case of a normal exit, so how about an exceptional exit:

```
>>> with logging_context_manager() as x:
...     raise ValueError('Something went wrong!')
...
logging_context_manager: enter
logging_context_manager: exceptional exit (<class 'ValueError'>, ValueError('Somethin\
g went wrong!',), <traceback object at 0x102c6f170>)
```

Great, our new context-manager works!

`contextmanager` and exception propagation

One thing you may have noticed is that our new context-manager didn't propagate the `ValueError` after it completed. Unlike standard context-managers, those created with the `contextmanager` decorator must use normal exception handling to determine if exceptions are propagated from the with-statement.

If the context-manager function propagates the exception, either via re-raising it or by simply not catching it at all, then the exception will propagate out of with-statement.

If the context-manager catches and doesn't re-raise an exception from the with-block, then the exception won't be propagated out of the with-statement.

In our case, we caught the `ValueError` and printed some information about it. Since we didn't re-raise it, the `ValueError` was not propagated out of our context-manager, and thus it wasn't propagated out of the with-statement.

Just to make this point explicit, let's update our new context-manager to propagate exceptions. To do this we simply add a bare `raise` call after logging the exception:

```
@contextlib.contextmanager
def logging_context_manager():
    print('logging_context_manager: enter')
    try:
        yield "You're in a with-block!"
        print('logging_context_manager: normal exit')
    except Exception:
        print('logging_context_manager: exceptional exit',
              sys.exc_info())
        raise
```

We can see that this works as we expected since the `ValueError` is propagated to the REPL:

```
>>> with logging_context_manager() as x:
...     raise ValueError('Something went wrong!')
...
logging_context_manager: enter
logging_context_manager: exceptional exit (<class 'ValueError'>, ValueError('Somethin\
g went wrong!',), <traceback object at 0x102c6f1b8>)
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: Something went wrong!
```

`contextlib.contextmanager` is a very useful tool, and it certainly eases the creation of context managers in many cases. It's good to know how to create context-managers using the lower-level protocols, of course, and knowing which technique to use in a given situation will require some experience and practice.

Multiple context-managers in a with-statement

So far when we've used with-statements, we've only used a single context-manager. But you can actually use as many context-managers as you want in a single with-statement. The syntax for this is like this:

```
with cm1() as a, cm2() as b, . . .:
   BODY
```

Each context-manager and optional variable binding is separated by a comma. From an execution point of view, this is exactly equivalent to using nested with-statements, with earlier context-managers "enclosing" later ones.

So this multi-context-manager form:

```
with cm1() as a, cm2() as b:
   BODY
```

is the same as this nested form:

```
with cm1() as a:
    with cm2() as b:
        BODY
```

Let's define a simple context-manager to test this out:

```
import contextlib


@contextlib.contextmanager
def nest_test(name):
    print('Entering', name)
    yield
    print('Exiting', name)
```

We'll pass two of these to a single with-statement:

```
>>> with nest_test('outer'), nest_test('inner'):
...     print('BODY')
...
Entering outer
Entering inner
BODY
Exiting inner
Exiting outer
```

We can see that this gives the same results as a nested form:

```
>>> with nest_test('outer'):
...     with nest_test('inner'):
...             print('BODY')
...
Entering outer
Entering inner
BODY
Exiting inner
Exiting outer
```

Later context-managers are simply part of their with-block's body as far as earlier context-managers are concerned. In fact, names bound in an as-clause in earlier parts of the with-statement can be used when defining later context-managers.

Let's modify `nest_test` to yield a value:

```
@contextlib.contextmanager
def nest_test(name):
    print('Entering', name)
    yield name
    print('Exiting', name)
```

We'll then use that yielded value when constructing the second context-manager:

```
>>> with nest_test('outer') as n1, nest_test('inner, nested in ' + n1):
...     print('BODY')
...
Entering outer
Entering inner, nested in outer
BODY
Exiting inner, nested in outer
Exiting outer
```

Again, this is precisely the behavior you would get if you simply used nested with-statements.

Nested with-statements and exceptions

With the nested with-statement notion in mind, it should be clear how exceptions are handled when using multiple context-managers. In short, any exception propagated from inner context-managers will be seen by outer context-managers. Likewise, if an inner context-manager swallows an exception, then it won't be seen by outer ones.

To illustrate, let's create one more simple context manager that can be configured to either propagate or swallow exceptions:

```
import contextlib


@contextlib.contextmanager
def propagater(name, propagate):
    try:
        yield
        print(name, 'exited normally.')
    except Exception:
        print(name, 'received an exception!')
        if propagate:
            raise
```

If the `propagate` argument to `propagater` is `True`, it will propagate any exceptions that come from the body nested beneath it. Otherwise, it will swallow those exceptions.

Now we can see how an inner context-manager can swallow exceptions so that an outer one never sees them:

```
>>> with propagater('outer', True), propagater('inner', False):
...     raise TypeError('Cannot convert lead into gold.')
...
inner received an exception!
outer exited normally.
```

Likewise, the inner one can propagate them while the outer one swallows them:

```
>>> with propagater('outer', False), propagater('inner', True):
...     raise TypeError('Cannot convert lead into gold.')
...
inner received an exception!
outer received an exception!
```

Since the REPL doesn't report an exception in this case, we can be sure that no exception escaped the with-statement.

Don't pass a collection!

When using multiple context-managers with a single with-statement, don't try to use a tuple or some other sequence of context-managers. This idea is seductive, because the comma used to separate multiple context managers looks like it's separating the members of a tuple which is missing its optional parentheses[106]; and if only we added those parentheses, we could leverage Python formatting rules which allow us to split collection literals over multiple lines!

Beware!

If you're drawn by the sirens of beautiful code in this direction, you'll find your dreams dashed upon the rocks of a mysterious-looking error message:

[106] For tuples with more than one element the delimiting parentheses are optional.

```
>>> with (nest_test('a'),
...       nest_test('b')):
...     pass
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: __exit__
```

At first glance it seems to be telling you that your context-manager doesn't have a `__exit__()` method, which we know is wrong. So what's going on?

The problem here is that we're actually trying to pass a *tuple* to the with-statement as a context-manager. The with-statement doesn't try to unpack sequences; it simply tries to use what you pass it as a context-manager. So the with-statement is looking for `__exit__()` on a the `tuple` you are passing it, of course, it fails with an `AttributeError` since `tuple`s do not support the context manager protocol.

To fix this, simply remove the square brackets and put everything back one line:

```
>>> with nest_test('a'), nest_test('b'):
...     pass
...
Entering a
Entering b
Exiting b
Exiting a
```

If you have several context-managers and you want to split them over several lines, you must use the line-continuation backslash to put the statements on one *logical* split over multiple *physical* lines.

For example:

```
>>> with nest_test('a'), \
...      nest_test('b'), \
...      nest_test('c'):
...     pass
...
Entering a
Entering b
Entering c
Exiting c
Exiting b
Exiting a
```

Context managers for transactions

As we've seen, developing context-managers is not difficult, and with the various tools we have for creating them we should not back away from using context-managers when they're appropriate. Most of the examples we've seen so far have been small, and they've focused on presenting a particular point. We'll use this section to develop a more realistic example, something like what you might see in the wild.

Our example will involve a `Connection` class which represents some sort of database connection, along with a `Transaction` class which manages transactions in the database. Users of our system can create `Connections` and then create `Transaction` objects to start transactions. To commit or rollback transactions, users can call methods on the `Transaction` instances.

Here's our `Connection` class:

```
class Connection:
    def __init__(self):
        self.xid = 0

    def _start_transaction(self):
        print('starting transaction', self.xid)
        rslt = self.xid
        self.xid = self.xid + 1
        return rslt

    def _commit_transaction(self, xid):
        print('committing transaction', xid)
    def _rollback_transaction(self, xid):
        print('rolling back transaction', xid)
```

Obviously this doesn't do *real* database work. All that it really does is increment a transaction-id whenever a new transaction is started. It also helpfully prints out what it's doing as transactions are processed.

Now here's our `Transaction` class:

```
class Transaction:
    def __init__(self, conn):
        self.conn = conn
        self.xid = conn._start_transaction()

    def commit(self):
        self.conn._commit_transaction(self.xid)

    def rollback(self):
        self.conn._rollback_transaction(self.xid)
```

This queries the `Connection` for a new transaction-id, and then calls into the `Connection` to commit or rollback transactions.

Here's how someone might use these classes without context-managers:

```
>>> conn = Connection()
>>> xact = Transaction(conn)
starting transaction 0
>>> xact.commit()
committing transaction 0
```

Here's how it might look in a function:

```
def some_important_operation(conn):
    xact = Transaction(conn)
    # Now the critical stuff:
    # Creating lots of business-critical, high-value stuff here.
    # These lines will probably open whole new markets.
    # And deliver hitherto unseen levels of shareholder value.
```

But of course, that last example is flawed because we never commit the transaction. Yet it's very easy to write this kind of code! And even if we *did* explicitly commit the transaction, it can be tricky to get it right in the presence of exceptions.

Managing transactions with context-managers

So let's see if we can't design a context-manager that starts a transaction for us, commits it if the with-block exits normally, but rolls it back if an exception occurs:

```
import contextlib


@contextlib.contextmanager
def start_transaction(connection):
    tx = Transaction(connection)

    try:
        yield tx
    except:
        tx.rollback()
        raise

    tx.commit()
```

This is very similar to the examples we've seen earlier. The beginning of the context-manager starts a transaction. Then, inside a try-block, we yield the transaction so that transactional operations can take place inside a with-block. Then, if the with-block raises an exception, we catch the exception, roll the transaction back, and re-raise the exception. If the with-block exits normally, we commit the transaction and continue normally.

So, let's see it in action!:

```
>>> conn = Connection()
>>> try:
...     with start_transaction(conn) as tx:
...         x = 1 + 1
...         raise ValueError()
...         y = x + 2
...         print('transaction 0 =', x, y)
... except ValueError:
...     print('Oops! Transaction 0 failed.')
...
starting transaction 0
rolling back transaction 0
Oops! Transaction 0 failed.
```

Great! The context manager detected the exception and rolled the transaction back.

Let's try that again, this time without the exception:

```
>>> try:
...     with start_transaction(conn) as tx:
...         x = 1 + 1
...         y = x + 2
...         print('transaction 1 =', x, y)
... except ValueError:
...     assert False
...
starting transaction 1
transaction 1 = 2 4
committing transaction 1
```

So now we've got a full-featured context manager that does some pretty realistic work. All it needs is a full database connected to it…but that's just an implementation detail, right?

Summary

In this chapter we've covered a lot of the details of context-manager development, and you should be comfortable enough with them to write your own. Here is a summary of the specific points we covered:

- The context manager protocol

  - Context managers are objects designed to work in with-statements
  - The expression of a with-statement must evaluate to a context-manager
  - Context managers have code that is run before and after with-blocks
  - Context managers are particularly useful for resource management
  - Files objects are a common used example of context-managers
  - The context manager protocol involves two methods, `__enter__()` and `__exit__()`
  - `__enter__()` is called before the with-block
  - The return value of `__enter__()` is bound to name in the optional as-clause of a with-statement
  - If `__enter__()` raises an exception, the with-block is never entered
  - `__enter__()` often returns its own `self`, but this is not required
  - A context-manager's `__exit__()` method is called when a with-block terminates
  - `__exit__()` is called both when the with-block exits normally and when it exits exceptionally
  - If a with-block exits with an exception, `__exit__()` is called with the type, value, and traceback of that exception as arguments
  - If a with-block exits without an exception, then `__exit__()` is called with `None` for each of those arguments
  - The return value of `file.__enter__()` is the file object itself
  - `__exit__()` can respond differently to exceptional and non-exceptional with-block exits by checking its arguments
  - If `__exit__()` returns a value that evaluates to `False`, then any exception that came from the with-block will be propagated by the with-statement
  - If `__exit__()` returns a value that evaluates to `True`, then any exception that came from the with-block will be swallowed by the with-statement
  - Since functions implicitly return `None` in the absence of an explicit return, with-statements will propagate exceptions when `__exit__()` doesn't explicitly return a value
  - `__exit__()` should not explicitly re-raise its exception arguments
  - `__exit__()` should only raise exceptions if the problem occurs in the method itself
  - PEP343 is the original PEP defining with-statements
  - PEP343 includes an expansion of with-statements into Python code that doesn't use with-statements
- `contextlib.contextmanager`
  - `contextlib` provides some utilities for working with with-statements

  - `contextlib.contextmanager` is a decorator used for creating context-manager factories out of generator functions
  - A `contextmanager` generator yields a value which will be bound to name in the optional as-clause
  - Execution moves from the `contextmanager` generator to the with-block when the `yield` statement is executed
  - Executions returns from the with-block to the `yield` statement when the with-block terminates
  - All of the code in a `contextmanager` generator *before* the `yield` is equivalent to the `__enter__()` method
  - If an exception is raised in the with-block, it is re-raised at the `yield` statement in the `contextmanager` generator
  - The code executed *after* the `yield` in a `contextmanager` generator is equivalent to the `__exit__()` method
  - If a `contextmanager` generator wants to propagate an exception, it needs to explicitly re-raise it
- Nested context-managers
  - A single with-statements can accept more than one context-manager
  - Separate each context-manager (including its optional as-clause) from the next with commas
  - Multiple context-managers in a with-statement are equivalent to nested with-statements
  - Later context-managers are nested inside earlier ones
  - Names bound in earlier context-managers can be used when creating later ones
  - With multiple context-managers, later context-managers can swallow exceptions such that earlier context-managers never see them
  - If you mistakenly pass a tuple or other sequence of context-managers to a with-statement, you may be surprised to get an `AttributeError`
  - To put context-managers on different lines in the same with-statement, use line-continuation backslashes

Chapter 12 - Introspection

Introspection is the ability of a program to examine its own structure and state; a process of looking inward to perform self-examination. We've already encountered many tools which allow programs to examine themselves, both in earlier chapters of this book and in *The Python Apprentice*[107]. We'll quickly review some of those tools here to bring them under the umbrella of introspection, and then will move on to more tools in this category, and some uses for them.

Introspecting types

Perhaps the simplest introspective tool we've met is the built-in function `type()`. We've used this extensively at the REPL for displaying object types:

```
>>> i = 7
>>> type(i)
<class 'int'>
```

Now just enter `int` into the REPL, and press Return:

```
>>> int
<class 'int'>
```

This indicates that `<class 'int'>` is just the representation of the type `int` as produced by `repr()` — which is what the REPL does when displaying the result of an expression. We can confirm this by evaluating this expression:

```
>>> repr(int)
"<class 'int'>"
```

So `type(7)` is actually returning `int`:

[107] https://leanpub.com/python-apprentice

```
>>> type(i) is int
True
```

We can even call the constructor on the returned type directly:

```
>>> type(i)(78)
78
```

Types of types

But what is the type of `int`? Or in other words, what does type does `type()` return?:

```
>>> type(type(i))
<class 'type'>
```

The `type()` of `type()` is `type`! Every object in Python has an associated `type` object, which is retrieved using the `type()` function.

In fact, the `type()` function just returns the special `__class__` attribute:

```
>>> i.__class__
<class 'int'>
>>> i.__class__.__class__
<class 'type'>
>>> i.__class__.__class__.__class__
<class 'type'>
```

This confirms that `type` is its own type.

Types are objects

Using another introspection facility we already know, we can see that `type` is itself an `object`:

```
>>> issubclass(type, object)
True
```

What is more, the `type()` of object is `type`!:

```
>>> type(object)
<class 'type'>
```

What this circular dependency shows is that both `type` and `object` are fundamental to the Python object model, and neither can stand alone without the other.

Notice that `issubclass()` performs introspection — it answers a question about an object in the program, as does the `isinstance()` method we are familiar with:

```
>>> isinstance(i, int)
True
```

In general, type tests should be avoided in Python, but on the rare occasions they are necessary, prefer to use the `isinstance()` or `issubclass()` functions rather than direct comparison of `type` objects.

Introspecting objects

We've already met another important introspection function, `dir()`, which returns a list of attribute names for an instance:

```
>>> a = 42
>>> dir(a)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
 '__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__',
 '__float__', '__floor__', '__floordiv__', '__format__', '__ge__',
 '__getattribute__', '__getnewargs__', '__gt__', '__hash__',
 '__index__', '__init__', '__int__', '__invert__', '__le__',
 '__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__',
 '__new__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__',
 '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__',
 '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__',
 '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
 '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__',
 '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__',
 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag',
 'numerator', 'real', 'to_bytes']
```

Both attributes and method names are returned in this list, for the simple reason that methods *are* attributes — they're just callable attributes.

`getattr()`

Given an object and an attribute name, we can retrieve the corresponding attribute object using the built-in `getattr()` function. It's fascinating to see that the `int` object has attributes called `numerator` and `denominator` allowing it to be used like the rational number object modelled by the `Fraction` type, and also `imag`, `real,` and `conjugate` attributes allowing it to used like a `complex` number under duck-typing rules. Let's retrieve one of the attributes using `getattr()`:

```
>>> getattr(a, 'denominator')
1
```

This returns the same value as accessing the denominator directly:

```
>>> a.denominator
1
```

Other attributes return more interesting objects:

```
>>> getattr(a, 'conjugate')
<built-in method conjugate of int object at 0x100233160>
```

The `conjugate` attribute is revealed to be a method. We can check that it is a callable object using another introspection tool, the built-in `callable()` function:

```
>>> callable(getattr(a, 'conjugate'))
True
```

By navigating around the attributes of an object, and the attributes of those attributes, it's possible to discover almost anything about an object in Python. Here we retrieve the type of the `int.conjugate` method through its `__class__` attribute, and then the `__name__` of that class as a string through the `__name__` attribute.

```
>>> i.conjugate.__class__.__name__
'builtin_function_or_method'
```

Trying to retrieve an attribute that does not exist will result in an `AttributeError`:

```
>>> getattr(a, 'index')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'int' object has no attribute 'index'
```

hasattr()

We can determine whether a particular object has an attribute of a given name using the built-in `hasattr()` function, which returns `True` if a particular attribute exists:

```
>>> hasattr(a, 'bit_length')
True
>>> hasattr(a, 'index')
False
```

In general, though, *Easier to Ask for Forgiveness than Permission* (EAFP) style programming using exception handlers is considered more Pythonic than *Look Before You Leap* (LBLY) style programs using type test and attribute existence tests. Programs using `hasattr()` can quickly get messy, particularly if you need to test for the existence of many different attributes. And perhaps surprisingly the optimistic code using `try..except` is faster than LBLY code using `hasattr()`, because internally `hasattr()` uses an exception handler anyway!

Here's a function in a module `numerals.py` which, given an object supporting the `numerator` and `denominator` attributes for rational numbers, returns a so-called mixed-numeral containing the separate whole number and fractional parts:

```
# numerals.py

from fractions import Fraction


def mixed_numeral(vulgar):
    if not (hasattr(vulgar, 'numerator') and hasattr(vulgar, 'denominator')):
        raise TypeError("{} is not a rational number".format(vulgar))

    integer = vulgar.numerator // vulgar.denominator
    fraction = Fraction(vulgar.numerator - integer * vulgar.denominator,
                        vulgar.denominator)
    return integer, fraction
```

This version uses `hasattr()` to check whether the supplied object supports the rational number interface. Here we call it with a `Fraction`:

```
>>> from numerals import mixed_numeral
>>> from fractions import Fraction
>>> mixed_numeral(Fraction('11/10'))
(1, Fraction(1, 10))
```

When we call it with a `float`, however, the type check fails:

```
>>> mixed_numeral(1.7)
Traceback (most recent call last):
  File "examples/eafp_vs_lbyl/numerals.py", line 5, in mixed_numeral
    raise TypeError("{} is not a rational number".format(vulgar))
  TypeError: 1.7 is not a rational number
```

In contrast, here's an alternative version which does not perform the `hasattr()` check:

```
from fractions import Fraction


def mixed_numeral(vulgar):
    integer = vulgar.numerator // vulgar.denominator
    fraction = Fraction(vulgar.numerator - integer * vulgar.denominator,
                        vulgar.denominator)
    return integer, fraction
```

The only difference in its runtime behaviour is that a different exception is raised:

```
>>> from fractions import Fraction
>>> from numerals import mixed_numeral
>>> mixed_numeral(Fraction('11/10'))
Out[5]: (1, Fraction(1, 10))
>>> mixed_numeral(1.7)
  File "examples/eafp_vs_lbyl/numerals.py", line 7, in mixed_numeral
    integer = vulgar.numerator // vulgar.denominator
AttributeError: 'float' object has no attribute 'numerator'
```

This exception is possibly more detailed, but certainly less informative. Of course, we can have the best of both worlds by using an exception handler to raise the more appropriate exception type of `TypeError` chaining it to the original `AttributeError` to provide the details:

```
from fractions import Fraction


def mixed_numeral(vulgar):
    try:
        integer = vulgar.numerator // vulgar.denominator
        fraction = Fraction(vulgar.numerator - integer * vulgar.denominator,
                            vulgar.denominator)

        return integer, fraction
    except AttributeError as e:
        raise TypeError("{} is not a rational number".format(vulgar)) from e
```

This approach yields the maximum amount of information about what went wrong and why:

```
>>> from fractions import Fraction
>>> from numerals import mixed_numeral
>>> mixed_numeral(Fraction('11/10'))
Out[5]: (1, Fraction(1, 10))
>>> mixed_numeral(1.7)
Traceback (most recent call last):
  File "examples/eafp_vs_lbyl/numerals.py", line 6, in mixed_numeral
    integer = vulgar.numerator // vulgar.denominator
AttributeError: 'float' object has no attribute 'numerator'

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "examples/eafp_vs_lbyl/numerals.py", line 12, in mixed_numeral
    raise TypeError("{} is not a rational number".format(vulgar)) from e
TypeError: 1.7 is not a rational number
```

The lesson here is that, if you find yourself using type introspection in Python, it might be better to avoid the checks and just “let it fail”.

Introspecting scopes

Python contains two built-in functions for examining the content of scopes. Let’s play with them at the REPL.

`globals()`

The first function we'll look at is `globals()`:

```
>>> globals()
{'__package__': None,
 '__builtins__': <module 'builtins' (built-in)>,
 '__name__': '__main__',
 '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
 '__doc__': None}
```

This returns a dictionary which represents the global namespace. For example, we can see in here the binding of the name `__name__` to the string `__main__`, which we've frequently used to determine how a program is being executed with the "main block" idiom:

```
if __name__ == '__main__':
    <main block>
```

Now we'll define a variable:

```
>>> a = 42
```

If we call `global()` again, we can see that the binding of the integer 42 to the name 'a' has been added to the namespace:

```
>>> globals()
{'__builtins__': <module 'builtins' (built-in)>,
 '__name__': '__main__',
 '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
 '__doc__': None,
 'a': 42}
```

`globals()` *is* the global namespace

In fact, the dictionary returned by `globals()` doesn't just represent the global namespace, it actually *is* the global namespace!

Let's create a variable by assigning to new keys in the dictionary:

```
>>> globals()['tau'] = 6.283185
```

We can now use this variable just like any other:

```
>>> tau
6.283185
>>> tau / 2
3.1415925
```

`locals()`

The second function for introspecting scopes in Python is `locals()`:

```
>>> locals()
{'__builtins__': <module 'builtins' (built-in)>,
 '__name__': '__main__',
 '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
 '__doc__': None,
 'a': 42,
 'tau': 6.283185,
 '__package__': None}
```

Since we're currently operating at a module — or in other words global — scope in the REPL, `locals()` returns the same dictionary as `globals()`.

To really see `locals()` in action, we're going to need to create another local scope, which we can do by defining a function:

```
>>> def report_scope(arg):
...     from pprint import pprint as pp
...     x = 496
...     pp(locals(), width=10)
...
>>> report_scope(42)
{'arg': 42,
 'pp': <function pprint at 0x1020dfef0>,
 'x': 496}
```

When run, see that this function has three entries in its local namespace: The function argument `arg`, a binding to the function imported from the pretty-printing standard library module, and the variable `x`.

Recall that extended function call syntax allows us to unpack a dictionary into function keyword arguments. Remember also that the string `format` method accepts named arguments which correspond to format placeholders. By using `locals()` to provide the dictionary, we can easily refer to local variables in format strings:

```
>>> name = "Joe Bloggs"
>>> age = 28
>>> country = "New Zealand"
>>>
>>> "{name} is {age} years old and is from {country}".format(**locals())
'Joe Bloggs is 28 years old and is from New Zealand'
```

Using this technique is less wise from a maintainability standpoint than it's cleverness might suggest, so use it sparingly. That said, you might come across this in the wild, and it can be a handy time-saver when debugging.

From Python 3.6 onwards, this trick is no longer needed, as Python 3.6 introduced support for f-strings which allow direct reference to in-scope names from within literal strings. Nonetheless, it's worth knowing this approach so you can recognise it in existing code.

The `inspect` module

The `inspect` module in the Python Standard Library contains advanced tools for introspecting Python objects in a great deal more detail. We won't cover all of its facilities here, but we'll get you off on the right foot. We'll use `inspect` to introspect the `sorted_set.py` module we built in Chapter 9, which contains the `SortedSet` class.

First, let's us `inspect.ismodule` to check that `sorted_set` is indeed a module:

```
>>> import inspect
>>> import sorted_set
>>> inspect.ismodule(sorted_set)
True
```

No surprises here.

inspect.getmembers()

The getmembers() function retrieves members as a list of name-value pairs. When we pass the module object to this it returns a vast output, including everything in the module namespace (which includes all the built-ins):

```
>>> inspect.getmembers(sorted_set)
[('Sequence', collections.abc.Sequence),
 ('Set', collections.abc.Set),
 ('SortedSet', sorted_set.SortedSet),
 ('__builtins__',
  {'ArithmeticError': builtins.ArithmeticError,
   'AssertionError': builtins.AssertionError,
   'AttributeError': builtins.AttributeError,
   'BaseException': builtins.BaseException,

   # ...

   'tuple': builtins.tuple,
   'type': builtins.type,
   'vars': <function builtins.vars>,
   'zip': builtins.zip}),
 ('__cached__',
  'examples/sorted_set/__pycache__/sorted_set.cpython-33.pyc'),
 ('__doc__', None),
 ('__file__',
  'examples/sorted_set/sorted_set.py'),
 ('__initializing__', False),
 ('__loader__', <_frozen_importlib.SourceFileLoader at 0x103b95d90>),
 ('__name__', 'sorted_set'),
 ('__package__', ''),
 ('bisect_left', <function _bisect.bisect_left>),
 ('chain', itertools.chain)]
```

Depending on the exact Python version you are using, the inspect module contains around twenty predicates similar to ismodule() for identifying different object types, from isabstract() to istraceback(). Let’s use another built-in introspective tool, the dir() function, in conjunction with a simple list comprehension, to see them:

```
>>> [m for m in dir(inspect) if m.startswith('is')]
['isabstract', 'isasyncgen', 'isasyncgenfunction', 'isawaitable', 'isbuiltin', 'iscla\
ss', 'iscode', 'iscoroutine', 'iscoroutinefunction', 'isdatadescriptor', 'isframe', '\
isfunction', 'isgenerator', 'isgeneratorfunction', 'isgetsetdescriptor', 'ismemberdes\
criptor', 'ismethod', 'ismethoddescriptor', 'ismodule', 'isroutine', 'istraceback']
```

This sort of filtering is useful, so let's see how to do something similar with `getmembers()`. We can attenuate the overwhelming output of `getmembers()` by providing a second argument which is a predicate function to filter the list for certain categories of object.

Let's pass the `inspect.isclass` predicate to filter for classes:

```
>>> inspect.getmembers(sorted_set, inspect.isclass)
[('Sequence', collections.abc.Sequence),
 ('Set', collections.abc.Set),
 ('SortedSet', sorted_set.SortedSet),
 ('chain', itertools.chain)]
```

You may have been expecting only `SortedSet` to be returned, since that is the only class we explicitly define in the `sorted_set` module. But in fact all the other classes we use — such as the `Sequence` and `Set` abstract base classes and even `chain`, which you probably didn't expect to be a class — are included. Presumably, since we call `chain` like a function it is a class which implements that `__call__()` special method.

But why are these objects returned? As it turns out, any class that ends up in the module namespace will be returned, whether defined or imported. It may surprise you to find that we can actually `import` any of these names *from* our `sorted_set` module, because `import` just binds objects in another module's namespace to names in the current namespace:

```
>>> from sorted_set import chain
>>> list(chain([1, 2, 3], [4, 5, 6]))
[1, 2, 3, 4, 5, 6]
```

Surprised? Perhaps even more surprising is that there's no straightforward way to stop this happening!

Of course, we can dig a little deeper into our module, for example, by retrieving all of the functions of the `SortedSet` class:

```
>>> inspect.getmembers(sorted_set.SortedSet, inspect.isfunction)
[('__add__', <function sorted_set.__add__>),
 ('__and__', <function collections.abc.__and__>),
 ('__contains__', <function sorted_set.__contains__>),
 ('__eq__', <function sorted_set.__eq__>),
 ('__ge__', <function collections.abc.__ge__>),
 ('__getitem__', <function sorted_set.__getitem__>),
 ('__gt__', <function collections.abc.__gt__>),
 ('__init__', <function sorted_set.__init__>),
 ('__iter__', <function sorted_set.__iter__>),
 ('__le__', <function collections.abc.__le__>),
 ('__len__', <function sorted_set.__len__>),
 ('__lt__', <function collections.abc.__lt__>),
 ('__mul__', <function sorted_set.__mul__>),
 ('__ne__', <function collections.abc.__ne__>),
 ('__or__', <function collections.abc.__or__>),
 ('__repr__', <function sorted_set.__repr__>),
 ('__reversed__', <function collections.abc.__reversed__>),
 ('__rmul__', <function sorted_set.__rmul__>),
 ('__sub__', <function collections.abc.__sub__>),
 ('__xor__', <function collections.abc.__xor__>),
 ('_hash', <function collections.abc._hash>),
 ('_is_unique_and_sorted', <function sorted_set._is_unique_and_sorted>),
 ('count', <function sorted_set.count>),
 ('difference', <function sorted_set.difference>),
 ('index', <function sorted_set.index>),
 ('intersection', <function sorted_set.intersection>),
 ('isdisjoint', <function collections.abc.isdisjoint>),
 ('issubset', <function sorted_set.issubset>),
 ('issuperset', <function sorted_set.issuperset>),
 ('symmetric_difference', <function sorted_set.symmetric_difference>),
 ('union', <function sorted_set.union>)]
```

Signatures

The inspect module contains tools for interrogating individual functions. This is done by retrieving a so-called `Signature` object for the function:

```
>>> init_sig = inspect.signature(sorted_set.SortedSet.__init__)
>>> init_sig
<inspect.Signature at 0x104186290>
```

From this signature object we can obtain a list of the parameters:

```
>>> init_sig.parameters
mappingproxy(OrderedDict(
    [('self', <Parameter at 0x104175f18 'self'>),
     ('items', <Parameter at 0x104175fc8 'items'>)]))
```

We can also query individual parameters for attributes such as their default values:

```
>>> repr(init_sig.parameters['items'].default)
'None'
```

Handlily, when we convert a Signature object to a string, we get a nice output:

```
>>> str(init_sig)
'(self, items=None)'
```

Be aware that inspect.signature() doesn't always work — some built in functions which are implemented in C don't provide sufficient metadata to be introspected this way. In this case, the function fails with a ValueError. Here we try to retrieve the signature of the built-in abs() function:

```
>>> inspect.signature(abs)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/inspect.py", \
line 1445, in signature
    raise ValueError(msg)
ValueError: no signature found for builtin function <built-in function abs>
```

There are many many tools in the inspect module which allow programs to instrospect themselves in very deep ways. We won't cover any more of them here, because they are very specialised.

An object introspection tool

We'll round off this chapter — and indeed this whole book! — by building tool to introspect objects. We'll use many of the useful and interesting techniques we have picked up throughout this book, bringing them all together into a useful program.

Our objective is to create a function which, when passed a single object, prints out a nicely formatted dump of that objects attributes and methods, similar to `getmembers()` but with rather more finesse.

Let's start by creating a module `introspector.py` and defining a `dump()` method that prints the outline of the object dump:

```python
# introspector.py

def dump(obj):
    print("Type")
    print("====")
    print(type(obj))
    print()

    print("Documentation")
    print("=============")
    # TODO
    print()

    print("Attributes")
    print("==========")
    # TODO
    print()

    print("Methods")
    print("=======")
    # TODO
    print()
```

We print four section headings, and the only object detail we print is the type retrieved using the `type()` built-in function. Let's work through this function, filling in the details as we go.

Introspecting documentation

First let's get some detail into the documentation section. The easiest thing to do here would be to simply print the `obj.__doc__` attribute, and indeed that would be a great start. We can do better, though, by using a function from the `inspect` module called `cleandoc()` which deals with tricky topics like uniformly removing leading whitespace from docstrings. Even better, there is a function `getdoc()` in the inspect module that will combine the operations of retrieving and tidying up the docstring. We'll use that instead, giving us very straightforward code for our documentation section:

```
# introspector.py

def dump(obj):
    . . .
    print("Documentation")
    print("=============")
    print(inspect.getdoc(obj))
    print()
```

Attributes and methods

Now for the attribute and methods. For this we need to produce a list of attributes and their values. We've just shown you how to do this using `inspect.getmembers()` — and although that's arguably the *right* course of action — we'll code up our own routine so that we have an opportunity to apply some of the other techniques we've picked up.

We know when can get a list of attributes using the `dir(obj)`, so we'll start with that. We'll put the attribute names into one of the `SortedSet` collections we defined earlier in the course:

```
all_attr_names = SortedSet(dir(obj))
```

Now we'll produce another `SortedSet` of all *method* names by determining which of those attributes, when retrieved with `getattr()`, is *callable*. We'll use `filter()` to select those attributes which are callable, and our predicate which acts on the attribute *name* will be a lambda function:

```
method_names = SortedSet(
      filter(lambda attr_name: callable(getattr(obj, attr_name)),
            all_attr_names))
```

The next part of our program will assume that `method_names` is a subset of `all_attr_names`, so we'll check that that is the case at this juncture by using an `assert` statement. Recall that the relational operators on `SortedSet` can be used to implement the subset test efficiently between two objects of the same type:

```
assert method_names <= all_attr_names
```

Now we use a set difference operation, called through the infix subtraction operator, to produce a `SortedSet` of all regular attributes — that is, those which are not callable — by subtracting the `method_names` from `all_attr_names`:

```
attr_names = all_attr_names - method_names
```

Now we're ready to print the attributes and their values. We'll do this by building a list of name-value pairs using a list comprehension. An attribute value could potentially have a huge text representation, which would spoil our output without adding much value, so we'll use the Python Standard Library `reprlib` module[108] to cut the values down to a reasonable size in an intelligent way:

```
attr_names_and_values = [(name, reprlib.repr(getattr(obj, name)))
                         for name in attr_names]
```

Then we'll print a nice table of names and values, using a putative `print_table()` function we'll implement later:

```
print_table(attr_names_and_values, "Name", "Value")
```

The function will need to accept a sequence of sequences representing rows of columns along with the requisite number of column headers as strings.

With the regular attributes dealt with, we can move onto methods. From our list of `method_names` we'll generate a series of method objects, simply by retrieving each one with `getattr()`:

[108] https://docs.python.org/3/library/reprlib.html

```
methods = (getattr(obj, method_name) for method_name in method_names)
```

Then for each method object, we'll build a full method signature and a brief documentation string, using functions we'll define in a moment. The signature and documentation will be assembled into a list of tuples using this list comprehension:

```
method_names_and_doc = [(full_sig(method), brief_doc(method))
                        for method in methods]
```

Finally, we'll print the table of methods using our yet-to-be- implemented `print_table()` function:

```
print_table(method_names_and_doc, "Name", "Description")
```

`full_sig()`

With the `dump()` function complete, let's now implement the three functions, on which it depends. First, `full_sig()`:

```
# introspector.py
. . .
def full_sig(method):
    try:
        return method.__name__ + inspect.signature(method)
    except ValueError:
        return method.__name__ + '(...)'
```

In this function we retrieve the method's name via the special `__name__` attribute, and then try to get the argument signatures using `inspect.signature()`. If that fails, we fall back in the exception handler to returning a string indicating that we couldn't determine the signature. This is a nice example of *Easier to Ask Forgiveness than Permission* programming style.

`brief_doc()`

Now, let's move onto implementing `brief_doc()`. Documentation strings can be very lengthy, and in our table we only want a brief description. Fortunately, there's a widely followed convention that the first line of a docstring contains exactly such a brief description. This function attempts to extract that line and return it:

```
# introspector.py
. . .
def brief_doc(obj):
    doc = obj.__doc__
    if doc is not None:
        lines = doc.splitlines()
        if len(lines) > 0:
            return lines[0]
    return ''
```

We must account for the fact that the docstring attribute may be set to `None`, in which case we wouldn't be able to call `splitlines()` on it. Likewise, the docstring may be defined but empty, in which case `splitlines()` would return an empty list. In either of these eventualities we return an empty string. This function follows a *Look Before You Leap* style of programming, because we felt the result was clearer than the alternative.

`print_table()`

Now we come on to the `print_table()` function. Recall that this function accepts a sequence of sequences as its first argument, with the outer sequence representing rows of the table and the inner sequences representing the columns within those rows. In addition the function accepts as many string arguments as are necessary to provide column headings. Let's start with the signature:

```
# introspector.py
. . .
def print_table(rows_of_columns, *headers):
```

We use extended argument syntax to accept any number of header arguments. We'll perform some basic validation by ensuring that the number of header values supplied is compatible with the first row of the table. We'll follow the lead of the built-in functions and raise a `TypeError` if too few arguments are provided:

```
# introspector.py
. . .
def print_table(rows_of_columns, *headers):
    num_columns = len(rows_of_columns[0])
    num_headers = len(headers)
    if len(headers) != num_columns:
        raise TypeError("Expected {} header arguments, "
                        "got {}".format(num_columns, num_headers))
```

See how we've split the long `raise` statement over several lines. First we use the fact we're allowed to continue over multiple lines inside open parentheses, second we use implicit string concatenation of adjacent string literals to glue two fragments of the message together.

Now we need to determine the width of each column in the table, taking into account the width of each item in the `rows_of_columns` data structure, but also the header widths. To do this, we'll lazily concatenate the header row and the data rows into a value `rows_of_columns_with_header` using `itertools.chain()`:

```
rows_of_columns_with_header = itertools.chain([headers], rows_of_columns)
```

Notice how we make a list containing the single headers row for the first argument.

Next, we use the zip-star idiom we covered in Chapter 2 to transpose the rows-of-columns into columns-of-rows. We force evaluation of the otherwise lazy `zip()` by constructing a `list`, binding the result to the name `columns_of_rows`:

```
columns_of_rows = list(zip(*rows_of_columns_with_header))
```

To find the maximum width of a column, we can use this expression:

```
max(map(len, column))
```

Let's break this down. We pass the `len` function, without calling it, to `map()`, which applies `len()` to each item in the `column` sequence, in effect yielding a series of widths. We then use `max()` to find the maximum width of the column.

We'd like to apply this to each column, so we put this expression in a list comprehension which does exactly that:

```
column_widths = [max(map(len, column)) for column in columns_of_rows]
```

Note that `map()` and list comprehensions are often considered to be alternatives. Here we've combined both because we think the result is more readable than using nested comprehensions or using two applications of `map()`. The resulting `column_widths` object is a list containing one integer for each column.

We want to print fixed width columns, and we know that we can do that with format specifiers and the `format()` string method[109]. However, we have a variable number of columns, so we need to build up our format string programmatically. Each column will need to have a format specifier something like this `"{:13}"` (for a column width of thirteen).

To do this, we can use the `format()` function to insert the width, but we need to escape the outer curly-braces so they carry through to the result; this escaping is performed by doubling the curly brace character. Applying this over all columns using a generator expression we get:

```
column_specs = ('{{:{w}}}'.format(w=width) for width in column_widths)
```

We concatenate all these column format specifications together using the `str.join()` method, inserting a space between each column:

```
format_spec = ' '.join(column_specs)
```

Now we're ready to begin printing our table. First, we use our `format_spec` to print the column headers, using extended argument unpacking to supply each header string as a separate argument to `format()`:

```
print(format_spec.format(*headers))
```

Now we'd like to print horizontal rules under the headers to separate them from the data below. We use string repetition with the multiply operator to create the individual rules by repeating hyphens, then we generate a rule for each column using a generator expression:

```
rules = ('-' * width for width in column_widths)
```

Then it's a simple matter to print the rules:

[109] https://www.python.org/dev/peps/pep-3101/#id19

```
print(format_spec.format(*rules))
```

Finally, we print each row of data:

```
for row in rows_of_columns:
    print(format_spec.format(*row))
```

Now we're done! Here's the complete code:

```
import itertools
import reprlib
import inspect

from sorted_set import SortedSet


def dump(obj):
    print("Type")
    print("====")
    print(type(obj))
    print()

    print("Documentation")
    print("=============")
    print(obj.__doc__)
    print()

    all_attr_names = SortedSet(dir(obj))
    method_names = SortedSet(
        filter(lambda attr_name: callable(getattr(obj, attr_name)),
               all_attr_names))
    assert method_names <= all_attr_names
    attr_names = all_attr_names - method_names

    print("Attributes")
    print("==========")
    attr_names_and_values = [(name, reprlib.repr(getattr(obj, name)))
                             for name in attr_names]
    print_table(attr_names_and_values, "Name", "Value")
    print()
    print("Methods")
    print("=======")
    methods = (getattr(obj, method_name) for method_name in method_names)
    method_names_and_doc = [(full_sig(method), brief_doc(method)) for method in metho\
ds]
    print_table(method_names_and_doc, "Name", "Description")
    print()


def full_sig(method):
    try:
        return method.__name__ + inspect.signature(method)
    except ValueError:
        return method.__name__ + '(...)'


def brief_doc(obj):
    doc = obj.__doc__
    if doc is not None:
        lines = doc.splitlines()
        if len(lines) > 0:
            return lines[0]
    return ''


def print_table(rows_of_columns, *headers):
    num_columns = len(rows_of_columns[0])
    num_headers = len(headers)
    if len(headers) != num_columns:
        raise TypeError("Expected {} header arguments, "
                        "got {}".format(num_columns, num_headers))

    rows_of_columns_with_header = itertools.chain([headers], rows_of_columns)
    columns_of_rows = list(zip(*rows_of_columns_with_header))
    column_widths = [max(map(len, column)) for column in columns_of_rows]
    column_specs = ('{{:{w}}}'.format(w=width) for width in column_widths)
    format_spec = ' '.join(column_specs)

    print(format_spec.format(*headers))

    rules = ('-' * width for width in column_widths)
    print(format_spec.format(*rules))
    for row in rows_of_columns:
        print(format_spec.format(*row))
```

Let see if the `dump()` function works on a simple object like the integer digit seven:

```
>>> from introspector import dump
>>> dump(7)
Type
====
<class 'int'>

Documentation
int(x=0) -> integer
int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments
are given.  If x is a number, return x.__int__().  For floating point
numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string,
bytes, or bytearray instance representing an integer literal in the
given base.  The literal can be preceded by '+' or '-' and be surrounded
by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
Base 0 means to interpret the base from the string as an integer literal.
int('0b100', base=0)
4

Attributes
==========
Name        Value
----------- ------------------------------
__doc__     "int(x=0) -> ...', base=0)\n4"
denominator 1
imag        0
numerator   7
real        7

Methods
=======
Name                  Description
--------------------- --------------------------------------------------------------
__abs__(...)          x.__abs__() <==> abs(x)
```

```
__add__(...)          x.__add__(y) <==> x+y
__and__(...)          x.__and__(y) <==> x&y
__bool__(...)         x.__bool__() <==> x != 0
__ceil__(...)         Ceiling of an Integral returns itself.
int(...)              int(x=0) -> integer
__delattr__(...)      x.__delattr__('name') <==> del x.name
__dir__(...)          __dir__() -> list
__divmod__(...)       x.__divmod__(y) <==> divmod(x, y)
__eq__(...)           x.__eq__(y) <==> x==y
__float__(...)        x.__float__() <==> float(x)
__floor__(...)        Flooring an Integral returns itself.
__floordiv__(...)     x.__floordiv__(y) <==> x//y
__format__(...)
__ge__(...)           x.__ge__(y) <==> x>=y
__getattribute__(...) x.__getattribute__('name') <==> x.name
__getnewargs__(...)
__gt__(...)           x.__gt__(y) <==> x>y
__hash__(...)         x.__hash__() <==> hash(x)
__index__(...)        x[y:z] <==> x[y.__index__():z.__index__()]
__init__(...)         x.__init__(...) initializes x; see help(type(x)) for signature
__int__(...)          x.__int__() <==> int(x)
__invert__(...)       x.__invert__() <==> ~x
__le__(...)           x.__le__(y) <==> x<=y
__lshift__(...)       x.__lshift__(y) <==> x<<y
__lt__(...)           x.__lt__(y) <==> x<y
__mod__(...)          x.__mod__(y) <==> x%y
__mul__(...)          x.__mul__(y) <==> x*y
__ne__(...)           x.__ne__(y) <==> x!=y
__neg__(...)          x.__neg__() <==> -x
__new__(...)          T.__new__(S, ...) -> a nameew object with type S, a subtype of T
__or__(...)           x.__or__(y) <==> x|y
__pos__(...)          x.__pos__() <==> +x
__pow__(...)          x.__pow__(y[, z]) <==> pow(x, y[, z])
__radd__(...)         x.__radd__(y) <==> y+x
__rand__(...)         x.__rand__(y) <==> y&x
__rdivmod__(...)      x.__rdivmod__(y) <==> divmod(y, x)
__reduce__(...)       helper for pickle
__reduce_ex__(...)    helper for pickle
__repr__(...)         x.__repr__() <==> repr(x)
__rfloordiv__(...)    x.__rfloordiv__(y) <==> y//x
__rlshift__(...)      x.__rlshift__(y) <==> y<<x
__rmod__(...)         x.__rmod__(y) <==> y%x
__rmul__(...)         x.__rmul__(y) <==> y*x
```

```
__ror__(...)         x.__ror__(y) <==> y|x
__round__(...)       Rounding an Integral returns itself.
__rpow__(...)        y.__rpow__(x[, z]) <==> pow(x, y[, z])
__rrshift__(...)     x.__rrshift__(y) <==> y>>x
__rshift__(...)      x.__rshift__(y) <==> x>>y
__rsub__(...)        x.__rsub__(y) <==> y-x
__rtruediv__(...)    x.__rtruediv__(y) <==> y/x
__rxor__(...)        x.__rxor__(y) <==> y^x
__setattr__(...)     x.__setattr__('name', value) <==> x.name = value
__sizeof__(...)      Returns size in memory, in bytes
__str__(...)         x.__str__() <==> str(x)
__sub__(...)         x.__sub__(y) <==> x-y
__subclasshook__(...) Abstract classes can override this to customize issubclass().
__truediv__(...)     x.__truediv__(y) <==> x/y
__trunc__(...)       Truncating an Integral returns itself.
__xor__(...)         x.__xor__(y) <==> x^y
bit_length(...)      int.bit_length() -> int
conjugate(...)       Returns self, the complex conjugate of any int.
from_bytes(...)      int.from_bytes(bytes, byteorder, *, signed=False) -> int
to_bytes(...)        int.to_bytes(length, byteorder, *, signed=False) -> bytes
```

Summary

Let’s summarise what we’ve covered in the module on introspection in Python.

- Types and `type()`
    - We investigated the type of objects using the `type()` function,
    - We asked deep questions such as what is the `type` of `type`, learning that the answer is `type`!
    - We saw the using `type()` is the equivalent of retrieving the special `__class__` attribute.
    - We learned that both `object` and `type` are the foundation of the Python type system.
- Built-in introspection functions
    - We introspected objects using the `dir()` function we had already met, and the `getattr()` and `hasattr()` methods which are new to us.
    - We showed how to determine whether or not an object can be called like a function using the `callable()` predicate.
    - We showed how to retrieve the name of a class or function by fetching the special `__name__` attribute.

  - We showed how `hasattr()` can be used to support a Look Before You Leap programming style, although we reminded you that Easier To Ask Forgiveness than Permission is generally preferred in Python.
  - We demonstrated the built-in `globals()` and `locals()` functions for gaining direct access to namespace dictionaries.
- `inspect`
  - We introduced the Python Standard Library `inspect` module and used it to extract information about certain types of module and class attributes.
  - We showed how to retrieve function signatures using the `inspect.signature()` function, allowing details such as default function arguments to be retrieved.
- We gave an extended example of an object instrospection tool which uses the techniques we have learned in this module together with many techniques from earlier in this course.

Afterword: Levelling Up

Having digested in the material in *The Python Apprentice* and *The Python Journeyman*, you have arrived at a significant point on your journey as a Python programmer. You should have the all core Python language tools *at hand* to tackle almost any Python development project.

It will take time to put these tools to use. Time to discover where certain tools are most effective. Time discover where certain techniques fall short. Time to discover that – out of necessity – we skipped seemingly minor details, which may inflate in significance in your own context. A good text or an inspiring teacher can be an profitable shortcut, but nothing can eliminate the need to spend time solving your *own* problems by writing Python code regularly, on the route to becoming an effective Python programmer.

So what lies beyond?

The journey alluded to in the name *The Python Journeyman* continues in the direction of mastery. As we've repeatedly said, Python is a large language, and though we have covered a great swathe in the *breadth* of Python, we have barely begun to peel back the layers of abstraction to see what lies in the *depths* beneath.

In *The Python Master*[110], we'll look at descriptors and metaclasses; constructs on which you've been relying almost since the beginning of your journey, but which Python has cleverly kept hidden from view. We'll look at advanced control flow. We'll look at memory allocation. And we'll look at Python's somewhat esoteric tastes in object-oriented modelling. In other words, there's plenty more to cover.

If you're interested in learning Python in other forms, be sure look at the Python courses on PluralSight, *Python Fundamentals*[111], *Python: Beyond the Basics*[112], and *Advanced Python*[113]. We also offer in-house Python training[114] and consulting[115] through our company Sixty North if you have more substantial needs.

Wherever your journey with Python takes you, we sincerely hope you've enjoyed this book. Python is a wonderful language with a great community, and we want you to get as much

[110] https://leanpub.com/python-master

[111] https://app.pluralsight.com/library/courses/python-fundamentals/

[112] https://app.pluralsight.com/library/courses/python-beyond-basics/

[113] https://app.pluralsight.com/library/courses/advanced-python/

[114] http://sixty-north.com/training.html

[115] http://sixty-north.com/consulting.html

joy from it as we have. Happy programming!

Appendix A - Python implementation of ISO6346

This is a Python implementation of the ISO6346 standard for BIC codes.

```python
# iso6346.py

"""
ISO 6346 shipping container codes.
"""


def create(owner_code, serial, category='U'):
    """Create an ISO 6346 shipping container code.

    Args:
        owner_code (str): Three character alphabetic container code.
        serial (str): Six digit numeric serial number.
        category (str): Equipment category identifier.

    Returns:
        An ISO 6346 container code including a check digit.

    Raises:
        ValueError: If incorrect values are provided.
    """
    if not (len(owner_code) == 3 and owner_code.isalpha()):
        raise ValueError("Invalid ISO 6346 owner code '{}'".format(owner_code))

    if category not in ('U', 'J', 'Z', 'R'):
        raise ValueError("Invalid ISO 6346 category identifier '{}'".format(category))

    if not (len(serial) == 6 and serial.isdigit()):
        raise ValueError("Invalid ISO 6346 serial number")

    raw_code = owner_code + category + serial
    full_code = raw_code + str(check_digit(raw_code))
    return full_code


def check_digit(raw_code):
    """Compute the check digit for an ISO 6346 code without that digit

    Args:
        raw_code (str): An ISO 6346 code lacking a check digit.

    Returns:
        An integer check digit between 0 and 9 inclusive.
    """
    s = sum(code(char) * 2**index for index, char in enumerate(raw_code))
    return s % 11 % 10


def code(char):
    """Determine the ISO 6346 numeric equivalent of a character.

    Args:
        char (str): A single character string.

    Return:
        An integer code equivalent to the supplied character.
    """
    return int(char) if char.isdigit() else letter_code(char)


def letter_code(letter):
    """Determine the ISO 6346 numeric code for a letter.

    Args:
        letter (str): A single letter.

    Returns:
        An integer character code equivalent to the supplied letter.
    """
    value = ord(letter.lower()) - ord('a') + 10
    return value + value // 11
```