Python: Journey from Novice to Expert

Learn core concepts of Python and unleash its power to script highest quality Python programs


Packt>

LEARNING PATH

Python: Journey from Novice to Expert

Learn core concepts of Python and unleash its power to script highest quality Python programs.

A course in three modules

Packt>

BIRMINGHAM - MUMBAI

Python: Journey from Novice to Expert

Published on: August 2016

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78712-076-1

www.packtpub.com

Credits

Authors

Fabrizio Romano

Dusty Phillips

Rick van Hattem

Reviewers

Simone Burol

Julio Vicente Trigo Guijarro

Veit Heller

AMahdy AbdElAziz

Grigoriy Beziuk

Krishna Bharadwaj

Justin Cano

Anthony Petitbois

Claudio Rodriguez

Randall Degges

Dave de Fijter

I. de Hoogt

Content Development Editor

Onkar Wani

Graphics

Abhinash Sahu

Production Coordinator

Melwyn Dsa

Preface

Python is a dynamic programming language. It is known for its high readability and hence it is often the first language learned by new programmers. Python being multi-paradigm, it can be used to achieve the same thing in different ways and it is compatible across different platforms. Coding in Python minimizes development time and increases productivity in comparison to other languages. Clean, maintainable code is easy to both read and write using Python's clear, concise syntax.

What this learning path covers

Module 1, Learning Python, This module begins by exploring the essentials of programming, data structures and teaches you how to manipulate them. It then moves on to controlling the flow of a program and writing reusable and error proof code. You will then explore different programming paradigms that will allow you to find the best approach to any situation, and also learn how to perform performance optimization as well as effective debugging. Throughout, the module steers you through the various types of applications, and it concludes with a complete mini website built upon all the concepts that you learned.

Module 2, Python 3 Object-Oriented Programming, Second Edition, You will learn how to use the Python programming language to clearly grasp key concepts from the object-oriented paradigm. This modules fully explains classes, data encapsulation, inheritance, polymorphism, abstraction, and exceptions with an emphasis on when you can use each principle to develop a well-designed software. You'll get an in-depth analysis of many common object-oriented design patterns that are more suitable to Python's unique style. This module will not just teach Python syntax, but will also build your confidence in how to program and create maintainable applications with higher level design patterns.

Module 3, Mastering Python, This module is an authoritative guide that will help you learn new advanced methods in a clear and contextualized way. It starts off by creating a project-specific environment using venv, introducing you to different Pythonic syntax and common pitfalls before moving on to cover the functional features in Python. It covers how to create different decorators, generators, and metaclasses. It also introduces you to functools.wraps and coroutines and how they work. Later on you will learn to use asyncio module for asynchronous clients and servers. You will also get familiar with different testing systems such as py.test, doctest, and unittest, and debugging tools such as Python debugger and faulthandler. You will learn to optimize application performance so that it works efficiently across multiple machines and Python versions. Finally, it will teach you how to access C functions with a simple Python call. By the end of the module, you will be able to write more advanced scripts and take on bigger challenges.

What you need for this learning path

Module 1:

You are encouraged to follow the examples in this module. In order to do so, you will need a computer, an Internet connection, and a browser. The module is written in Python 3.4, but it should also work with any Python 3.* version. It has written instructions on how to install Python on the three main operating systems used today: Windows, Mac, and Linux. This module also explained how to install all the extra libraries used in the various examples and provided suggestions if the reader finds any issues during the installation of any of them. No particular editor is required to type the code; however, module suggest that those who are interested in following the examples should consider adopting a proper coding environment.

Module 2:

All the examples in this module rely on the Python 3 interpreter. Make sure you are not using Python 2.7 or earlier. At the time of writing, Python 3.4 was the latest release of Python. Most examples will work on earlier revisions of Python 3, but you are encouraged to use the latest version to minimize frustration. All of the examples should run on any operating system supported by Python.

If this is not the case, please report it as a bug. Some of the examples need a working Internet connection. You'll probably want to have one of these for extracurricular research and debugging anyway! In addition, some of the examples in this module rely on third-party libraries that do not ship with Python. These are introduced within the module at the time they are used, so you do not need to install them in advance. However, for completeness, here is a list:

- pip
- requests
- pillow
- bitarray

Module 3:

The only hard requirement for this module is a Python interpreter. A Python 3.5 or newer interpreter is recommended, but many of the code examples will function in older Python versions, such as 2.7, with a simple from __future__ import print_ statement added at the top of the file.Additionally, Chapter 14, Extensions in C/C++, System Calls, and C/C++ Librariesrequires a C/C++ compiler, such as GCC, Visual Studio, or XCode. A Linux machine is by far the easiest to execute the C/C++ examples, but these should function on Windows and OS X machines without too much effort as well.

Who this learning path is for

This course is meant for programmes who wants learn Python programming from a basic to an expert level. The course is mostly self-contained and introduces Python Programming to a new reader and can help him become an expert in this trade. Intended for students and practitioners from novice to experts.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the course's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a course, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt course, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this course from your account at `http://www.packtpub.com`. If you purchased this course elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the course is also hosted on GitHub at `https://github.com/PacktPublishing/Python-Journey-from-Novice-to-Expert`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our courses—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this course. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your course, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the course in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this course, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

Module 1: Learning Python

Module 2: Python 3 Object-Oriented Programming

Module 3: Mastering Python

Module 1

Learning Python

Learn to code like a professional with Python – an open source, versatile and powerful programming language

1

Introduction and First Steps – Take a Deep Breath

> "*Give a man a fish and you feed him for a day. Teach a man to fish and you feed him for a lifetime.*"
>
> *– Chinese proverb*

According to Wikipedia, **computer programming** is:

> "*...a process that leads from an original formulation of a computing problem to executable computer programs. Programming involves activities such as analysis, developing understanding, generating algorithms, verification of requirements of algorithms including their correctness and resources consumption, and implementation (commonly referred to as coding) of algorithms in a target programming language*".

In a nutshell, coding is telling a computer to do something using a language it understands.

Computers are very powerful tools, but unfortunately, they can't think for themselves. So they need to be told everything. They need to be told how to perform a task, how to evaluate a condition to decide which path to follow, how to handle data that comes from a device such as the network or a disk, and how to react when something unforeseen happens, say, something is broken or missing.

You can code in many different styles and languages. Is it hard? I would say "yes" and "no". It's a bit like writing. Everybody can learn how to write, and you can too. But what if you wanted to become a poet? Then writing alone is not enough. You have to acquire a whole other set of skills and this will take a longer and greater effort.

In the end, it all comes down to how far you want to go down the road. Coding is not just putting together some instructions that work. It is so much more!

Good code is short, fast, elegant, easy to read and understand, simple, easy to modify and extend, easy to scale and refactor, and easy to test. It takes time to be able to write code that has all these qualities at the same time, but the good news is that you're taking the first step towards it at this very moment by reading this book. And I have no doubt you can do it. Anyone can, in fact, we all program all the time, only we aren't aware of it.

Would you like an example?

Say you want to make instant coffee. You have to get a mug, the instant coffee jar, a teaspoon, water, and the kettle. Even if you're not aware of it, you're evaluating a lot of data. You're making sure that there is water in the kettle as well as the kettle is plugged-in, that the mug is clean, and that there is enough coffee in the jar. Then, you boil the water and maybe in the meantime you put some coffee in the mug. When the water is ready, you pour it into the cup, and stir.

So, how is this programming?

Well, we gathered resources (the kettle, coffee, water, teaspoon, and mug) and we verified some conditions on them (kettle is plugged-in, mug is clean, there is enough coffee). Then we started two actions (boiling the water and putting coffee in the mug), and when both of them were completed, we finally ended the procedure by pouring water in the mug and stirring.

Can you see it? I have just described the high-level functionality of a coffee program. It wasn't that hard because this is what the brain does all day long: evaluate conditions, decide to take actions, carry out tasks, repeat some of them, and stop at some point. Clean objects, put them back, and so on.

All you need now is to learn how to deconstruct all those actions you do automatically in real life so that a computer can actually make some sense of them. And you need to learn a language as well, to instruct it.

So this is what this book is for. I'll tell you how to do it and I'll try to do that by means of many simple but focused examples (my favorite kind).

A proper introduction

I love to make references to the real world when I teach coding; I believe they help people retain the concepts better. However, now is the time to be a bit more rigorous and see what coding is from a more technical perspective.

When we write code, we're instructing a computer on what are the things it has to do. Where does the action happen? In many places: the computer memory, hard drives, network cables, CPU, and so on. It's a whole "world", which most of the time is the representation of a subset of the real world.

If you write a piece of software that allows people to buy clothes online, you will have to represent real people, real clothes, real brands, sizes, and so on and so forth, within the boundaries of a program.

In order to do so, you will need to create and handle objects in the program you're writing. A person can be an object. A car is an object. A pair of socks is an object. Luckily, Python understands objects very well.

The two main features any object has are properties and methods. Let's take a person object as an example. Typically in a computer program, you'll represent people as customers or employees. The properties that you store against them are things like the name, the SSN, the age, if they have a driving license, their e-mail, gender, and so on. In a computer program, you store all the data you need in order to use an object for the purpose you're serving. If you are coding a website to sell clothes, you probably want to store the height and weight as well as other measures of your customers so that you can suggest the appropriate clothes for them. So, properties are characteristics of an object. We use them all the time: "Could you pass me that pen?" – "Which one?" – "The black one." Here, we used the "black" property of a pen to identify it (most likely amongst a blue and a red one).

Methods are things that an object can do. As a person, I have methods such as *speak*, *walk*, *sleep*, *wake-up*, *eat*, *dream*, *write*, *read*, and so on. All the things that I can do could be seen as methods of the objects that represents me.

So, now that you know what objects are and that they expose methods that you can run and properties that you can inspect, you're ready to start coding. Coding in fact is simply about managing those objects that live in the subset of the world that we're reproducing in our software. You can create, use, reuse, and delete objects as you please.

According to the *Data Model* chapter on the official Python documentation:

> "*Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects.*"

We'll take a closer look at Python objects in *Chapter 6*, *Advanced Concepts – OOP, Decorators, and Iterators*. For now, all we need to know is that every object in Python has an ID (or identity), a type, and a value.

Once created, the identity of an object is never changed. It's a unique identifier for it, and it's used behind the scenes by Python to retrieve the object when we want to use it.

The type as well, never changes. The type tells what operations are supported by the object and the possible values that can be assigned to it.

We'll see Python's most important data types in *Chapter 2, Built-in Data Types*.

The value can either change or not. If it can, the object is said to be **mutable**, while when it cannot, the object is said to be **immutable**.

How do we use an object? We give it a name of course! When you give an object a name, then you can use the name to retrieve the object and use it.

In a more generic sense, objects such as numbers, strings (text), collections, and so on are associated with a name. Usually, we say that this name is the name of a variable. You can see the variable as being like a box, which you can use to hold data.

So, you have all the objects you need: what now? Well, we need to use them, right? We may want to send them over a network connection or store them in a database. Maybe display them on a web page or write them into a file. In order to do so, we need to react to a user filling in a form, or pressing a button, or opening a web page and performing a search. We react by running our code, evaluating conditions to choose which parts to execute, how many times, and under which circumstances.

And to do all this, basically we need a language. That's what Python is for. Python is the language we'll use together throughout this book to instruct the computer to do something for us.

Now, enough of this theoretical stuff, let's get started.

Enter the Python

Python is the marvelous creature of Guido Van Rossum, a Dutch computer scientist and mathematician who decided to gift the world with a project he was playing around with over Christmas 1989. The language appeared to the public somewhere around 1991, and since then has evolved to be one of the leading programming languages used worldwide today.

I started programming when I was 7 years old, on a Commodore VIC 20, which was later replaced by its bigger brother, the Commodore 64. The language was BASIC. Later on, I landed on Pascal, Assembly, C, C++, Java, JavaScript, Visual Basic, PHP, ASP, ASP .NET, C#, and other minor languages I cannot even remember, but only when I landed on Python, I finally had that feeling that you have when you find the right couch in the shop. When all of your body parts are yelling, "Buy this one! This one is perfect for us!"

It took me about a day to get used to it. Its syntax is a bit different from what I was used to, and in general, I very rarely worked with a language that defines scoping with indentation. But after getting past that initial feeling of discomfort (like having new shoes), I just fell in love with it. Deeply. Let's see why.

About Python

Before we get into the gory details, let's get a sense of why someone would want to use Python (I would recommend you to read the Python page on Wikipedia to get a more detailed introduction).

To my mind, Python exposes the following qualities.

Portability

Python runs everywhere, and porting a program from Linux to Windows or Mac is usually just a matter of fixing paths and settings. Python is designed for portability and it takes care of **operating system** (**OS**) specific quirks behind interfaces that shield you from the pain of having to write code tailored to a specific platform.

Coherence

Python is extremely logical and coherent. You can see it was designed by a brilliant computer scientist. Most of the time you can just guess how a method is called, if you don't know it.

You may not realize how important this is right now, especially if you are at the beginning, but this is a major feature. It means less cluttering in your head, less skimming through the documentation, and less need for mapping in your brain when you code.

Developer productivity

According to Mark Lutz (*Learning Python, 5th Edition*, *O'Reilly Media*), a Python program is typically one-fifth to one-third the size of equivalent Java or C++ code. This means the job gets done faster. And faster is good. Faster means a faster response on the market. Less code not only means less code to write, but also less code to read (and professional coders read much more than they write), less code to maintain, to debug, and to refactor.

Another important aspect is that Python runs without the need of lengthy and time consuming compilation and linkage steps, so you don't have to wait to see the results of your work.

An extensive library

Python has an incredibly wide standard library (it's said to come with "batteries included"). If that wasn't enough, the Python community all over the world maintains a body of third party libraries, tailored to specific needs, which you can access freely at the **Python Package Index** (**PyPI**). When you code Python and you realize that you need a certain feature, in most cases, there is at least one library where that feature has already been implemented for you.

Software quality

Python is heavily focused on readability, coherence, and quality. The language uniformity allows for high readability and this is crucial nowadays where code is more of a collective effort than a solo experience. Another important aspect of Python is its intrinsic multi-paradigm nature. You can use it as scripting language, but you also can exploit object-oriented, imperative, and functional programming styles. It is versatile.

Software integration

Another important aspect is that Python can be extended and integrated with many other languages, which means that even when a company is using a different language as their mainstream tool, Python can come in and act as a glue agent between complex applications that need to talk to each other in some way. This is kind of an advanced topic, but in the real world, this feature is very important.

Satisfaction and enjoyment

Last but not least, the fun of it! Working with Python is fun. I can code for 8 hours and leave the office happy and satisfied, alien to the struggle other coders have to endure because they use languages that don't provide them with the same amount of well-designed data structures and constructs. Python makes coding fun, no doubt about it. And fun promotes motivation and productivity.

These are the major aspects why I would recommend Python to everyone for. Of course, there are many other technical and advanced features that I could have talked about, but they don't really pertain to an introductory section like this one. They will come up naturally, chapter after chapter, in this book.

What are the drawbacks?

Probably, the only drawback that one could find in Python, which is not due to personal preferences, is the *execution speed*. Typically, Python is slower than its compiled brothers. The standard implementation of Python produces, when you run an application, a compiled version of the source code called byte code (with the extension `.pyc`), which is then run by the Python interpreter. The advantage of this approach is portability, which we pay for with a slowdown due to the fact that Python is not compiled down to machine level as are other languages.

However, Python speed is rarely a problem today, hence its wide use regardless of this suboptimal feature. What happens is that in real life, hardware cost is no longer a problem, and usually it's easy enough to gain speed by parallelizing tasks. When it comes to number crunching though, one can switch to faster Python implementations, such as PyPy, which provides an average 7-fold speedup by implementing advanced compilation techniques (check `http://pypy.org/` for reference).

When doing data science, you'll most likely find that the libraries that you use with Python, such as Pandas and Numpy, achieve native speed due to the way they are implemented.

If that wasn't a good enough argument, you can always consider that Python is driving the backend of services such as Spotify and Instagram, where performance is a concern. Nonetheless, Python does its job perfectly adequately.

Who is using Python today?

Not yet convinced? Let's take a very brief look at the companies that are using Python today: Google, YouTube, Dropbox, Yahoo, Zope Corporation, Industrial Light & Magic, Walt Disney Feature Animation, Pixar, NASA, NSA, Red Hat, Nokia, IBM, Netflix, Yelp, Intel, Cisco, HP, Qualcomm, and JPMorgan Chase, just to name a few.

Even games such as *Battlefield 2*, *Civilization 4*, and *QuArK* are implemented using Python.

Python is used in many different contexts, such as system programming, web programming, GUI applications, gaming and robotics, rapid prototyping, system integration, data science, database applications, and much more.

Setting up the environment

Before we talk about installing Python on your system, let me tell you about which Python version I'll be using in this book.

Python 2 versus Python 3 – the great debate

Python comes in two main versions—Python 2, which is the past—and Python 3, which is the present. The two versions, though very similar, are incompatible on some aspects.

In the real world, Python 2 is actually quite far from being the past. In short, even though Python 3 has been out since 2008, the transition phase is still far from being over. This is mostly due to the fact that Python 2 is widely used in the industry, and of course, companies aren't so keen on updating their systems just for the sake of updating, following the *if it ain't broke, don't fix it* philosophy. You can read all about the transition between the two versions on the Web.

Another issue that was hindering the transition is the availability of third-party libraries. Usually, a Python project relies on tens of external libraries, and of course, when you start a new project, you need to be sure that there is already a version 3 compatible library for any business requirement that may come up. If that's not the case, starting a brand new project in Python 3 means introducing a potential risk, which many companies are not happy to take.

At the time of writing, the majority of the most widely used libraries have been ported to Python 3, and it's quite safe to start a project in Python 3 for most cases. Many of the libraries have been rewritten so that they are compatible with both versions, mostly harnessing the power of the six (2 x 3) library, which helps introspecting and adapting the behavior according to the version used.

On my Linux box (Ubuntu 14.04), I have the following Python version:

```
>>> import sys
>>> print(sys.version)
3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2]
```

So you can see that my Python version is 3.4.0. The preceding text is a little bit of Python code that I typed into my console. We'll talk about it in a moment.

All the examples in this book will be run using this Python version. Most of them will run also in Python 2 (I have version 2.7.6 installed as well), and those that won't will just require some minor adjustments to cater for the small incompatibilities between the two versions. Another reason behind this choice is that I think it's better to learn Python 3, and then, if you need to, learn the differences it has with Python 2, rather than going the other way around.

Don't worry about this version thing though: it's not that big an issue in practice.

Installing Python

I never really got the point of having a *setup* section in a book, regardless of what it is that you have to set up. Most of the time, between the time the author writes the instruction and the time you actually try them out, months have passed. That is, if you're lucky. One version change and things may not work the way it is described in the book. Luckily, we have the Web now, so in order to help you get up and running, I'll just give you pointers and objectives.

If any of the URLs or resources I'll point you to are no longer there by the time you read this book, just remember: Google is your friend.

Setting up the Python interpreter

First of all, let's talk about your OS. Python is fully integrated and most likely already installed in basically almost every Linux distribution. If you have a Mac, it's likely that Python is already there as well (however, possibly only Python 2.7), whereas if you're using Windows, you probably need to install it.

Getting Python and the libraries you need up and running requires a bit of handiwork. Linux happens to be the most user friendly OS for Python programmers, Windows on the other hand is the one that requires the biggest effort, Mac being somewhere in between. For this reason, if you can choose, I suggest you to use Linux. If you can't, and you have a Mac, then go for it anyway. If you use Windows, you'll be fine for the examples in this book, but in general working with Python will require you a bit more tweaking.

My OS is Ubuntu 14.04, and this is what I will use throughout the book, along with Python 3.4.0.

The place you want to start is the official Python website: `https://www.python.org`. This website hosts the official Python documentation and many other resources that you will find very useful. Take the time to explore it.

Another excellent, resourceful website on Python and its ecosystem is `http://docs.python-guide.org`.

Find the download section and choose the installer for your OS. If you are on Windows, make sure that when you run the installer, you check the option `install pip` (actually, I would suggest to make a complete installation, just to be safe, of all the components the installer holds). We'll talk about pip later.

Now that Python is installed in your system, the objective is to be able to open a console and run the Python interactive shell by typing `python`.

Please note that I usually refer to the *Python interactive shell* simply as *Python console*.

To open the console in Windows, go to the **Start** menu, choose **Run**, and type `cmd`. If you encounter anything that looks like a permission problem while working on the examples of this book, please make sure you are running the console with administrator rights.

On the Mac OS X, you can start a terminal by going to **Applications** | **Utilities** | **Terminal**.

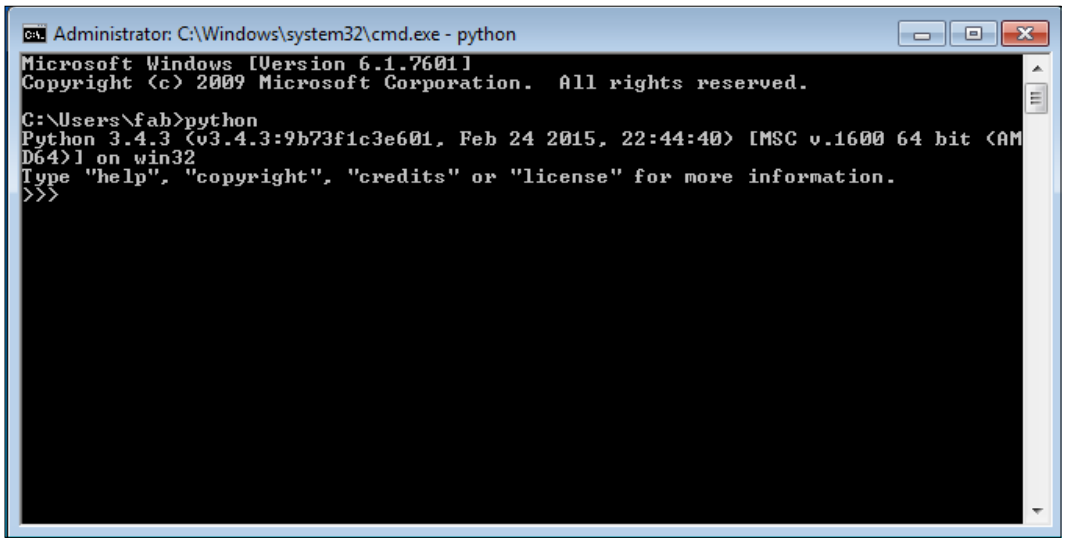
If you are on Linux, you know all that there is to know about the console.

I will use the term *console* interchangeably to indicate the Linux **console**, the Windows **command prompt**, and the Mac **terminal**. I will also indicate the command-line prompt with the Linux default format, like this:

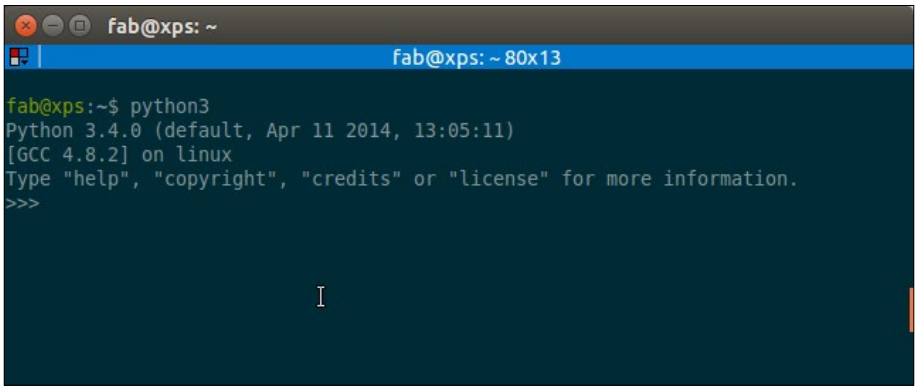
```
$ sudo apt-get update
```

Whatever console you open, type `python` at the prompt, and make sure the Python interactive shell shows up. Type `exit()` to quit. Keep in mind that you may have to specify `python3` if your OS comes with Python 2.* preinstalled.

This is how it should look on Windows 7:



And this is how it should look on Linux:



Now that Python is set up and you can run it, it's time to make sure you have the other tool that will be indispensable to follow the examples in the book: virtualenv.

About virtualenv

As you probably have guessed by its name, **virtualenv** is all about virtual environments. Let me explain what they are and why we need them and let me do it by means of a simple example.

You install Python on your system and you start working on a website for client X. You create a project folder and start coding. Along the way you also install some libraries, for example the Django framework, which we'll see in depth in *Chapter 10, Web Development Done Right*. Let's say the Django version you install for project X is 1.7.1.

Now, your website is so good that you get another client, Y. He wants you to build another website, so you start project Y and, along the way, you need to install Django again. The only issue is that now the Django version is 1.8 and you cannot install it on your system because this would replace the version you installed for project X. You don't want to risk introducing incompatibility issues, so you have two choices: either you stick with the version you have currently on your machine, or you upgrade it and make sure the first project is still fully working correctly with the new version.

Let's be honest, neither of these options is very appealing, right? Definitely not. So, here's the solution: virtualenv!

virtualenv is a tool that allows you to create a virtual environment. In other words, it is a tool to create isolated Python environments, each of which is a folder that contains all the necessary executables to use the packages that a Python project would need (think of packages as libraries for the time being).

So you create a virtual environment for project X, install all the dependencies, and then you create a virtual environment for project Y, installing all its dependencies without the slightest worry because every library you install ends up within the boundaries of the appropriate virtual environment. In our example, project X will hold Django 1.7.1, while project Y will hold Django 1.8.

It is of vital importance that you never install libraries directly at the system level. Linux for example relies on Python for many different tasks and operations, and if you fiddle with the system installation of Python, you risk compromising the integrity of the whole system (guess to whom this happened…). So take this as a rule, such as brushing your teeth before going to bed: *always, always create a virtual environment when you start a new project*.

To install virtualenv on your system, there are a few different ways. On a Debian-based distribution of Linux for example, you can install it with the following command:

```
$ sudo apt-get install python-virtualenv
```

Probably, the easiest way is to use `pip` though, with the following command:

```
$ sudo pip install virtualenv # sudo may by optional
```

`pip` is a package management system used to install and manage software packages written in Python.

Python 3 has built-in support for virtual environments, but in practice, the external libraries are still the default on production systems. If you have trouble getting virtualenv up and running, please refer to the virtualenv official website: `https://virtualenv.pypa.io`.

Your first virtual environment

It is very easy to create a virtual environment, but according to how your system is configured and which Python version you want the virtual environment to run, you need to run the command properly. Another thing you will need to do with a virtualenv, when you want to work with it, is to activate it. Activating a virtualenv basically produces some path juggling behind the scenes so that when you call the Python interpreter, you're actually calling the active virtual environment one, instead of the mere system one.

I'll show you a full example on both Linux and Windows. We will:

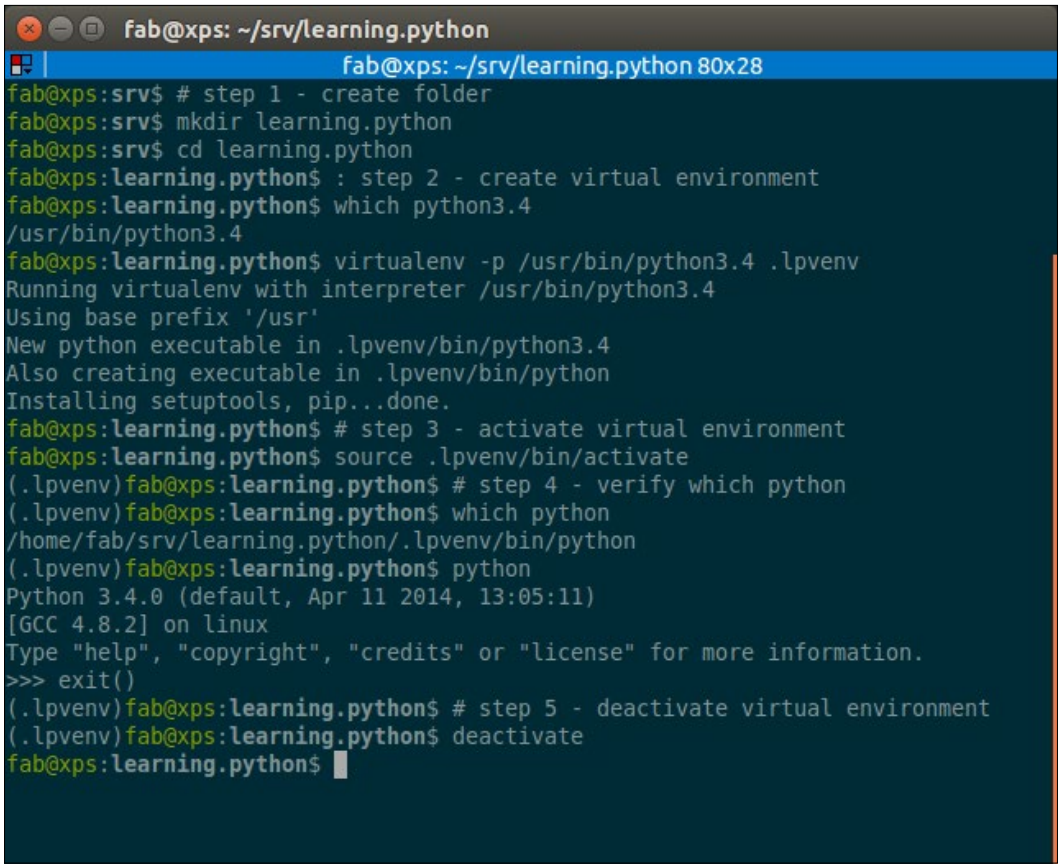
1. Create a folder named `learning.python` under your project root (which in my case is a folder called `srv`, in my home folder). Please adapt the paths according to the setup you fancy on your box.
2. Within the `learning.python` folder, we will create a virtual environment called `.lpvenv`.

Some developers prefer to call all virtual environments using the same name (for example, `.venv`). This way they can run scripts against any virtualenv by just knowing the name of the project they dwell in. This is a very common technique that I use as well. The dot in `.venv` is because in Linux/Mac prepending a name with a dot makes that file or folder invisible.

3. After creating the virtual environment, we will activate it (this is slightly different between Linux, Mac, and Windows).
4. Then, we'll make sure that we are running the desired Python version (3.4.*) by running the Python interactive shell.
5. Finally, we will deactivate the virtual environment using the deactivate command.

These five simple steps will show you all you have to do to start and use a project.

Here's an example of how those steps might look like on Linux (commands that start with a `#` are comments):



Notice that I had to explicitly tell virtualenv to use the Python 3.4 interpreter because on my box Python 2.7 is the default one. Had I not done that, I would have had a virtual environment with Python 2.7 instead of Python 3.4.

You can combine the two instructions for *step 2* in one single command like this:

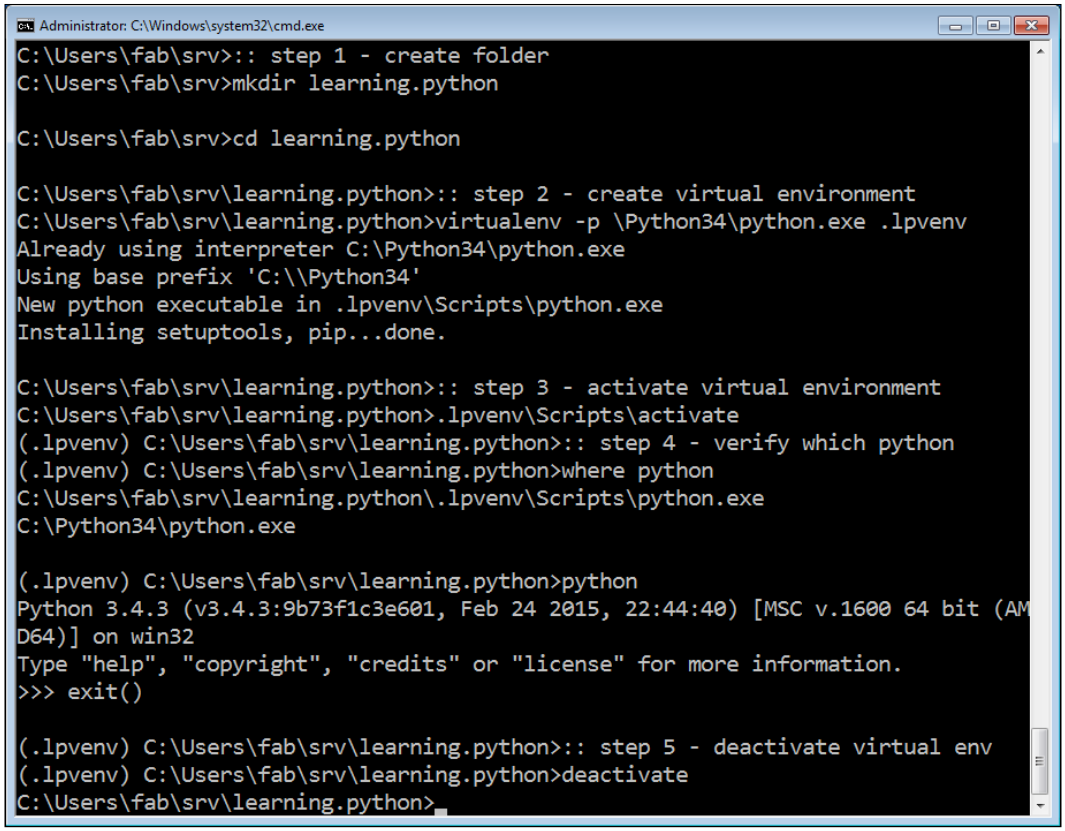
```
$ virtualenv -p $( which python3.4 ) .lpvenv
```

I preferred to be explicitly verbose in this instance, to help you understand each bit of the procedure.

Another thing to notice is that in order to activate a virtual environment, we need to run the `/bin/activate` script, which needs to be sourced (when a script is "sourced", it means that its effects stick around when it's done running). This is very important. Also notice how the prompt changes after we activate the virtual environment, showing its name on the left (and how it disappears when we deactivate). In Mac OS, the steps are the same so I won't repeat them here.

Now let's have a look at how we can achieve the same result in Windows. You will probably have to play around a bit, especially if you have a different Windows or Python version than I'm using here. This is all good experience though, so try and think positively at the initial struggle that every coder has to go through in order to get things going.

Here's how it should look on Windows (commands that start with `::` are comments):



Notice there are a few small differences from the Linux version. Apart from the commands to create and navigate the folders, one important difference is how you activate your virtualenv. Also, in Windows there is no `which` command, so we used the `where` command.

At this point, you should be able to create and activate a virtual environment. Please try and create another one without me guiding you, get acquainted to this procedure because it's something that you will always be doing: *we never work system-wide with Python*, remember? It's extremely important.

So, with the scaffolding out of the way, we're ready to talk a bit more about Python and how you can use it. Before we do it though, allow me to spend a few words about the console.

Your friend, the console

In this era of GUIs and touchscreen devices, it seems a little ridiculous to have to resort to a tool such as the console, when everything is just about one click away.

But the truth is every time you remove your right hand from the keyboard (or the left one, if you're a lefty) to grab your mouse and move the cursor over to the spot you want to click, you're losing time. Getting things done with the console, counter-intuitively as it may be, results in higher productivity and speed. I know, you have to trust me on this.

Speed and productivity are important and personally, I have nothing against the mouse, but there is another very good reason for which you may want to get well acquainted with the console: when you develop code that ends up on some server, the console might be the only available tool. If you make friends with it, I promise you, you will never get lost when it's of utmost importance that you don't (typically, when the website is down and you have to investigate very quickly what's going on).

So it's really up to you. If you're in doubt, please grant me the benefit of the doubt and give it a try. It's easier than you think, and you'll never regret it. There is nothing more pitiful than a good developer who gets lost within an SSH connection to a server because they are used to their own custom set of tools, and only to that.

Now, let's get back to Python.

How you can run a Python program

There are a few different ways in which you can run a Python program.

Running Python scripts

Python can be used as a scripting language. In fact, it always proves itself very useful. Scripts are files (usually of small dimensions) that you normally execute to do something like a task. Many developers end up having their own arsenal of tools that they fire when they need to perform a task. For example, you can have scripts to parse data in a format and render it into another different format. Or you can use a script to work with files and folders. You can create or modify configuration files, and much more. Technically, there is not much that cannot be done in a script.

It's quite common to have scripts running at a precise time on a server. For example, if your website database needs cleaning every 24 hours (for example, the table that stores the user sessions, which expire pretty quickly but aren't cleaned automatically), you could set up a cron job that fires your script at 3:00 A.M. every day.

According to Wikipedia, the software utility Cron is a time-based job scheduler in Unix-like computer operating systems. People who set up and maintain software environments use cron to schedule jobs (commands or shell scripts) to run periodically at fixed times, dates, or intervals.

I have Python scripts to do all the menial tasks that would take me minutes or more to do manually, and at some point, I decided to automate. For example, I have a laptop that doesn't have a *Fn* key to toggle the touchpad on and off. I find this very annoying, and I don't want to go clicking about through several menus when I need to do it, so I wrote a small script that is smart enough to tell my system to toggle the touchpad active state, and now I can do it with one simple click from my launcher. Priceless.

We'll devote half of *Chapter 8*, *The Edges – GUIs and Scripts* on scripting with Python.

Running the Python interactive shell

Another way of running Python is by calling the interactive shell. This is something we already saw when we typed `python` on the command line of our console.

So open a console, activate your virtual environment (which by now should be second nature to you, right?), and type `python`. You will be presented with a couple of lines that should look like this (if you are on Linux):

```
Python 3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
```

Those `>>>` are the prompt of the shell. They tell you that Python is waiting for you to type something. If you type a simple instruction, something that fits in one line, that's all you'll see. However, if you type something that requires more than one line of code, the shell will change the prompt to `...`, giving you a visual clue that you're typing a multiline statement (or anything that would require more than one line of code).

Go on, try it out, let's do some basic maths:

```
>>> 2 + 4
6
>>> 10 / 4
2.5
>>> 2 ** 1024
179769313486231590772930519078902473361797697894230657273430081157732675805500963132708477322407536021120113879871393357658789768814416622492847430639474124377767893424865485276302219601246094119453082952085005768838150682342462881473913110540827237163350510684586298239947245938479716304835356329624224137216
```

The last operation is showing you something incredible. We raise 2 to the power of 1024, and Python is handling this task with no trouble at all. Try to do it in Java, C++, or C#. It won't work, unless you use special libraries to handle such big numbers.

I use the interactive shell every day. It's extremely useful to debug very quickly, for example, to check if a data structure supports an operation. Or maybe to inspect or run a piece of code.

When you use Django (a web framework), the interactive shell is coupled with it and allows you to work your way through the framework tools, to inspect the data in the database, and many more things. You will find that the interactive shell will soon become one of your dearest friends on the journey you are embarking on.

Another solution, which comes in a much nicer graphic layout, is to use **IDLE** (**Integrated DeveLopment Environment**). It's quite a simple IDE, which is intended mostly for beginners. It has a slightly larger set of capabilities than the naked interactive shell you get in the console, so you may want to explore it. It comes for free in the Windows Python installer and you can easily install it in any other system. You can find information about it on the Python website.

Guido Van Rossum named Python after the British comedy group Monty Python, so it's rumored that the name IDLE has been chosen in honor of Erik Idle, one of Monty Python's founding members.

Running Python as a service

Apart from being run as a script, and within the boundaries of a shell, Python can be coded and run as proper software. We'll see many examples throughout the book about this mode. And we'll understand more about it in a moment, when we'll talk about how Python code is organized and run.

Running Python as a GUI application

Python can also be run as a **GUI** (**Graphical User Interface**). There are several frameworks available, some of which are cross-platform and some others are platform-specific. In *Chapter 8*, *The Edges – GUIs and Scripts*, we'll see an example of a GUI application created using *Tkinter*, which is an object-oriented layer that lives on top of **Tk** (Tkinter means Tk Interface).

Tk is a graphical user interface toolkit that takes desktop application development to a higher level than the conventional approach. It is the standard GUI for **Tcl** (**Tool Command Language**), but also for many other dynamic languages and can produce rich native applications that run seamlessly under Windows, Linux, Mac OS X, and more.

Tkinter comes bundled with Python, therefore it gives the programmer easy access to the GUI world, and for these reasons, I have chosen it to be the framework for the GUI examples that I'll present in this book.

Among the other GUI frameworks, we find that the following are the most widely used:

- PyQt
- wxPython
- PyGtk

Describing them in detail is outside the scope of this book, but you can find all the information you need on the Python website in the *GUI Programming* section. If GUIs are what you're looking for, remember to choose the one you want according to some principles. Make sure they:

- Offer all the features you may need to develop your project
- Run on all the platforms you may need to support
- Rely on a community that is as wide and active as possible
- Wrap graphic drivers/tools that you can easily install/access

How is Python code organized

Let's talk a little bit about how Python code is organized. In this paragraph, we'll start going down the rabbit hole a little bit more and introduce a bit more technical names and concepts.

Starting with the basics, how is Python code organized? Of course, you write your code into files. When you save a file with the extension `.py`, that file is said to be a Python module.

If you're on Windows or Mac, which typically hide file extensions to the user, please make sure you change the configuration so that you can see the complete name of the files. This is not strictly a requirement, but a hearty suggestion.

It would be impractical to save all the code that it is required for software to work within one single file. That solution works for *scripts*, which are usually not longer than a few hundred lines (and often they are quite shorter than that).

A complete Python application can be made of hundreds of thousands of lines of code, so you will have to scatter it through different modules. Better, but not nearly good enough. It turns out that even like this it would still be impractical to work with the code. So Python gives you another structure, called **package**, which allows you to group modules together. A package is nothing more than a folder, which must contain a special file, `__init__.py` that doesn't need to hold any code but whose presence is required to tell Python that the folder is not just some folder, but it's actually a package (note that as of Python 3.3 `__init__.py` is not strictly required any more).

As always, an example will make all of this much clearer. I have created an example structure in my book project, and when I type in my Linux console:

```
$ tree -v example
```

I get a tree representation of the contents of the `ch1/example` folder, which holds the code for the examples of this chapter. Here's how a structure of a real simple application could look like:

```
example/
├── core.py
├── run.py
└── util
    ├── __init__.py
    ├── db.py
    ├── math.py
    └── network.py
```

You can see that within the root of this example, we have two modules, `core.py` and `run.py`, and one package: `util`. Within `core.py`, there may be the core logic of our application. On the other hand, within the `run.py` module, we can probably find the logic to start the application. Within the `util` package, I expect to find various utility tools, and in fact, we can guess that the modules there are called by the type of tools they hold: `db.py` would hold tools to work with databases, `math.py` would of course hold mathematical tools (maybe our application deals with financial data), and `network.py` would probably hold tools to send/receive data on networks.

As explained before, the `__init__.py` file is there just to tell Python that `util` is a package and not just a mere folder.

Had this software been organized within modules only, it would have been much harder to infer its structure. I put a *module only* example under the `ch1/files_only` folder, see it for yourself:

```
$ tree -v files_only
```

This shows us a completely different picture:

```
files_only/
├── core.py
├── db.py
├── math.py
├── network.py
└── run.py
```

It is a little harder to guess what each module does, right? Now, consider that this is just a simple example, so you can guess how much harder it would be to understand a real application if we couldn't organize the code in packages and modules.

How do we use modules and packages

When a developer is writing an application, it is very likely that they will need to apply the same piece of logic in different parts of it. For example, when writing a parser for the data that comes from a form that a user can fill in a web page, the application will have to validate whether a certain field is holding a number or not. Regardless of how the logic for this kind of validation is written, it's very likely that it will be needed in more than one place. For example in a poll application, where the user is asked many question, it's likely that several of them will require a numeric answer. For example:

- What is your age
- How many pets do you own
- How many children do you have
- How many times have you been married

It would be very bad practice to copy paste (or, more properly said: duplicate) the validation logic in every place where we expect a numeric answer. This would violate the **DRY** (**Don't Repeat Yourself**) principle, which states that you should never repeat the same piece of code more than once in your application. I feel the need to stress the importance of this principle: *you should never repeat the same piece of code more than once in your application* (got the irony?).

There are several reasons why repeating the same piece of logic can be very bad, the most important ones being:

- There could be a bug in the logic, and therefore, you would have to correct it in every place that logic is applied.
- You may want to amend the way you carry out the validation, and again you would have to change it in every place it is applied.
- You may forget to fix/amend a piece of logic because you missed it when searching for all its occurrences. This would leave wrong/inconsistent behavior in your application.
- Your code would be longer than needed, for no good reason.

Python is a wonderful language and provides you with all the tools you need to apply all the coding best practices. For this particular example, we need to be able to reuse a piece of code. To be able to reuse a piece of code, we need to have a construct that will hold the code for us so that we can call that construct every time we need to repeat the logic inside it. That construct exists, and it's called **function**.

I'm not going too deep into the specifics here, so please just remember that a function is a block of organized, reusable code which is used to perform a task. Functions can assume many forms and names, according to what kind of environment they belong to, but for now this is not important. We'll see the details when we are able to appreciate them, later on, in the book. Functions are the building blocks of modularity in your application, and they are almost indispensable (unless you're writing a super simple script, you'll use functions all the time). We'll explore functions in *Chapter 4*, *Functions, the Building Blocks of Code*.

Python comes with a very extensive library, as I already said a few pages ago. Now, maybe it's a good time to define what a library is: a **library** is a collection of functions and objects that provide functionalities that enrich the abilities of a language.

For example, within Python's `math` library we can find a plethora of functions, one of which is the `factorial` function, which of course calculates the factorial of a number.

In mathematics, the **factorial** of a non-negative integer number *N*, denoted as *N!*, is defined as the product of all positive integers less than or equal to N. For example, the factorial of 5 is calculated as:

```
5! = 5 * 4 * 3 * 2 * 1 = 120
```

The factorial of `0` is `0! = 1`, to respect the convention for an empty product.

So, if you wanted to use this function in your code, all you would have to do is to import it and call it with the right input values. Don't worry too much if input values and the concept of calling is not very clear for now, please just concentrate on the import part.

We use a library by importing what we need from it, and then we use it.

In Python, to calculate the factorial of number 5, we just need the following code:

```
>>> from math import factorial
>>> factorial(5)
120
```

Whatever we type in the shell, if it has a printable representation, will be printed on the console for us (in this case, the result of the function call: 120).

So, let's go back to our example, the one with `core.py`, `run.py`, `util`, and so on.

In our example, the package `util` is our utility library. Our custom utility belt that holds all those reusable tools (that is, functions), which we need in our application. Some of them will deal with databases (`db.py`), some with the network (`network.py`), and some will perform mathematical calculations (`math.py`) that are outside the scope of Python's standard `math` library and therefore, we had to code them for ourselves.

We will see in detail how to import functions and use them in their dedicated chapter. Let's now talk about another very important concept: Python's execution model.

Python's execution model

In this paragraph, I would like to introduce you to a few very important concepts, such as scope, names, and namespaces. You can read all about Python's execution model in the official Language reference, of course, but I would argue that it is quite technical and abstract, so let me give you a less formal explanation first.

Names and namespaces

Say you are looking for a book, so you go to the library and ask someone for the book you want to fetch. They tell you something like "second floor, section X, row three". So you go up the stairs, look for section X, and so on.

It would be very different to enter a library where all the books are piled together in random order in one big room. No floors, no sections, no rows, no order. Fetching a book would be extremely hard.

When we write code we have the same issue: we have to try and organize it so that it will be easy for someone who has no prior knowledge about it to find what they're looking for. When software is structured correctly, it also promotes code reuse. On the other hand, disorganized software is more likely to expose scattered pieces of duplicated logic.

First of all, let's start with the book. We refer to a book by its title and in Python lingo, that would be a name. Python names are the closest abstraction to what other languages call variables. Names basically refer to objects and are introduced by name binding operations. Let's make a quick example (notice that anything that follows a `#` is a comment):

```
>>> n = 3  # integer number
>>> address = "221b Baker Street, NW1 6XE, London"  # S. Holmes
>>> employee = {
...     'age': 45,
...     'role': 'CTO',
...     'SSN': 'AB1234567',
... }
>>> # let's print them
>>> n
3
>>> address
'221b Baker Street, NW1 6XE, London'
>>> employee
```

```
{'role': 'CTO', 'SSN': 'AB1234567', 'age': 45}
>>> # what if I try to print a name I didn't define?
>>> other_name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'other_name' is not defined
```

We defined three objects in the preceding code (do you remember what are the three features every Python object has?):

- An integer number `n` (type: `int`, value: `3`)
- A string `address` (type: `str`, value: Sherlock Holmes' address)
- A dictionary `employee` (type: `dict`, value: a dictionary which holds three key/value pairs)

Don't worry, I know you're not supposed to know what a dictionary is. We'll see in the next chapter that it's the king of Python data structures.

Have you noticed that the prompt changed from `>>>` to `...` when I typed in the definition of employee? That's because the definition spans over multiple lines.

So, what are `n`, `address` and `employee`? They are **names**. Names that we can use to retrieve data within our code. They need to be kept somewhere so that whenever we need to retrieve those objects, we can use their names to fetch them. We need some space to hold them, hence: namespaces!

A **namespace** is therefore a mapping from names to objects. Examples are the set of built-in names (containing functions that are always accessible for free in any Python program), the global names in a module, and the local names in a function. Even the set of attributes of an object can be considered a namespace.

The beauty of namespaces is that they allow you to define and organize your names with clarity, without overlapping or interference. For example, the namespace associated with that book we were looking for in the library can be used to import the book itself, like this:

```
from library.second_floor.section_x.row_three import book
```

We start from the `library` namespace, and by means of the dot (`.`) operator, we walk into that namespace. Within this namespace, we look for `second_floor`, and again we walk into it with the `.` operator. We then walk into `section_x`, and finally within the last namespace, `row_three`, we find the name we were looking for: `book`.

Walking through a namespace will be clearer when we'll be dealing with real code examples. For now, just keep in mind that namespaces are places where names are associated to objects.

There is another concept, which is closely related to that of a namespace, which I'd like to briefly talk about: the **scope**.

Scopes

According to Python's documentation, *a scope is a textual region of a Python program, where a namespace is directly accessible*. Directly accessible means that when you're looking for an unqualified reference to a name, Python tries to find it in the namespace.

Scopes are determined statically, but actually during runtime they are used dynamically. This means that by inspecting the source code you can tell what the scope of an object is, but this doesn't prevent the software to alter that during runtime. There are four different scopes that Python makes accessible (not necessarily all of them present at the same time, of course):

- The **local** scope, which is the innermost one and contains the local names.
- The **enclosing** scope, that is, the scope of any enclosing function. It contains non-local names and also non-global names.
- The **global** scope contains the global names.
- The **built-in** scope contains the built-in names. Python comes with a set of functions that you can use in a off-the-shelf fashion, such as `print`, `all`, `abs`, and so on. They live in the built-in scope.

The rule is the following: when we refer to a name, Python starts looking for it in the current namespace. If the name is not found, Python continues the search to the enclosing scope and this continue until the built-in scope is searched. If a name hasn't been found after searching the built-in scope, then Python raises a `NameError` **exception**, which basically means that the name hasn't been defined (you saw this in the preceding example).

The order in which the namespaces are scanned when looking for a name is therefore: **local**, **enclosing**, **global**, **built-in** (**LEGB**).

This is all very theoretical, so let's see an example. In order to show you Local and Enclosing namespaces, I will have to define a few functions. Don't worry if you are not familiar with their syntax for the moment, we'll study functions in *Chapter 4*, *Functions, the Building Blocks of Code*. Just remember that in the following code, when you see `def`, it means I'm defining a function.

```
scopes1.py
    # Local versus Global

    # we define a function, called local
    def local():
        m = 7
        print(m)

    m = 5
    print(m)

    # we call, or `execute` the function local
    local()
```

In the preceding example, we define the same name `m`, both in the global scope and in the local one (the one defined by the function local). When we execute this program with the following command (have you activated your virtualenv?):

```
$ python scopes1.py
```

We see two numbers printed on the console: `5` and `7`.

What happens is that the Python interpreter parses the file, top to bottom. First, it finds a couple of comment lines, which are skipped, then it parses the definition of the function `local`. When called, this function does two things: it sets up a name to an object representing number 7 and prints it. The Python interpreter keeps going and it finds another name binding. This time the binding happens in the global scope and the value is 5. The next line is a call to the `print` function, which is executed (and so we get the first value printed on the console: `5`).

After this, there is a call to the function `local`. At this point, Python executes the function, so at this time, the binding `m = 7` happens and it's printed.

One very important thing to notice is that the part of the code that belongs to the definition of the function local is indented by four spaces on the right. Python in fact defines scopes by indenting the code. You walk into a scope by indenting and walk out of it by unindenting. Some coders use two spaces, others three, but the suggested number of spaces to use is four. It's a good measure to maximize readability. We'll talk more about all the conventions you should embrace when writing Python code later.

What would happen if we removed that `m = 7` line? Remember the LEGB rule. Python would start looking for `m` in the local scope (function `local`), and, not finding it, it would go to the next enclosing scope. The next one in this case is the global one because there is no enclosing function wrapped around `local`. Therefore, we would see two number `5` printed on the console. Let's actually see how the code would look like:

```
scopes2.py
    # Local versus Global

    def local():
        # m doesn't belong to the scope defined by the local function
        # so Python will keep looking into the next enclosing scope.
        # m is finally found in the global scope
        print(m, 'printing from the local scope')

    m = 5
    print(m, 'printing from the global scope')

    local()
```

Running `scopes2.py` will print this:

```
(.lpvenv)fab@xps:ch1$ python scopes2.py
5 printing from the global scope
5 printing from the local scope
```

As expected, Python prints `m` the first time, then when the function `local` is called, `m` isn't found in its scope, so Python looks for it following the LEGB chain until `m` is found in the global scope.

Let's see an example with an extra layer, the enclosing scope:

```
scopes3.py
    # Local, Enclosing and Global

    def enclosing_func():
        m = 13
        def local():
            # m doesn't belong to the scope defined by the local
            # function so Python will keep looking into the next
            # enclosing scope. This time m is found in the enclosing
            # scope
            print(m, 'printing from the local scope')

        # calling the function local
        local()
```

```
m = 5
print(m, 'printing from the global scope')

enclosing_func()
```

Running `scopes3.py` will print on the console:

```
(.lpvenv)fab@xps:ch1$ python scopes3.py
5 printing from the global scope
13 printing from the local scope
```

As you can see, the `print` instruction from the function `local` is referring to `m` as before. `m` is still not defined within the function itself, so Python starts walking scopes following the LEGB order. This time `m` is found in the enclosing scope.

Don't worry if this is still not perfectly clear for now. It will come to you as we go through the examples in the book. The *Classes* section of the Python tutorial (official documentation) has an interesting paragraph about scopes and namespaces. Make sure you read it at some point if you wish for a deeper understanding of the subject.

Before we finish off this chapter, I would like to talk a bit more about objects. After all, basically everything in Python is an object, so I think they deserve a bit more attention.

Object and classes

When I introduced objects in the *A proper introduction* section, I said that we use them to represent real-life objects. For example, we sell goods of any kind on the Web nowadays and we need to be able to handle, store, and represent them properly. But objects are actually so much more than that. Most of what you will ever do, in Python, has to do with manipulating objects.

So, without going too much into detail (we'll do that in *Chapter 6*, *Advanced Concepts – OOP, Decorators, and Iterators*), I want to give you the *in a nutshell* kind of explanation about classes and objects.

We've already seen that objects are Python's abstraction for data. In fact, everything in Python is an object. Numbers, strings (data structures that hold text), containers, collections, even functions. You can think of them as if they were boxes with at least three features: an ID (unique), a type, and a value.

But how do they come to life? How do we create them? How to we write our own custom objects? The answer lies in one simple word: classes.

Objects are, in fact, instances of classes. The beauty of Python is that classes are objects themselves, but let's not go down this road. It leads to one of the most advanced concepts of this language: **metaclasses**. We'll talk very briefly about them in *Chapter 6*, *Advanced Concepts – OOP, Decorators, and Iterators*. For now, the best way for you to get the difference between classes and objects, is by means of an example.

Say a friend tells you "I bought a new bike!" You immediately understand what she's talking about. Have you seen the bike? No. Do you know what color it is? Nope. The brand? Nope. Do you know anything about it? Nope. But at the same time, you know everything you need in order to understand what your friend meant when she told you she bought a new bike. You know that a bike has two wheels attached to a frame, a saddle, pedals, handlebars, brakes, and so on. In other words, even if you haven't seen the bike itself, you know the concept of bike. An abstract set of features and characteristics that together form something called bike.

In computer programming, that is called a **class**. It's that simple. Classes are used to create objects. In fact, objects are said to be **instances of classes**.

In other words, we all know what a bike is, we know the class. But then I have my own bike, which is an instance of the class bike. And my bike is an object with its own characteristics and methods. You have your own bike. Same class, but different instance. Every bike ever created in the world is an instance of the bike class.

Let's see an example. We will write a class that defines a bike and then we'll create two bikes, one red and one blue. I'll keep the code very simple, but don't fret if you don't understand everything about it; all you need to care about at this moment is to understand the difference between class and object (or instance of a class):

bike.py

```
# let's define the class Bike
class Bike:
    def __init__(self, colour, frame_material):
        self.colour = colour
        self.frame_material = frame_material

    def brake(self):
        print("Braking!")
```

```
# let's create a couple of instances
red_bike = Bike('Red', 'Carbon fiber')
blue_bike = Bike('Blue', 'Steel')

# let's inspect the objects we have, instances of the Bike class.
print(red_bike.colour)  # prints: Red
print(red_bike.frame_material)  # prints: Carbon fiber
print(blue_bike.colour)  # prints: Blue
print(blue_bike.frame_material)  #  prints: Steel

# let's brake!
red_bike.brake()  # prints: Braking!
```

I hope by now I don't need to tell you to run the file every time, right? The filename is indicated in the first line of the code block. Just run `$ python filename`, and you'll be fine.

So many interesting things to notice here. First things first; the definition of a class happens with the `class` statement (highlighted in the code). Whatever code comes after the `class` statement, and is indented, is called the body of the class. In our case, the last line that belongs to the class definition is the `print("Braking!")` one.

After having defined the class we're ready to create instances. You can see that the class body hosts the definition of two methods. A method is basically (and simplistically) a function that belongs to a class.

The first method, `__init__` is an **initializer**. It uses some Python magic to set up the objects with the values we pass when we create it.

Every method that has leading and trailing double underscore, in Python, is called **magic method**. Magic methods are used by Python for a multitude of different purposes, hence it's never a good idea to name a custom method using two leading and trailing underscores. This naming convention is best left to Python.

The other method we defined, `brake`, is just an example of an additional method that we could call if we wanted to brake the bike. It contains just a `print` statement, of course, it's an example.

We created two bikes then. One has red color and a carbon fiber frame, and the other one has blue color and steel frame. We pass those values upon creation. After creation, we print out the color property and frame type of the red bike, and the frame type of the blue one just as an example. We also call the `brake` method of the `red_bike`.

One last thing to notice. You remember I told you that the set of attributes of an object is considered to be a namespace? I hope it's clearer now, what I meant. You see that by getting to the `frame_type` property through different namespaces (`red_bike`, `blue_bike`) we obtain different values. No overlapping, no confusion.

The dot (`.`) operator is of course the means we use to walk into a namespace, in the case of objects as well.

Guidelines on how to write good code

Writing good code is not as easy as it seems. As I already said before, good code exposes a long list of qualities that is quite hard to put together. Writing good code is, to some extent, an art. Regardless of where on the path you will be happy to settle, there is something that you can embrace which will make your code instantly better: **PEP8**.

According to Wikipedia:

> *"Python's development is conducted largely through the Python Enhancement Proposal (PEP) process. The PEP process is the primary mechanism for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python."*

Among all the PEPs, probably the most famous one is PEP8. It lays out a simple but effective set of guidelines to define Python aesthetic so that we write beautiful Python code. If you take one suggestion out of this chapter, please let it be this: use it. Embrace it. You will thank me later.

Coding today is no longer a check-in/check-out business. Rather, it's more of a social effort. Several developers collaborate to a piece of code through tools like git and mercurial, and the result is code that is fathered by many different hands.

Git and Mercurial are probably the most used distributed revision control systems today. They are essential tools designed to help teams of developers collaborate on the same software.

These days, more than ever, we need to have a consistent way of writing code, so that readability is maximized. When all developers of a company abide with PEP8, it's not uncommon for any of them landing on a piece of code to think they wrote it themselves. It actually happens to me all the time (I always forget the code I write).

This has a tremendous advantage: when you read code that you could have written yourself, you read it easily. Without a convention, every coder would structure the code the way they like most, or simply the way they were taught or are used to, and this would mean having to interpret every line according to someone else's style. It would mean having to lose much more time just trying to understand it. Thanks to PEP8, we can avoid this. I'm such a fan of it that I won't sign off a code review if the code doesn't respect it. So please take the time to study it, it's very important.

In the examples of this book, I will try to respect it as much as I can. Unfortunately, I don't have the luxury of 79 characters (which is the maximum line length suggested by PEP*), and I will have to cut down on blank lines and other things, but I promise you I'll try to layout my code so that it's as readable as possible.

The Python culture

Python has been adopted widely in all coding industries. It's used by many different companies for many different purposes, and it's also used in education (it's an excellent language for that purpose, because of its many qualities and the fact that it's easy to learn).

One of the reasons Python is so popular today is that the community around it is vast, vibrant, and full of brilliant people. Many events are organized all over the world, mostly either around Python or its main web framework, Django.

Python is open, and very often so are the minds of those who embrace it. Check out the community page on the Python website for more information and get involved!

There is another aspect to Python which revolves around the notion of being **Pythonic**. It has to do with the fact that Python allows you to use some idioms that aren't found elsewhere, at least not in the same form or easiness of use (I feel quite claustrophobic when I have to code in a language which is not Python now).

Anyway, over the years, this concept of being Pythonic has emerged and, the way I understand it, is something along the lines of *doing things the way they are supposed to be done in Python*.

To help you understand a little bit more about Python's culture and about being Pythonic, I will show you the *Zen of Python*. A lovely Easter egg that is very popular. Open up a Python console and type `import this`. What follows is the result of this line:

```
>>> import this
The Zen of Python, by Tim Peters
```

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

There are two levels of reading here. One is to consider it as a set of guidelines that have been put down in a fun way. The other one is to keep it in mind, and maybe read it once in a while, trying to understand how it refers to something deeper. Some Python characteristics that you will have to understand deeply in order to write Python the way it's supposed to be written. Start with the fun level, and then dig deeper. Always dig deeper.

A note on the IDEs

Just a few words about **Integrated Development Environments** (**IDEs**). To follow the examples in this book you don't need one, any text editor will do fine. If you want to have more advanced features such as syntax coloring and auto completion, you will have to fetch yourself an IDE. You can find a comprehensive list of open source IDEs (just Google "python ides") on the Python website. I personally use Sublime Text editor. It's free to try out and it costs just a few dollars. I have tried many IDEs in my life, but this is the one that makes me most productive.

Two extremely important pieces of advice:

- Whatever IDE you will chose to use, try to learn it well so that you can exploit its strengths, but *don't depend on it*. Exercise yourself to work with VIM (or any other text editor) once in a while, learn to be able to do some work on any platform, with any set of tools.
- Whatever text editor/IDE you will use, when it comes to writing Python, *indentation is four spaces*. Don't use tabs, don't mix them with spaces. Use four spaces, not two, not three, not five. Just use four. The whole world works like that, and you don't want to become an outcast because you were fond of the three-space layout.

Summary

In this chapter, we started to explore the world of programming and that of Python. We've barely scratched the surface, just a little, touching concepts that will be discussed later on in the book in greater detail.

We talked about Python's main features, who is using it and for what, and what are the different ways in which we can write a Python program.

In the last part of the chapter, we flew over the fundamental notions of namespace, scope, class, and object. We also saw how Python code can be organized using modules and packages.

On a practical level, we learned how to install Python on our system, how to make sure we have the tools we need, pip and virtualenv, and we also created and activated our first virtual environment. This will allow us to work in a self-contained environment without the risk of compromising the Python system installation.

Now you're ready to start this journey with me. All you need is enthusiasm, an activated virtual environment, this book, your fingers, and some coffee.

Try to follow the examples, I'll keep them simple and short. If you put them under your fingertips, you will retain them much better than if you just read them.

In the next chapter, we will explore Python's rich set of built-in data types. There's much to cover and much to learn!

2
Built-in Data Types

"Data! Data! Data!" he cried impatiently. "I can't make bricks without clay."

- Sherlock Holmes - The Adventure of the Copper Beeches

Everything you do with a computer is managing data. Data comes in many different shapes and flavors. It's the music you listen, the movie you stream, the PDFs you open. Even the chapter you're reading at this very moment is just a file, which is data.

Data can be simple, an integer number to represent an age, or complex, like an order placed on a website. It can be about a single object or about a collection of them.

Data can even be about data, that is, metadata. Data that describes the design of other data structures or data that describes application data or its context.

In Python, *objects are abstraction for data*, and Python has an amazing variety of data structures that you can use to represent data, or combine them to create your own custom data. Before we delve into the specifics, I want you to be very clear about objects in Python, so let's talk a little bit more about them.

Everything is an object

As we already said, everything in Python is an object. But what really happens when you type an instruction like `age = 42` in a Python module?

If you go to `http://pythontutor.com/`, you can type that instruction into a text box and get its visual representation. Keep this website in mind, it's very useful to consolidate your understanding of what goes on behind the scenes.

So, what happens is that an object is created. It gets an `id`, the `type` is set to `int` (integer number), and the `value` to `42`. A name `age` is placed in the global namespace, pointing to that object. Therefore, whenever we are in the global namespace, after the execution of that line, we can retrieve that object by simply accessing it through its name: `age`.

If you were to move house, you would put all the knives, forks, and spoons in a box and label it cutlery. Can you see it's exactly the same concept? Here's a screenshot of how it may look like (you may have to tweak the settings to get to the same view):

So, for the rest of this chapter, whenever you read something such as `name = some_value`, think of a name placed in the namespace that is tied to the scope in which the instruction was written, with a nice arrow pointing to an object that has an `id`, a `type`, and a `value`. There is a little bit more to say about this mechanism, but it's much easier to talk about it over an example, so we'll get back to this later.

Mutable or immutable? That is the question

A first fundamental distinction that Python makes on data is about whether or not the value of an object changes. If the value can change, the object is called **mutable**, while if the value cannot change, the object is called **immutable**.

It is very important that you understand the distinction between mutable and immutable because it affects the code you write, so here's a question:

```
>>> age = 42
>>> age
42
```

```
>>> age = 43  #A
>>> age
43
```

In the preceding code, on the line `#A`, have I changed the value of age? Well, no. But now it's 43 (I hear you say...). Yes, it's 43, but 42 was an integer number, of the type `int`, which is immutable. So, what happened is really that on the first line, `age` is a name that is set to point to an `int` object, whose value is 42. When we type `age = 43`, what happens is that another object is created, of the type `int` and value 43 (also, the `id` will be different), and the name `age` is set to point to it. So, we didn't change that 42 to 43. We actually just pointed `age` to a different location: the new `int` object whose value is 43. Let's see the same code also printing the IDs:

```
>>> age = 42
>>> id(age)
10456352
>>> age = 43
>>> id(age)
10456384
```

Notice that we print the IDs by calling the built-in `id` function. As you can see, they are different, as expected. Bear in mind that `age` points to one object at a time: 42 first, then 43. Never together.

Now, let's see the same example using a mutable object. For this example, let's just use a `Person` object, that has a property `age`:

```
>>> fab = Person(age=39)
>>> fab.age
39
>>> id(fab)
139632387887456
>>> fab.age = 29  # I wish!
>>> id(fab)
139632387887456  # still the same id
```

In this case, I set up an object `fab` whose `type` is `Person` (a custom class). On creation, the object is given the `age` of 39. I'm printing it, along with the object `id`, right afterwards. Notice that, even after I change `age` to be 29, the ID of `fab` stays the same (while the ID of `age` has changed, of course). Custom objects in Python are mutable (unless you code them not to be). Keep this concept in mind, it's very important. I'll remind you about it through the rest of the chapter.

Numbers

Let's start by exploring Python's built-in data types for numbers. Python was designed by a man with a master's degree in mathematics and computer science, so it's only logical that it has amazing support for numbers.

Numbers are immutable objects.

Integers

Python integers have unlimited range, subject only to the available virtual memory. This means that it doesn't really matter how big a number you want to store: as long as it can fit in your computer's memory, Python will take care of it. Integer numbers can be positive, negative, and 0 (zero). They support all the basic mathematical operations, as shown in the following example:

```
>>> a = 12
>>> b = 3
>>> a + b  # addition
15
>>> b - a  # subtraction
-9
>>> a // b  # integer division
4
>>> a / b  # true division
4.0
>>> a * b  # multiplication
36
>>> b ** a  # power operator
531441
>>> 2 ** 1024  # a very big number, Python handles it gracefully
179769313486231590772930519078902473361797697894230657273430081157732675805500963132708477322407536021120113879871393357658789768814416622492847430639474124377767893424865485276302219601246094119453082952085005768838150682342462881473913110540827237163350510684586298239947245938479716304835356329624224137216
```

The preceding code should be easy to understand. Just notice one important thing: Python has two division operators, one performs the so-called **true division** (`/`), which returns the quotient of the operands, and the other one, the so-called **integer division** (`//`), which returns the *floored* quotient of the operands. See how that is different for positive and negative numbers:

```
>>> 7 / 4  # true division
1.75
>>> 7 // 4  # integer division, flooring returns 1
1
>>> -7 / 4  # true division again, result is opposite of previous
-1.75
>>> -7 // 4  # integer div., result not the opposite of previous
-2
```

This is an interesting example. If you were expecting a `-1` on the last line, don't feel bad, it's just the way Python works. The result of an integer division in Python is always rounded towards minus infinity. If instead of flooring you want to truncate a number to an integer, you can use the built-in `int` function, like shown in the following example:

```
>>> int(1.75)
1
>>> int(-1.75)
-1
```

Notice that truncation is done towards 0.

There is also an operator to calculate the remainder of a division. It's called modulo operator, and it's represented by a percent (`%`):

```
>>> 10 % 3  # remainder of the division 10 // 3
1
>>> 10 % 4  # remainder of the division 10 // 4
2
```

Booleans

Boolean algebra is that subset of algebra in which the values of the variables are the truth values: true and false. In Python, `True` and `False` are two keywords that are used to represent truth values. Booleans are a subclass of integers, and behave respectively like 1 and 0. The equivalent of the `int` class for Booleans is the `bool` class, which returns either `True` or `False`. Every built-in Python object has a value in the Boolean context, which means they basically evaluate to either `True` or `False` when fed to the `bool` function. We'll see all about this in *Chapter 3*, *Iterating and Making Decisions*.

Boolean values can be combined in Boolean expressions using the logical operators `and`, `or`, and `not`. Again, we'll see them in full in the next chapter, so for now let's just see a simple example:

```
>>> int(True)  # True behaves like 1
1
>>> int(False)  # False behaves like 0
0
>>> bool(1)  # 1 evaluates to True in a boolean context
True
>>> bool(-42)  # and so does every non-zero number
True
>>> bool(0)  # 0 evaluates to False
False
>>> # quick peak at the operators (and, or, not)
>>> not True
False
>>> not False
True
>>> True and True
True
>>> False or True
True
```

You can see that `True` and `False` are subclasses of integers when you try to add them. Python upcasts them to integers and performs addition:

```
>>> 1 + True
2
>>> False + 42
```

```
42
>>> 7 - True
6
```

Upcasting is a type conversion operation that goes from a subclass to its parent. In the example presented here, `True` and `False`, which belong to a class derived from the integer class, are converted back to integers when needed. This topic is about inheritance and will be explained in detail in *Chapter 6*, *Advanced Concepts – OOP, Decorators, and Iterators*.

Reals

Real numbers, or floating point numbers, are represented in Python according to the IEEE 754 double-precision binary floating-point format, which is stored in 64 bits of information divided into three sections: sign, exponent, and mantissa.

Quench your thirst for knowledge about this format on Wikipedia: `http://en.wikipedia.org/wiki/Double-precision_floating-point_format`

Usually programming languages give coders two different formats: single and double precision. The former taking up 32 bits of memory, and the latter 64. Python supports only the double format. Let's see a simple example:

```
>>> pi = 3.1415926536  # how many digits of PI can you remember?
>>> radius = 4.5
>>> area = pi * (radius ** 2)
>>> area
63.61725123519331
```

In the calculation of the area, I wrapped the `radius ** 2` within braces. Even though that wasn't necessary because the power operator has higher precedence than the multiplication one, I think the formula reads more easily like that.

The `sys.float_info` struct sequence holds information about how floating point numbers will behave on your system. This is what I see on my box:

```
>>> 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)
```

Let's make a few considerations here: we have 64 bits to represent float numbers. This means we can represent at most `2 ** 64 == 18,446,744,073,709,551,616` numbers with that amount of bits. Take a look at the `max` and `epsilon` value for the float numbers, and you'll realize it's impossible to represent them all. There is just not enough space so they are approximated to the closest representable number. You probably think that only extremely big or extremely small numbers suffer from this issue. Well, think again:

```
>>> 3 * 0.1 - 0.3  # this should be 0!!!
5.551115123125783e-17
```

What does this tell you? It tells you that double precision numbers suffer from approximation issues even when it comes to simple numbers like 0.1 or 0.3. Why is this important? It can be a big problem if you're handling prices, or financial calculations, or any kind of data that needs not to be approximated. Don't worry, Python gives you the **Decimal** type, which doesn't suffer from these issues, we'll see them in a bit.

Complex numbers

Python gives you complex numbers support out of the box. If you don't know what complex numbers are, you can look them up on the Web. They are numbers that can be expressed in the form *a* + *ib* where *a* and *b* are real numbers, and *i* (or *j* if you're an engineer) is the imaginary unit, that is, the square root of -1. *a* and *b* are called respectively the *real* and *imaginary* part of the number.

It's actually unlikely you'll be using them, unless you're coding something scientific. Let's see a small example:

```
>>> c = 3.14 + 2.73j
>>> c.real  # real part
3.14
>>> c.imag  # imaginary part
2.73
```

```
>>> c.conjugate()  # conjugate of A + Bj is A - Bj
(3.14-2.73j)
>>> c * 2  # multiplication is allowed
(6.28+5.46j)
>>> c ** 2  # power operation as well
(2.4067000000000007+17.1444j)
>>> d = 1 + 1j  # addition and subtraction as well
>>> c - d
(2.14+1.73j)
```

Fractions and decimals

Let's finish the tour of the number department with a look at fractions and decimals. Fractions hold a rational numerator and denominator in their lowest forms. Let's see a quick example:

```
>>> from fractions import Fraction
>>> Fraction(10, 6)  # mad hatter?
Fraction(5, 3)  # notice it's been reduced to lowest terms
>>> Fraction(1, 3) + Fraction(2, 3)  # 1/3 + 2/3 = 3/3 = 1/1
Fraction(1, 1)
>>> f = Fraction(10, 6)
>>> f.numerator
5
>>> f.denominator
3
```

Although they can be very useful at times, it's not that common to spot them in commercial software. Much easier instead, is to see decimal numbers being used in all those contexts where precision is everything, for example, scientific and financial calculations.

It's important to remember that arbitrary precision decimal numbers come at a price in performance, of course. The amount of data to be stored for each number is far greater than it is for fractions or floats as well as the way they are handled, which requires the Python interpreter much more work behind the scenes. Another interesting thing to know is that you can get and set the precision by accessing `decimal.getcontext().prec`.

Let's see a quick example with `Decimal` numbers:

```
>>> from decimal import Decimal as D  # rename for brevity
>>> D(3.14)  # pi, from float, so approximation issues
Decimal('3.140000000000000124344978758017532527446746826171875')
>>> D('3.14')  # pi, from a string, so no approximation issues
Decimal('3.14')
>>> D(0.1) * D(3) - D(0.3)  # from float, we still have the issue
Decimal('2.775557561565156540423631668E-17')
>>> D('0.1') * D(3) - D('0.3')  # from string, all perfect
Decimal('0.0')
```

Notice that when we construct a `Decimal` number from a `float`, it takes on all the approximation issues the `float` may come from. On the other hand, when the `Decimal` has no approximation issues, for example, when we feed an `int` or a `string` representation to the constructor, then the calculation has no quirky behavior. When it comes to money, use decimals.

This concludes our introduction to built-in numeric types, let's now see sequences.

Immutable sequences

Let's start with immutable sequences: strings, tuples, and bytes.

Strings and bytes

Textual data in Python is handled with **str** objects, more commonly known as strings. They are immutable sequences of **unicode code points**. Unicode code points can represent a character, but can also have other meanings, such as formatting data for example. Python, unlike other languages, doesn't have a `char` type, so a single character is rendered simply by a string of length 1. Unicode is an excellent way to handle data, and should be used for the internals of any application. When it comes to store textual data though, or send it on the network, you may want to encode it, using an appropriate encoding for the medium you're using. String literals are written in Python using single, double or triple quotes (both single or double). If built with triple quotes, a string can span on multiple lines. An example will clarify the picture:

```
>>> # 4 ways to make a string
>>> str1 = 'This is a string. We built it with single quotes.'
>>> str2 = "This is also a string, but built with double quotes."
```

```
>>> str3 = '''This is built using triple quotes,
... so it can span multiple lines.'''
>>> str4 = """This too
... is a multiline one
... built with triple double-quotes."""
>>> str4  #A
'This too\nis a multiline one\nbuilt with triple double-quotes.'
>>> print(str4)  #B
This too
is a multiline one
built with triple double-quotes.
```

In `#A` and `#B`, we print `str4`, first implicitly, then explicitly using the `print` function. A nice exercise would be to find out why they are different. Are you up to the challenge? (hint, look up the `str` function)

Strings, like any sequence, have a length. You can get this by calling the `len` function:

```
>>> len(str1)
49
```

Encoding and decoding strings

Using the `encode`/`decode` methods, we can encode unicode strings and decode bytes objects. **Utf-8** is a variable length character encoding, capable of encoding all possible unicode code points. It is the dominant encoding for the Web (and not only). Notice also that by adding a literal `b` in front of a string declaration, we're creating a *bytes* object.

```
>>> s = "This is üŋíc0de"  # unicode string: code points
>>> type(s)
<class 'str'>
>>> encoded_s = s.encode('utf-8')  # utf-8 encoded version of s
>>> encoded_s
b'This is \xc3\xbc\xc5\x8b\xc3\xadc0de'  # result: bytes object
>>> type(encoded_s)  # another way to verify it
<class 'bytes'>
>>> encoded_s.decode('utf-8')  # let's revert to the original
'This is üŋíc0de'
>>> bytes_obj = b"A bytes object"  # a bytes object
>>> type(bytes_obj)
<class 'bytes'>
```

Indexing and slicing strings

When manipulating sequences, it's very common to have to access them at one precise position (indexing), or to get a subsequence out of them (slicing). When dealing with immutable sequences, both operations are read-only.

While indexing comes in one form, a zero-based access to any position within the sequence, slicing comes in different forms. When you get a slice of a sequence, you can specify the `start` and `stop` positions, and the `step`. They are separated with a colon (`:`) like this: `my_sequence[start:stop:step]`. All the arguments are optional, `start` is inclusive, `stop` is exclusive. It's much easier to show an example, rather than explain them further in words:

```
>>> s = "The trouble is you think you have time."
>>> s[0]  # indexing at position 0, which is the first char
'T'
>>> s[5]  # indexing at position 5, which is the sixth char
'r'
>>> s[:4]  # slicing, we specify only the stop position
'The '
>>> s[4:]  # slicing, we specify only the start position
'trouble is you think you have time.'
>>> s[2:14]  # slicing, both start and stop positions
'e trouble is'
>>> s[2:14:3]  # slicing, start, stop and step (every 3 chars)
'erb '
>>> s[:]  # quick way of making a copy
'The trouble is you think you have time.'
```

Of all the lines, the last one is probably the most interesting. If you don't specify a parameter, Python will fill in the default for you. In this case, `start` will be the start of the string, `stop` will be the end of the sting, and `step` will be the default 1. This is an easy and quick way of obtaining a copy of the string `s` (same value, but different object). Can you find a way to get the reversed copy of a string using slicing? (don't look it up, find it for yourself)

Tuples

The last immutable sequence type we're going to see is the tuple. A **tuple** is a sequence of arbitrary Python objects. In a tuple, items are separated by commas. They are used everywhere in Python, because they allow for patterns that are hard to reproduce in other languages. Sometimes tuples are used implicitly, for example to set up multiple variables on one line, or to allow a function to return multiple different objects (usually a function returns one object only, in many other languages), and even in the Python console, you can use tuples implicitly to print multiple elements with one single instruction. We'll see examples for all these cases:

```
>>> t = ()  # empty tuple
>>> type(t)
<class 'tuple'>
>>> one_element_tuple = (42, )  # you need the comma!
>>> three_elements_tuple = (1, 3, 5)
>>> a, b, c = 1, 2, 3  # tuple for multiple assignment
>>> a, b, c  # implicit tuple to print with one instruction
(1, 2, 3)
>>> 3 in three_elements_tuple  # membership test
True
```

Notice that the membership operator `in` can also be used with lists, strings, dictionaries, and in general with collection and sequence objects.

Notice that to create a tuple with one item, we need to put that comma after the item. The reason is that without the comma that item is just itself wrapped in braces, kind of in a redundant mathematical expression. Notice also that on assignment, braces are optional so `my_tuple = 1, 2, 3` is the same as `my_tuple = (1, 2, 3)`.

One thing that tuple assignment allows us to do, is *one-line swaps*, with no need for a third temporary variable. Let's see first a more traditional way of doing it:

```
>>> a, b = 1, 2
>>> c = a  # we need three lines and a temporary var c
>>> a = b
>>> b = c
>>> a, b  # a and b have been swapped
(2, 1)
```

And now let's see how we would do it in Python:

```
>>> a, b = b, a  # this is the Pythonic way to do it
>>> a, b
(1, 2)
```

Take a look at the line that shows you the Pythonic way of swapping two values: do you remember what I wrote in *Chapter 1, Introduction and First Steps – Take a Deep Breath*. A Python program is typically one-fifth to one-third the size of equivalent Java or C++ code, and features like one-line swaps contribute to this. Python is elegant, where elegance in this context means also economy.

Because they are immutable, tuples can be used as keys for dictionaries (we'll see this shortly). The `dict` objects need keys to be immutable because if they could change, then the value they reference wouldn't be found any more (because the path to it depends on the key). If you are into data structures, you know how nice a feature this one is to have. To me, tuples are Python's built-in data that most closely represent a mathematical vector. This doesn't mean that this was the reason for which they were created though. Tuples usually contain an heterogeneous sequence of elements, while on the other hand lists are most of the times homogeneous. Moreover, tuples are normally accessed via unpacking or indexing, while lists are usually iterated over.

Mutable sequences

Mutable sequences differ from their immutable sisters in that they can be changed after creation. There are two mutable sequence types in Python: lists and byte arrays. I said before that the dictionary is the king of data structures in Python. I guess this makes the list its rightful queen.

Lists

Python lists are mutable sequences. They are very similar to tuples, but they don't have the restrictions due to immutability. Lists are commonly used to store collections of homogeneous objects, but there is nothing preventing you to store heterogeneous collections as well. Lists can be created in many different ways, let's see an example:

```
>>> []  # empty list
[]
>>> list()  # same as []
[]
>>> [1, 2, 3]  # as with tuples, items are comma separated
```

```
[1, 2, 3]
>>> [x + 5 for x in [2, 3, 4]]  # Python is magic
[7, 8, 9]
>>> list((1, 3, 5, 7, 9))  # list from a tuple
[1, 3, 5, 7, 9]
>>> list('hello')  # list from a string
['h', 'e', 'l', 'l', 'o']
```

In the previous example, I showed you how to create a list using different techniques. I would like you to take a good look at the line that says `Python is magic`, which I am not expecting you to fully understand at this point (unless you cheated and you're not a novice!). That is called a **list comprehension**, a very powerful functional feature of Python, which we'll see in detail in *Chapter 5, Saving Time and Memory*. I just wanted to make your mouth water at this point.

Creating lists is good, but the real fun comes when we use them, so let's see the main methods they gift us with:

```
>>> a = [1, 2, 1, 3]
>>> a.append(13)  # we can append anything at the end
>>> a
[1, 2, 1, 3, 13]
>>> a.count(1)  # how many `1` are there in the list?
2
>>> a.extend([5, 7])  # extend the list by another (or sequence)
>>> a
[1, 2, 1, 3, 13, 5, 7]
>>> a.index(13)  # position of `13` in the list (0-based indexing)
4
>>> a.insert(0, 17)  # insert `17` at position 0
>>> a
[17, 1, 2, 1, 3, 13, 5, 7]
>>> a.pop()  # pop (remove and return) last element
7
>>> a.pop(3)  # pop element at position 3
1
>>> a
[17, 1, 2, 3, 13, 5]
>>> a.remove(17)  # remove `17` from the list
>>> a
[1, 2, 3, 13, 5]
>>> a.reverse()  # reverse the order of the elements in the list
```

```
>>> a
[5, 13, 3, 2, 1]
>>> a.sort()  # sort the list
>>> a
[1, 2, 3, 5, 13]
>>> a.clear()  # remove all elements from the list
>>> a
[]
```

The preceding code gives you a roundup of list's main methods. I want to show you how powerful they are, using `extend` as an example. You can extend lists using any sequence type:

```
>>> a = list('hello')  # makes a list from a string
>>> a
['h', 'e', 'l', 'l', 'o']
>>> a.append(100)  # append 100, heterogeneous type
>>> a
['h', 'e', 'l', 'l', 'o', 100]
>>> a.extend((1, 2, 3))  # extend using tuple
>>> a
['h', 'e', 'l', 'l', 'o', 100, 1, 2, 3]
>>> a.extend('...')  # extend using string
>>> a
['h', 'e', 'l', 'l', 'o', 100, 1, 2, 3, '.', '.', '.']
```

Now, let's see what are the most common operations you can do with lists:

```
>>> a = [1, 3, 5, 7]
>>> min(a)  # minimum value in the list
1
>>> max(a)  # maximum value in the list
7
>>> sum(a)  # sum of all values in the list
16
>>> len(a)  # number of elements in the list
4
>>> b = [6, 7, 8]
>>> a + b  # `+` with list means concatenation
[1, 3, 5, 7, 6, 7, 8]
>>> a * 2  # `*` has also a special meaning
[1, 3, 5, 7, 1, 3, 5, 7]
```

The last two lines in the preceding code are quite interesting because they introduce us to a concept called **operator overloading**. In short, it means that operators such as `+`, `-`. `*`, `%`, and so on, may represent different operations according to the context they are used in. It doesn't make any sense to sum two lists, right? Therefore, the `+` sign is used to concatenate them. Hence, the `*` sign is used to concatenate the list to itself according to the right operand. Now, let's take a step further down the rabbit hole and see something a little more interesting. I want to show you how powerful the sort method can be and how easy it is in Python to achieve results that require a great deal of effort in other languages:

```
>>> from operator import itemgetter
>>> a = [(5, 3), (1, 3), (1, 2), (2, -1), (4, 9)]
>>> sorted(a)
[(1, 2), (1, 3), (2, -1), (4, 9), (5, 3)]
>>> sorted(a, key=itemgetter(0))
[(1, 3), (1, 2), (2, -1), (4, 9), (5, 3)]
>>> sorted(a, key=itemgetter(0, 1))
[(1, 2), (1, 3), (2, -1), (4, 9), (5, 3)]
>>> sorted(a, key=itemgetter(1))
[(2, -1), (1, 2), (5, 3), (1, 3), (4, 9)]
>>> sorted(a, key=itemgetter(1), reverse=True)
[(4, 9), (5, 3), (1, 3), (1, 2), (2, -1)]
```

The preceding code deserves a little explanation. First of all, `a` is a list of tuples. This means each element in `a` is a tuple (a *2-tuple*, to be picky). When we call `sorted(some_list)`, we get a sorted version of `some_list`. In this case, the sorting on a 2-tuple works by sorting them on the first item in the tuple, and on the second when the first one is the same. You can see this behavior in the result of `sorted(a)`, which yields `[(1, 2), (1, 3), ...]`. Python also gives us the ability to control on which element(s) of the tuple the sorting must be run against. Notice that when we instruct the `sorted` function to work on the first element of each tuple (by `key=itemgetter(0)`), the result is different: `[(1, 3), (1, 2), ...]`. The sorting is done only on the first element of each tuple (which is the one at position 0). If we want to replicate the default behavior of a simple `sorted(a)` call, we need to use `key=itemgetter(0, 1)`, which tells Python to sort first on the elements at position 0 within the tuples, and then on those at position 1. Compare the results and you'll see they match.

For completeness, I included an example of sorting only on the elements at position 1, and the same but in reverse order. If you have ever seen sorting in Java, I expect you to be on your knees crying with joy at this very moment.

The Python sorting algorithm is very powerful, and it was written by Tim Peters (we've already seen this name, can you recall when?). It is aptly named **Timsort**, and it is a blend between **merge** and **insertion sort** and has better time performances than most other algorithms used for mainstream programming languages. Timsort is a stable sorting algorithm, which means that when multiple records have the same key, their original order is preserved. We've seen this in the result of `sorted(a, key=itemgetter(0))` which has yielded `[(1, 3), (1, 2), ...]` in which the order of those two tuples has been preserved because they have the same value at position 0.

Byte arrays

To conclude our overview of mutable sequence types, let's spend a couple of minutes on the `bytearray` type. Basically, they represent the mutable version of `bytes` objects. They expose most of the usual methods of mutable sequences as well as most of the methods of the `bytes` type. Items are integers in the range [0, 256).

When it comes to intervals, I'm going to use the standard notation for open/closed ranges. A square bracket on one end means that the value is included, while a round brace means it's excluded. The granularity is usually inferred by the type of the edge elements so, for example, the interval [3, 7] means all integers between 3 and 7, inclusive. On the other hand, (3, 7) means all integers between 3 and 7 exclusive (hence 4, 5, and 6). Items in a `bytearray` type are integers between 0 and 256, 0 is included, 256 is not. One reason intervals are often expressed like this is to ease coding. If we break a range [a, b) into N consecutive ranges, we can easily represent the original one as a concatenation like this:

$$[a, k_1) + [k_1, k_2) + [k_2, k_3) + ... + [k_{N-1}, b)$$

The middle points (k_i) being excluded on one end, and included on the other end, allow for easy concatenation and splitting when intervals are handled in the code.

Let's see a quick example with the type `bytearray`:

```
>>> bytearray()  # empty bytearray object
bytearray(b'')
>>> bytearray(10)  # zero-filled instance with given length
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
>>> bytearray(range(5))  # bytearray from iterable of integers
bytearray(b'\x00\x01\x02\x03\x04')
```

```
>>> name = bytearray(b'Lina')  # A - bytearray from bytes
>>> name.replace(b'L', b'l')
bytearray(b'lina')
>>> name.endswith(b'na')
True
>>> name.upper()
bytearray(b'LINA')
>>> name.count(b'L')
1
```

As you can see in the preceding code, there are a few ways to create a `bytearray` object. They can be useful in many situations, for example, when receiving data through a **socket**, they eliminate the need to concatenate data while polling, hence they prove very handy. On the line `#A`, I created the `name` `bytearray` from the string `b'Lina'` to show you how the `bytearray` object exposes methods from both sequences and strings, which is extremely handy. If you think about it, they can be considered as mutable strings.

Set types

Python also provides two set types, `set` and `frozenset`. The `set` type is mutable, while `frozenset` is immutable. They are unordered collections of immutable objects.

Hashability is a characteristic that allows an object to be used as a set member as well as a key for a dictionary, as we'll see very soon.

An object is hashable if it has a hash value which never changes during its lifetime.

Objects that compare equally must have the same hash value. Sets are very commonly used to test for membership, so let's introduce the `in` operator in the following example:

```
>>> small_primes = set()  # empty set
>>> small_primes.add(2)  # adding one element at a time
>>> small_primes.add(3)
>>> small_primes.add(5)
>>> small_primes
{2, 3, 5}
```

```
>>> small_primes.add(1)  # Look what I've done, 1 is not a prime!
>>> small_primes
{1, 2, 3, 5}
>>> small_primes.remove(1)  # so let's remove it
>>> 3 in small_primes  # membership test
True
>>> 4 in small_primes
False
>>> 4 not in small_primes  # negated membership test
True
>>> small_primes.add(3)  # trying to add 3 again
>>> small_primes
{2, 3, 5}  # no change, duplication is not allowed
>>> bigger_primes = set([5, 7, 11, 13])  # faster creation
>>> small_primes | bigger_primes  # union operator `|`
{2, 3, 5, 7, 11, 13}
>>> small_primes & bigger_primes  # intersection operator `&`
{5}
>>> small_primes - bigger_primes  # difference operator `-`
{2, 3}
```

In the preceding code, you can see two different ways to create a set. One creates an empty set and then adds elements one at a time. The other creates the set using a list of numbers as argument to the constructor, which does all the work for us. Of course, you can create a set from a list or tuple (or any iterable) and then you can add and remove members from the set as you please.

Another way of creating a set is by simply using the curly braces notation, like this:

```
>>> small_primes = {2, 3, 5, 5, 3}
>>> small_primes
{2, 3, 5}
```

Notice I added some duplication to emphasize that the result set won't have any.

We'll see iterable objects and iteration in the next chapter. For now, just know that iterable objects are objects you can iterate on in a direction.

Let's see an example about the immutable counterpart of the set type: `frozenset`.

```
>>> small_primes = frozenset([2, 3, 5, 7])
>>> bigger_primes = frozenset([5, 7, 11])
>>> small_primes.add(11)  # we cannot add to a frozenset
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
>>> small_primes.remove(2)  # neither we can remove
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'remove'
>>> small_primes & bigger_primes  # intersect, union, etc. allowed
frozenset({5, 7})
```

As you can see, `frozenset` objects are quite limited in respect of their mutable counterpart. They still prove very effective for membership test, union, intersection and difference operations, and for performance reasons.

Mapping types – dictionaries

Of all the built-in Python data types, the dictionary is probably the most interesting one. It's the only standard mapping type, and it is the backbone of every Python object.

A dictionary maps keys to values. Keys need to be hashable objects, while values can be of any arbitrary type. Dictionaries are mutable objects.

There are quite a few different ways to create a dictionary, so let me give you a simple example of how to create a dictionary equal to `{'A': 1, 'Z': -1}` in five different ways:

```
>>> a = dict(A=1, Z=-1)
>>> b = {'A': 1, 'Z': -1}
>>> c = dict(zip(['A', 'Z'], [1, -1]))
>>> d = dict([('A', 1), ('Z', -1)])
>>> e = dict({'Z': -1, 'A': 1})
>>> a == b == c == d == e  # are they all the same?
True  # indeed!
```

Have you noticed those double equals? Assignment is done with one equal, while to check whether an object is the same as another one (or 5 in one go, in this case), we use double equals. There is also another way to compare objects, which involves the **is** operator, and checks whether the two objects are the same (if they have the same ID, not just the value), but unless you have a good reason to use it, you should use the double equal instead. In the preceding code, I also used one nice function: `zip`. It is named after the real-life zip, which glues together two things taking one element from each at a time. Let me show you an example:

```
>>> list(zip(['h', 'e', 'l', 'l', 'o'], [1, 2, 3, 4, 5]))
[('h', 1), ('e', 2), ('l', 3), ('l', 4), ('o', 5)]
>>> list(zip('hello', range(1, 6)))  # equivalent, more Pythonic
[('h', 1), ('e', 2), ('l', 3), ('l', 4), ('o', 5)]
```

In the preceding example, I have created the same list in two different ways, one more explicit, and the other a little bit more Pythonic. Forget for a moment that I had to wrap the `list` constructor around the `zip` call (the reason is because `zip` returns an iterator, not a `list`), and concentrate on the result. See how `zip` has coupled the first elements of its two arguments together, then the second ones, then the third ones, and so on and so forth? Take a look at your pants (or at your purse if you're a lady) and you'll see the same behavior in your actual zip. But let's go back to dictionaries and see how many wonderful methods they expose for allowing us to manipulate them as we want. Let's start with the basic operations:

```
>>> d = {}
>>> d['a'] = 1  # let's set a couple of (key, value) pairs
>>> d['b'] = 2
>>> len(d)  # how many pairs?
2
>>> d['a']  # what is the value of 'a'?
1
>>> d  # how does `d` look now?
{'a': 1, 'b': 2}
>>> del d['a']  # let's remove `a`
>>> d
{'b': 2}
>>> d['c'] = 3  # let's add 'c': 3
>>> 'c' in d  # membership is checked against the keys
True
>>> 3 in d  # not the values
```

```
False
>>> 'e' in d
False
>>> d.clear()  # let's clean everything from this dictionary
>>> d
{}
```

Notice how accessing keys of a dictionary, regardless of the type of operation we're performing, is done through square brackets. Do you remember strings, list, and tuples? We were accessing elements at some position through square brackets as well. Yet another example of Python's consistency.

Let's see now three special objects called dictionary views: `keys`, `values`, and `items`. These objects provide a dynamic view of the dictionary entries and they change when the dictionary changes. `keys()` returns all the keys in the dictionary, `values()` returns all the values in the dictionary, and `items()` returns all the *(key, value)* pairs in the dictionary.

It's very important to know that, even if a dictionary is not intrinsically ordered, according to the Python documentation: "*Keys and values are iterated over in an arbitrary order which is non-random, varies across Python implementations, and depends on the dictionary's history of insertions and deletions. If keys, values and items views are iterated over with no intervening modifications to the dictionary, the order of items will directly correspond.*"

Enough with this chatter, let's put all this down into code:

```
>>> d = dict(zip('hello', range(5)))
>>> d
{'e': 1, 'h': 0, 'o': 4, 'l': 3}
>>> d.keys()
dict_keys(['e', 'h', 'o', 'l'])
>>> d.values()
dict_values([1, 0, 4, 3])
>>> d.items()
dict_items([('e', 1), ('h', 0), ('o', 4), ('l', 3)])
>>> 3 in d.values()
True
>>> ('o', 4) in d.items()
True
```

A few things to notice in the preceding code. First, notice how we're creating a dictionary by iterating over the zipped version of the string `'hello'` and the list `[0, 1, 2, 3, 4]`. The string `'hello'` has two `'l'` characters inside, and they are paired up with the values 2 and 3 by the zip function. Notice how in the dictionary, the second occurrence of the `'l'` key (the one with value 3), overwrites the first one (the one with value 2). Another thing to notice is that when asking for any view, the original order is lost, but is consistent within the views, as expected. Notice also that you may have different results when you try this code on your machine. Python doesn't guarantee that, it only guarantees the consistency of the order in which the views are presented.

We'll see how these views are fundamental tools when we talk about iterating over collections. Let's take a look now at some other methods exposed by Python's dictionaries, there's plenty of them and they are very useful:

```
>>> d
{'e': 1, 'h': 0, 'o': 4, 'l': 3}
>>> d.popitem()  # removes a random item
('e', 1)
>>> d
{'h': 0, 'o': 4, 'l': 3}
>>> d.pop('l')  # remove item with key `l`
3
>>> d.pop('not-a-key')  # remove a key not in dictionary: KeyError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'not-a-key'
>>> d.pop('not-a-key', 'default-value')  # with a default value?
'default-value'  # we get the default value
>>> d.update({'another': 'value'})  # we can update dict this way
>>> d.update(a=13)  # or this way (like a function call)
>>> d
{'a': 13, 'another': 'value', 'h': 0, 'o': 4}
>>> d.get('a')  # same as d['a'] but if key is missing no KeyError
13
>>> d.get('a', 177)  # default value used if key is missing
13
>>> d.get('b', 177)  # like in this case
177
>>> d.get('b')  # key is not there, so None is returned
```

All these methods are quite simple to understand, but it's worth talking about that `None`, for a moment. Every function in Python returns `None`, unless the `return` statement is explicitly used, but we'll see this when we explore functions. `None` is frequently used to represent the absence of a value, as when default arguments are not passed to a function. Some inexperienced coders sometimes write code that returns either `False` or `None`. Both `False` and `None` evaluate to `False` so it may seem there is not much difference between them. But actually, I would argue there is quite an important difference: `False` means that we have information, and the information we have is `False`. `None` means *no information*. And no information is very different from an information, which is `False`. In layman's terms, if you ask your mechanic "is my car ready?" there is a big difference between the answer "No, it's not" (*False*) and "I have no idea" (*None*).

One last method I really like of dictionaries is `setdefault`. It behaves like `get`, but also sets the key with the given value if it is not there. Let's see and example:

```
>>> d = {}
>>> d.setdefault('a', 1)  # 'a' is missing, we get default value
1
>>> d
{'a': 1}  # also, the key/value pair ('a', 1) has now been added
>>> d.setdefault('a', 5)  # let's try to override the value
1
>>> d
{'a': 1}  # didn't work, as expected
```

So, we're now at the end of this tour. Test your knowledge about dictionaries trying to foresee how `d` looks like after this line.

```
>>> d = {}
>>> d.setdefault('a', {}).setdefault('b', []).append(1)
```

It's not that complicated, but don't worry if you don't get it immediately. I just wanted to spur you to experiment with dictionaries.

This concludes our tour of built-in data types. Before I make some considerations about what we've seen in this chapter, I want to briefly take a peek at the `collections` module.

The collections module

When Python general purpose built-in containers (`tuple`, `list`, `set`, and `dict`) aren't enough, we can find specialized container data types in the `collections` module. They are:

Data type	Description
`namedtuple()`	A factory function for creating tuple subclasses with named fields
`deque`	A `list`-like container with fast appends and pops on either end
`ChainMap`	A `dict`-like class for creating a single view of multiple mappings
`Counter`	A `dict` subclass for counting hashable objects
`OrderedDict`	A `dict` subclass that remembers the order entries were added
`defaultdict`	A `dict` subclass that calls a factory function to supply missing values
`UserDict`	A wrapper around dictionary objects for easier dict subclassing
`UserList`	A wrapper around list objects for easier list subclassing
`UserString`	A wrapper around string objects for easier string subclassing

We don't have the room to cover all of them, but you can find plenty of examples in the official documentation, so here I'll just give a small example to show you `namedtuple`, `defaultdict`, and `ChainMap`.

Named tuples

A `namedtuple` is a `tuple`-like object that has fields accessible by attribute lookup as well as being indexable and iterable (it's actually a subclass of `tuple`). This is sort of a compromise between a full-fledged object and a tuple, and it can be useful in those cases where you don't need the full power of a custom object, but you want your code to be more readable by avoiding weird indexing. Another use case is when there is a chance that items in the tuple need to change their position after refactoring, forcing the coder to refactor also all the logic involved, which can be very tricky. As usual, an example is better than a thousand words (or was it a picture?). Say we are handling data about the left and right eye of a patient. We save one value for the left eye (position 0) and one for the right eye (position 1) in a regular tuple. Here's how that might be:

```
>>> vision = (9.5, 8.8)
>>> vision
(9.5, 8.8)
>>> vision[0]  # left eye (implicit positional reference)
```

```
9.5
>>> vision[1]  # right eye (implicit positional reference)
8.8
```

Now let's pretend we handle `vision` object all the time, and at some point the designer decides to enhance them by adding information for the combined vision, so that a `vision` object stores data in this format: *(left eye, combined, right eye)*.

Do you see the trouble we're in now? We may have a lot of code that depends on `vision[0]` being the left eye information (which still is) and `vision[1]` being the right eye information (which is no longer the case). We have to refactor our code wherever we handle these objects, changing `vision[1]` to `vision[2]`, and it can be painful. We could have probably approached this a bit better from the beginning, by using a `namedtuple`. Let me show you what I mean:

```
>>> from collections import namedtuple
>>> Vision = namedtuple('Vision', ['left', 'right'])
>>> vision = Vision(9.5, 8.8)
>>> vision[0]
9.5
>>> vision.left  # same as vision[0], but explicit
9.5
>>> vision.right  # same as vision[1], but explicit
8.8
```

If within our code we refer to left and right eye using `vision.left` and `vision.right`, all we need to do to fix the new design issue is to change our factory and the way we create instances. The rest of the code won't need to change.

```
>>> Vision = namedtuple('Vision', ['left', 'combined', 'right'])
>>> vision = Vision(9.5, 9.2, 8.8)
>>> vision.left  # still perfect
9.5
>>> vision.right  # still perfect (though now is vision[2])
8.8
>>> vision.combined  # the new vision[1]
9.2
```

You can see how convenient it is to refer to those values by name rather than by position. After all, a wise man once wrote "*Explicit is better than implicit*" (can you recall where? Think *zen* if you don't...). This example may be a little extreme, of course it's not likely that our code designer will go for a change like this, but you'd be amazed to see how frequently issues similar to this one happen in a professional environment, and how painful it is to refactor them.

Defaultdict

The `defaultdict` data type is one of my favorites. It allows you to avoid checking if a key is in a dictionary by simply inserting it for you on your first access attempt, with a default value whose type you pass on creation. In some cases, this tool can be very handy and shorten your code a little. Let's see a quick example: say we are updating the value of `age`, by adding one year. If `age` is not there, we assume it was `0` and we update it to `1`.

```
>>> d = {}
>>> d['age'] = d.get('age', 0) + 1  # age not there, we get 0 + 1
>>> d
{'age': 1}
>>> d = {'age': 39}
>>> d['age'] = d.get('age', 0) + 1 # age is there, we get 40
>>> d
{'age': 40}
```

Now let's see how it would work with a `defaultdict` data type. The second line is actually the short version of a 4-lines long `if` clause that we would have to write if dictionaries didn't have the `get` method. We'll see all about `if` clauses in *Chapter 3, Iterating and Making Decisions*.

```
>>> from collections import defaultdict
>>> dd = defaultdict(int)  # int is the default type (0 the value)
>>> dd['age'] += 1  # short for dd['age'] = dd['age'] + 1
>>> dd
defaultdict(<class 'int'>, {'age': 1})  # 1, as expected
>>> dd['age'] = 39
>>> dd['age'] += 1
>>> dd
defaultdict(<class 'int'>, {'age': 40})  # 40, as expected
```

Notice how we just need to instruct the `defaultdict` factory that we want an `int` number to be used in case the key is missing (we'll get `0`, which is the default for the `int` type). Also, notice that even though in this example there is no gain on the number of lines, there is definitely a gain in readability, which is very important. You can also use a different technique to instantiate a `defaultdict` data type, which involves creating a factory object. For digging deeper, please refer to the official documentation.

ChainMap

The `ChainMap` is an extremely nice data type which was introduced in Python 3.3. It behaves like a normal dictionary but according to the Python documentation: *is provided for quickly linking a number of mappings so they can be treated as a single unit.* This is usually much faster than creating one dictionary and running multiple update calls on it. `ChainMap` can be used to simulate nested scopes and is useful in templating. The underlying mappings are stored in a list. That list is public and can be accessed or updated using the maps attribute. Lookups search the underlying mappings successively until a key is found. In contrast, writes, updates, and deletions only operate on the first mapping.

A very common use case is providing defaults, so let's see an example:

```
>>> from collections import ChainMap
>>> default_connection = {'host': 'localhost', 'port': 4567}
>>> connection = {'port': 5678}
>>> conn = ChainMap(connection, default_connection) # map creation
>>> conn['port']  # port is found in the first dictionary
5678
>>> conn['host']  # host is fetched from the second dictionary
'localhost'
>>> conn.maps  # we can see the mapping objects
[{'port': 5678}, {'host': 'localhost', 'port': 4567}]
>>> conn['host'] = 'packtpub.com'  # let's add host
>>> conn.maps
[{'host': 'packtpub.com', 'port': 5678},
 {'host': 'localhost', 'port': 4567}]
>>> del conn['port']  # let's remove the port information
>>> conn.maps
[{'host': 'packtpub.com'},
 {'host': 'localhost', 'port': 4567}]
>>> conn['port']  # now port is fetched from the second dictionary
```

```
4567
>>> dict(conn)  # easy to merge and convert to regular dictionary
{'host': 'packtpub.com', 'port': 4567}
```

I just love how Python makes your life easy. You work on a `ChainMap` object, configure the first mapping as you want, and when you need a complete dictionary with all the defaults as well as the customized items, you just feed the `ChainMap` object to a `dict` constructor. If you have never coded in other languages, such as Java or C++, you probably won't be able to fully appreciate how precious this is, how Python makes your life so much easier. I do, I feel claustrophobic every time I have to code in some other language.

Final considerations

That's it. Now you have seen a very good portion of the data structures that you will use in Python. I encourage you to take a dive into the Python documentation and experiment further with each and every data type we've seen in this chapter. It's worth it, believe me. Everything you'll write will be about handling data, so make sure your knowledge about it is rock solid.

Before we leap into the next chapter, I'd like to make some final considerations about different aspects that to my mind are important and not to be neglected.

Small values caching

When we discussed objects at the beginning of this chapter, we saw that when we assigned a name to an object, Python creates the object, sets its value, and then points the name to it. We can assign different names to the same value and we expect different objects to be created, like this:

```
>>> a = 1000000
>>> b = 1000000
>>> id(a) == id(b)
False
```

In the preceding example, `a` and `b` are assigned to two `int` objects, which have the same value but they are not the same object, as you can see, their `id` is not the same. So let's do it again:

```
>>> a = 5
>>> b = 5
>>> id(a) == id(b)
True
```

Oh oh! Is Python broken? Why are the two objects the same now? We didn't do `a = b = 5`, we set them up separately. Well, the answer is performances. Python caches short strings and small numbers, to avoid having many copies of them clogging up the system memory. Everything is handled properly under the hood so you don't need to worry a bit, but make sure that you remember this behavior should your code ever need to fiddle with IDs.

How to choose data structures

As we've seen, Python provides you with several built-in data types and sometimes, if you're not that experienced, choosing the one that serves you best can be tricky, especially when it comes to collections. For example, say you have many dictionaries to store, each of which represents a customer. Within each customer dictionary there's an `'id': 'code'` unique identification code. In what kind of collection would you place them? Well, unless I know more about these customers, it's very hard to answer. What kind of access will I need? What sort of operations will I have to perform on each of them, and how many times? Will the collection change over time? Will I need to modify the customer dictionaries in any way? What is going to be the most frequent operation I will have to perform on the collection?

If you can answer the preceding questions, then you will know what to choose. If the collection never shrinks or grows (in other words, it won't need to add/delete any customer object after creation) or shuffles, then tuples are a possible choice. Otherwise lists are a good candidate. Every customer dictionary has a unique identifier though, so even a dictionary could work. Let me draft these options for you:

```
# example customer objects
customer1 = {'id': 'abc123', 'full_name': 'Master Yoda'}
customer2 = {'id': 'def456', 'full_name': 'Obi-Wan Kenobi'}
customer3 = {'id': 'ghi789', 'full_name': 'Anakin Skywalker'}
# collect them in a tuple
customers = (customer1, customer2, customer3)
# or collect them in a list
customers = [customer1, customer2, customer3]
# or maybe within a dictionary, they have a unique id after all
customers = {
    'abc123': customer1,
    'def456': customer2,
    'ghi789': customer3,
}
```

Some customers we have there, right? I probably wouldn't go with the tuple option, unless I wanted to highlight that the collection is not going to change. I'd say usually a list is better, it allows for more flexibility.

Another factor to keep in mind is that tuples and lists are ordered collections, while if you use a dictionary or a set you lose the ordering, so you need to know if ordering is important in your application.

What about performances? For example in a list, operations such as insertion and membership can take *O(n)*, while they are *O(1)* for a dictionary. It's not always possible to use dictionaries though, if we don't have the guarantee that we can uniquely identify each item of the collection by means of one of its properties, and that the property in question is hashable (so it can be a key in `dict`).

If you're wondering what *O(n)* and *O(1)* mean, please Google "*big O notation*" and get a gist of it from anywhere. In this context, let's just say that if performing an operation *Op* on a data structure takes *O(f(n))*, it would mean that *Op* takes at most a time $t \leq c \cdot f(n)$ to complete, where *c* is some positive constant, *n* is the size of the input, and *f* is some function. So, think of *O(...)* as an upper bound for the running time of an operation (it can be used also to size other measurable quantities, of course).

Another way of understanding if you have chosen the right data structure is by looking at the code you have to write in order to manipulate it. If everything comes easily and flows naturally, then you probably have chosen correctly, but if you find yourself thinking your code is getting unnecessarily complicated, then you probably should try and decide whether you need to reconsider your choices. It's quite hard to give advice without a practical case though, so when you choose a data structure for your data, try to keep ease of use and performance in mind and give precedence to what matters most in the context you are.

About indexing and slicing

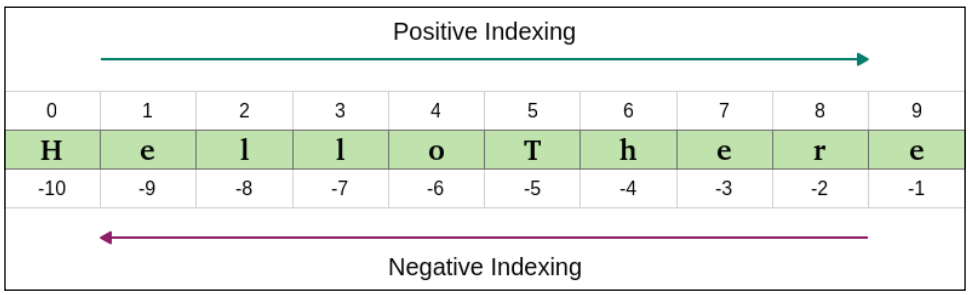
At the beginning of this chapter, we saw slicing applied on strings. Slicing in general applies to a sequence, so tuples, lists, strings, etc. With lists, slicing can also be used for assignment. I've almost never seen this used in professional code, but still, you know you can. Could you slice dictionaries or sets? I hear you scream "*Of course not! They are not ordered!*". Excellent, I see we're on the same page here, so let's talk about indexing.

There is one characteristic about Python indexing I haven't mentioned before. I'll show you by example. How do you address the last element of a collection? Let's see:

```
>>> a = list(range(10))  # `a` has 10 elements. Last one is 9.
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> len(a)  # its length is 10 elements
10
>>> a[len(a) - 1]  # position of last one is len(a) - 1
9
>>> a[-1]  # but we don't need len(a)! Python rocks!
9
>>> a[-2]  # equivalent to len(a) - 2
8
>>> a[-3]  # equivalent to len(a) - 3
7
```

If the list `a` has 10 elements, because of the *0-index* positioning system of Python, the first one is at position 0 and the last one is at position 9. In the preceding example, the elements are conveniently placed in a position equal to their value: 0 is at position 0, 1 at position 1, and so on.

So, in order to fetch the last element, we need to know the length of the whole list (or tuple, or string, and so on) and then subtract 1. Hence: `len(a) - 1`. This is so common an operation that Python provides you with a way to retrieve elements using **negative indexing**. This proves very useful when you do some serious data manipulation. Here's a nice diagram about how indexing works on the string `"HelloThere"`:



Trying to address indexes greater than 9 or smaller than -10 will raise an `IndexError`, as expected.

About the names

You may have noticed that, in order to keep the example as short as possible, I have called many objects using simple letters, like `a`, `b`, `c`, `d`, and so on. This is perfectly ok when you debug on the console or when you show that `a + b == 7`, but it's bad practice when it comes to professional coding (or any type of coding, for all that matter). I hope you will indulge me if I sometimes do it, the reason is to present the code in a more compact way.

In a real environment though, when you choose names for your data, you should choose them carefully and they should reflect what the data is about. So, if you have a collection of `Customer` objects, `customers` is a perfectly good name for it. Would `customers_list`, `customers_tuple`, or `customers_collection` work as well? Think about it for a second. Is it good to tie the name of the collection to the data type? I don't think so, at least in most cases. So I'd say if you have an excellent reason to do so go ahead, otherwise don't. The reason is, once that `customers_tuple` starts being used in different places of your code, and you realize you actually want to use a list instead of a tuple, you're up for some fun refactoring (also known as **wasted time**). Names for data should be nouns, and names for functions should be verbs. Names should be as expressive as possible. Python is actually a very good example when it comes to names. Most of the time you can just guess what a function is called if you know what it does. Crazy, huh?

Chapter 2, *Meaningful Names* of *Clean Code*, *Robert C. Martin*, *Prentice Hall* is entirely dedicated to names. It's an amazing book that helped me improve my coding style in many different ways, a must read if you want to take your coding to the next level.

Summary

In this chapter, we've explored the built-in data types of Python. We've seen how many they are and how much can be achieved by just using them in different combinations.

We've seen number types, sequences, sets, mappings, collections, we've seen that everything is an object, we've learned the difference between mutable and immutable, and we've also learned about slicing and indexing (and, proudly, negative indexing as well).

We've presented simple examples, but there's much more that you can learn about this subject, so stick your nose into the official documentation and explore.

Most of all, I encourage you to try out all the exercises by yourself, get your fingers using that code, build some muscle memory, and experiment, experiment, experiment. Learn what happens when you divide by zero, when you combine different number types into a single expression, when you manage strings. Play with all data types. Exercise them, break them, discover all their methods, enjoy them and learn them well, damn well.

If your foundation is not rock solid, how good can your code be? And data is the foundation for everything. Data shapes what dances around it.

The more you progress with the book, the more it's likely that you will find some discrepancies or maybe a small typo here and there in my code (or yours). You will get an error message, something will break. That's wonderful! When you code, things break all the time, you debug and fix all the time, so consider errors as useful exercises to learn something new about the language you're using, and not as failures or problems. Errors will keep coming up until your very last line of code, that's for sure, so you may as well start making your peace with them now.

The next chapter is about iterating and making decisions. We'll see how to actually put those collections in use, and take decisions based on the data we're presented with. We'll start to go a little faster now that your knowledge is building up, so make sure you're comfortable with the contents of this chapter before you move to the next one. Once more, have fun, explore, break things. It's a very good way to learn.

3
Iterating and Making Decisions

> "*Insanity: doing the same thing over and over again and expecting different results.*"
>
> *- Albert Einstein*

In the previous chapter, we've seen Python built-in data types. Now that you're familiar with data in its many forms and shapes, it's time to start looking at how a program can use it.

According to Wikipedia:

> *In computer science, control flow (or alternatively, flow of control) refers to the specification of the order in which the individual statements, instructions or function calls of an imperative program are executed or evaluated.*

In order to control the flow of a program, we have two main weapons: **conditional programming** (also known as **branching**) and **looping**. We can use them in many different combinations and variations, but in this chapter, instead of going through all possible various forms of those two constructs in a "documentation" fashion, I'd rather give you the basics and then I'll write a couple of small scripts with you. In the first one, we'll see how to create a rudimentary prime number generator, while in the second one, we'll see how to apply discounts to customers based on coupons. This way you should get a better feeling about how conditional programming and looping can be used.

Conditional programming

Conditional programming, or branching, is something you do every day, every moment. It's about evaluating conditions: *if the light is green, then I can cross*, *if it's raining, then I'm taking the umbrella*, and *if I'm late for work, then I'll call my manager*.

The main tool is the `if` statement, which comes in different forms and colors, but basically what it does is evaluate an expression and, based on the result, choose which part of the code to execute. As usual, let's see an example:

`conditional.1.py`

```
late = True
if late:
    print('I need to call my manager!')
```

This is possibly the simplest example: when fed to the `if` statement, `late` acts as a conditional expression, which is evaluated in a Boolean context (exactly like if we were calling `bool(late)`). If the result of the evaluation is `True`, then we enter the body of code immediately after the `if` statement. Notice that the `print` instruction is indented: this means it belongs to a scope defined by the `if` clause. Execution of this code yields:

```
$ python conditional.1.py
I need to call my manager!
```

Since `late` is `True`, the `print` statement was executed. Let's expand on this example:

`conditional.2.py`

```
late = False
if late:
    print('I need to call my manager!')  #1
else:
    print('no need to call my manager...')  #2
```

This time I set `late = False`, so when I execute the code, the result is different:

```
$ python conditional.2.py
no need to call my manager...
```

Depending on the result of evaluating the `late` expression, we can either enter block `#1` or block `#2`, *but not both*. Block `#1` is executed when `late` evaluates to `True`, while block `#2` is executed when `late` evaluates to `False`. Try assigning `False/True` values to the `late` name, and see how the output for this code changes accordingly.

The preceding example also introduces the `else` clause, which becomes very handy when we want to provide an alternative set of instructions to be executed when an expression evaluates to `False` within an `if` clause. The else clause is optional, as it's evident by comparing the preceding two examples.

A specialized else: elif

Sometimes all you need is to do something if a condition is met (simple `if` clause). Other times you need to provide an alternative, in case the condition is `False` (`if/else` clause), but there are situations where you may have more than two paths to choose from, so, since calling the manager (or not calling them) is kind of a binary type of example (either you call or you don't), let's change the type of example and keep expanding. This time we decide tax percentages. If my income is less then 10k, I won't pay any taxes. If it is between 10k and 30k, I'll pay 20% taxes. If it is between 30k and 100k, I'll pay 35% taxes, and over 100k, I'll (gladly) pay 45% taxes. Let's put this all down into beautiful Python code:

`taxes.py`

```
income = 15000
if income < 10000:
    tax_coefficient = 0.0  #1
elif income < 30000:
    tax_coefficient = 0.2  #2
elif income < 100000:
    tax_coefficient = 0.35  #3
else:
    tax_coefficient = 0.45  #4

print('I will pay:', income * tax_coefficient, 'in taxes')
```

Executing the preceding code yields:

```
$ python taxes.py
I will pay: 3000.0 in taxes
```

Let's go through the example line by line: we start by setting up the income value. In the example, my income is 15k. We enter the `if` clause. Notice that this time we also introduced the `elif` clause, which is a contraction for `else-if`, and it's different from a bare `else` clause in that it also has its own condition. So, the `if` expression `income < 10000`, evaluates to `False`, therefore block `#1` is not executed. The control passes to the next condition evaluator: `elif income < 30000`. This one evaluates to `True`, therefore block `#2` is executed, and because of this, Python then resumes execution after the whole `if/elif/elif/else` clause (which we can just call `if` clause from now on). There is only one instruction after the `if` clause, the `print` call, which tells us I will pay 3k in taxes this year (*15k * 20%*). Notice that the order is mandatory: `if` comes first, then (optionally) as many `elif` as you need, and then (optionally) an `else` clause.

Interesting, right? No matter how many lines of code you may have within each block, when one of the conditions evaluates to `True`, the associated block is executed and then execution resumes after the whole clause. If none of the conditions evaluates to `True` (for example, `income = 200000`), then the body of the `else` clause would be executed (block `#4`). This example expands our understanding of the behavior of the `else` clause. Its block of code is executed when none of the preceding `if/elif/.../elif` expressions has evaluated to `True`.

Try to modify the value of `income` until you can comfortably execute all blocks at your will (one per execution, of course). And then try the **boundaries**. This is crucial, whenever you have conditions expressed as **equalities** or **inequalities** (`==`, `!=`, `<`, `>`, `<=`, `>=`), those numbers represent boundaries. It is essential to test boundaries thoroughly. Should I allow you to drive at 18 or 17? Am I checking your age with `age < 18`, or `age <= 18`? You can't imagine how many times I had to fix subtle bugs that stemmed from using the wrong operator, so go ahead and experiment with the preceding code. Change some `<` to `<=` and set income to be one of the boundary values (10k, 30k, 100k) as well as any value in between. See how the result changes, get a good understanding of it before proceeding.

Before we move to the next topic, let's see another example that shows us how to nest `if` clauses. Say your program encounters an error. If the alert system is the console, we print the error. If the alert system is an e-mail, we send it according to the severity of the error. If the alert system is anything other than console or e-mail, we don't know what to do, therefore we do nothing. Let's put this into code:

`errorsalert.py`

```
alert_system = 'console'  # other value can be 'email'
error_severity = 'critical'  # other values: 'medium' or 'low'
error_message = 'OMG! Something terrible happened!'
```

```
if alert_system == 'console':
    print(error_message)  #1
elif alert_system == 'email':
    if error_severity == 'critical':
        send_email('admin@example.com', error_message)  #2
    elif error_severity == 'medium':
        send_email('support.1@example.com', error_message)  #3
    else:
        send_email('support.2@example.com', error_message)  #4
```

The preceding example is quite interesting, in its silliness. It shows us two nested `if` clauses (**outer** and **inner**). It also shows us the outer `if` clause doesn't have any `else`, while the inner one does. Notice how indentation is what allows us to nest one clause within another one.

If `alert_system == 'console'`, body `#1` is executed, and nothing else happens. On the other hand, if `alert_system == 'email'`, then we enter into another `if` clause, which we called inner. In the inner `if` clause, according to `error_severity`, we send an e-mail to either an admin, first-level support, or second-level support (blocks `#2`, `#3`, and `#4`). The `send_email` function is not defined in this example, therefore trying to run it would give you an error. In the source code of the book, which you can download from the website, I included a trick to redirect that call to a regular `print` function, just so you can experiment on the console without actually sending an e-mail. Try changing the values and see how it all works.

The ternary operator

One last thing I would like to show you before moving on to the next subject, is the **ternary operator** or, in layman's terms, the short version of an `if/else` clause. When the value of a name is to be assigned according to some condition, sometimes it's easier and more readable to use the ternary operator instead of a proper `if` clause. In the following example, the two code blocks do exactly the same thing:

`ternary.py`

```
order_total = 247  # GBP

# classic if/else form
if order_total > 100:
    discount = 25  # GBP
else:
    discount = 0  # GBP
print(order_total, discount)
```

```
# ternary operator
discount = 25 if order_total > 100 else 0
print(order_total, discount)
```

For simple cases like this, I find it very nice to be able to express that logic in one line instead of four. Remember, as a coder, you spend much more time reading code then writing it, so Python conciseness is invaluable.

Are you clear on how the ternary operator works? Basically is `name = something if condition else something-else`. So `name` is assigned `something` if `condition` evaluates to `True`, and `something-else` if `condition` evaluates to `False`.

Now that you know everything about controlling the path of the code, let's move on to the next subject: looping.

Looping

If you have any experience with looping in other programming languages, you will find Python's way of looping a bit different. First of all, what is looping? **Looping** means being able to repeat the execution of a code block more than once, according to the loop parameters we're given. There are different looping constructs, which serve different purposes, and Python has distilled all of them down to just two, which you can use to achieve everything you need. These are the **for** and **while** statements.

While it's definitely possible to do everything you need using either of them, they serve different purposes and therefore they're usually used in different contexts. We'll explore this difference thoroughly through this chapter.

The for loop

The `for` loop is used when looping over a sequence, like a list, tuple, or a collection of objects. Let's start with a simple example that is more like C++ style, and then let's gradually see how to achieve the same results in Python (you'll love Python's syntax).

`simple.for.py`

```
for number in [0, 1, 2, 3, 4]:
    print(number)
```

This simple snippet of code, when executed, prints all numbers from 0 to 4. The `for` loop is fed the list `[0, 1, 2, 3, 4]` and at each iteration, `number` is given a value from the sequence (which is iterated sequentially, in order), then the body of the loop is executed (the print line). `number` changes at every iteration, according to which value is coming next from the sequence. When the sequence is exhausted, the `for` loop terminates, and the execution of the code resumes normally with the code after the loop.

Iterating over a range

Sometimes we need to iterate over a range of numbers, and it would be quite unpleasant to have to do so by hardcoding the list somewhere. In such cases, the `range` function comes to the rescue. Let's see the equivalent of the previous snippet of code:

`simple.for.py`

```
for number in range(5):
    print(number)
```

The range function is used extensively in Python programs when it comes to creating sequences: you can call it by passing one value, which acts as `stop` (counting from 0), or you can pass two values (`start` and `stop`), or even three (`start`, `stop`, and `step`). Check out the following example:

```
>>> list(range(10))  # one value: from 0 to value (excluded)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(3, 8))  # two values: from start to stop (excluded)
[3, 4, 5, 6, 7]
>>> list(range(-10, 10, 4))  # three values: step is added
[-10, -6, -2, 2, 6]
```

For the moment, ignore that we need to wrap `range(...)` within a `list`. The `range` object is a little bit special, but in this case we're just interested in understanding what are the values it will return to us. You see that the deal is the same with slicing: `start` is included, `stop` excluded, and optionally you can add a `step` parameter, which by default is 1.

Try modifying the parameters of the `range()` call in our `simple.for.py` code and see what it prints, get comfortable with it.

Iterating over a sequence

Now we have all the tools to iterate over a sequence, so let's build on that example:

`simple.for.2.py`

```
surnames = ['Rivest', 'Shamir', 'Adleman']
for position in range(len(surnames)):
    print(position, surnames[position])
```

The preceding code adds a little bit of complexity to the game. Execution will show this result:

```
$ python simple.for.2.py
0 Rivest
1 Shamir
2 Adleman
```

Let's use the **inside-out** technique to break it down, ok? We start from the innermost part of what we're trying to understand, and we expand outwards. So, `len(surnames)` is the length of the `surnames` list: `3`. Therefore, `range(len(surnames))` is actually transformed into `range(3)`. This gives us the range [0, 3), which is basically a sequence `(0, 1, 2)`. This means that the `for` loop will run three iterations. In the first one, `position` will take value `0`, while in the second one, it will take value `1`, and finally value `2` in the third and last iteration. What is `(0, 1, 2)`, if not the possible indexing positions for the `surnames` list? At position `0` we find `'Rivest'`, at position `1`, `'Shamir'`, and at position `2`, `'Adleman'`. If you are curious about what these three men created together, change `print(position, surnames[position])` to `print(surnames[position][0], end='')` add a final `print()` outside of the loop, and run the code again.

Now, this style of looping is actually much closer to languages like Java or C++. In Python it's quite rare to see code like this. You can just iterate over any sequence or collection, so there is no need to get the list of positions and retrieve elements out of a sequence at each iteration. It's expensive, needlessly expensive. Let's change the example into a more Pythonic form:

`simple.for.3.py`

```
surnames = ['Rivest', 'Shamir', 'Adleman']
for surname in surnames:
    print(surname)
```

Now that's something! It's practically English. The `for` loop can iterate over the `surnames` list, and it gives back each element in order at each interaction. Running this code will print the three surnames, one at a time. It's much easier to read, right?

What if you wanted to print the position as well though? Or what if you actually needed it for any reason? Should you go back to the `range(len(...))` form? No. You can use the `enumerate` built-in function, like this:

`simple.for.4.py`

```
surnames = ['Rivest', 'Shamir', 'Adleman']
for position, surname in enumerate(surnames):
    print(position, surname)
```

This code is very interesting as well. Notice that enumerate gives back a 2-tuple `(position, surname)` at each iteration, but still, it's much more readable (and more efficient) than the `range(len(...))` example. You can call `enumerate` with a `start` parameter, like `enumerate(iterable, start)`, and it will start from `start`, rather than `0`. Just another little thing that shows you how much thought has been given in designing Python so that it makes your life easy.

Using a `for` loop it is possible to iterate over lists, tuples, and in general anything that in Python is called iterable. This is a very important concept, so let's talk about it a bit more.

Iterators and iterables

According to the Python documentation, an iterable is:

> *"An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as* `list`*,* `str`*, and tuple) and some non-sequence types like* `dict`*,* `file` *objects, and objects of any classes you define with an* `__iter__()` *or* `__getitem__()` *method. Iterables can be used in a* `for` *loop and in many other places where a sequence is needed (*`zip()`*,* `map()`*, ...). When an iterable object is passed as an argument to the built-in function* `iter()`*, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call* `iter()` *or deal with iterator objects yourself. The* `for` *statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop."*

Simply put, what happens when you write `for k in sequence: ... body ...`, is that the `for` loop asks `sequence` for the next element, it gets something back, it calls that something `k`, and then executes its body. Then, once again, the `for` loop asks `sequence` again for the next element, it calls it `k` again, and executes the body again, and so on and so forth, until the sequence is exhausted. Empty sequences will result in zero executions of the body.

Some data structures, when iterated over, produce their elements in order, like lists, tuples, and strings, while some others don't, like sets and dictionaries.

Python gives us the ability to iterate over iterables, using a type of object called **iterator**. According to the official documentation, an iterator is:

> "*An object representing a stream of data. Repeated calls to the iterator's* `__next__()` *method (or passing it to the built-in function* `next()`*) return successive items in the stream. When no more data are available a* `StopIteration` *exception is raised instead. At this point, the iterator object is exhausted and any further calls to its* `__next__()` *method just raise* `StopIteration` *again. Iterators are required to have an* `__iter__()` *method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a* `list`*) produces a fresh new iterator each time you pass it to the* `iter()` *function or use it in a* `for` *loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.*"

Don't worry if you don't fully understand all the preceding legalese, you will in due time. I put it here as a handy reference for the future.

In practice, the whole iterable/iterator mechanism is somewhat hidden behind the code. Unless you need to code your own iterable or iterator for some reason, you won't have to worry about this too much. But it's very important to understand how Python handles this key aspect of control flow because it will shape the way you will write your code.

Iterating over multiple sequences

Let's see another example of how to iterate over two sequences of the same length, in order to work on their respective elements in pairs. Say we have a list of people and a list of numbers representing the age of the people in the first list. We want to print a pair person/age on one line for all of them. Let's start with an example and let's refine it gradually.

multiple.sequences.py

```
people = ['Jonas', 'Julio', 'Mike', 'Mez']
ages = [25, 30, 31, 39]
for position in range(len(people)):
    person = people[position]
    age = ages[position]
    print(person, age)
```

By now, this code should be pretty straightforward for you to understand. We need to iterate over the list of positions (0, 1, 2, 3) because we want to retrieve elements from two different lists. Executing it we get the following:

```
$ python multiple.sequences.py
Jonas 25
Julio 30
Mike 31
Mez 39
```

This code is both inefficient and not Pythonic. Inefficient because retrieving an element given the position can be an expensive operation, and we're doing it from scratch at each iteration. The mail man doesn't go back to the beginning of the road each time he delivers a letter, right? He moves from house to house. From one to the next one. Let's try to make it better using enumerate:

multiple.sequences.enumerate.py

```
people = ['Jonas', 'Julio', 'Mike', 'Mez']
ages = [25, 30, 31, 39]
for position, person in enumerate(people):
    age = ages[position]
    print(person, age)
```

Better, but still not perfect. And still a bit ugly. We're iterating properly on `people`, but we're still fetching `age` using positional indexing, which we want to lose as well. Well, no worries, Python gives you the `zip` function, remember? Let's use it!

`multiple.sequences.zip.py`

```
people = ['Jonas', 'Julio', 'Mike', 'Mez']
ages = [25, 30, 31, 39]
for person, age in zip(people, ages):
    print(person, age)
```

Ah! So much better! Once again, compare the preceding code with the first example and admire Python's elegance. The reason I wanted to show this example is twofold. On the one hand, I wanted to give you an idea of how shorter the code in Python can be compared to other languages where the syntax doesn't allow you to iterate over sequences or collections as easily. And on the other hand, and much more importantly, notice that when the `for` loop asks `zip(sequenceA, sequenceB)` for the next element, it gets back a `tuple`, not just a single object. It gets back a `tuple` with as many elements as the number of sequences we feed to the `zip` function. Let's expand a little on the previous example in two ways: using explicit and implicit assignment:

`multiple.sequences.explicit.py`

```
people = ['Jonas', 'Julio', 'Mike', 'Mez']
ages = [25, 30, 31, 39]
nationalities = ['Belgium', 'Spain', 'England', 'Bangladesh']
for person, age, nationality in zip(people, ages, nationalities):
    print(person, age, nationality)
```

In the preceding code, we added the nationalities list. Now that we feed three sequences to the `zip` function, the for loop gets back a *3-tuple* at each iteration. Notice that the position of the elements in the tuple respects the position of the sequences in the `zip` call. Executing the code will yield the following result:

```
$ python multiple.sequences.explicit.py
Jonas 25 Belgium
Julio 30 Spain
Mike 31 England
Mez 39 Bangladesh
```

Sometimes, for reasons that may not be clear in a simple example like the preceding one, you may want to explode the tuple within the body of the `for` loop. If that is your desire, it's perfectly possible to do so.

multiple.sequences.implicit.py

```
people = ['Jonas', 'Julio', 'Mike', 'Mez']
ages = [25, 30, 31, 39]
nationalities = ['Belgium', 'Spain', 'England', 'Bangladesh']
for data in zip(people, ages, nationalities):
    person, age, nationality = data
    print(person, age, nationality)
```

It's basically doing what the `for` loop does automatically for you, but in some cases you may want to do it yourself. Here, the 3-tuple `data` that comes from `zip(...)`, is exploded within the body of the `for` loop into three variables: `person`, `age`, and `nationality`.

The while loop

In the preceding pages, we saw the `for` loop in action. It's incredibly useful when you need to loop over a sequence or a collection. The key point to keep in mind, when you need to be able to discriminate which looping construct to use, is that the `for` loop rocks when you have to iterate over a finite amount of elements. It can be a huge amount, but still, something that at some point ends.

There are other cases though, when you just need to loop until some condition is satisfied, or even loop indefinitely until the application is stopped. Cases where we don't really have something to iterate on, and therefore the `for` loop would be a poor choice. But fear not, for these cases Python provides us with the `while` loop.

The `while` loop is similar to the `for` loop, in that they both loop and at each iteration they execute a body of instructions. What is different between them is that the `while` loop doesn't loop over a sequence (it can, but you have to manually write the logic and it wouldn't make any sense, you would just want to use a `for` loop), rather, it loops as long as a certain condition is satisfied. When the condition is no longer satisfied, the loop ends.

As usual, let's see an example which will clarify everything for us. We want to print the binary representation of a positive number. In order to do so, we repeatedly divide the number by two, collecting the remainder, and then produce the inverse of the list of remainders. Let me give you a small example using number 6, which is 110 in binary.

```
6 / 2 = 3 (remainder: 0)
3 / 2 = 1 (remainder: 1)
1 / 2 = 0 (remainder: 1)
List of remainders: 0, 1, 1.
Inverse is 1, 1, 0, which is also the binary representation of 6: 110
```

Let's write some code to calculate the binary representation for number 39: 100111_2.

`binary.py`

```
n = 39
remainders = []
while n > 0:
    remainder = n % 2  # remainder of division by 2
    remainders.append(remainder)  # we keep track of remainders
    n //= 2  # we divide n by 2

# reassign the list to its reversed copy and print it
remainders = remainders[::-1]
print(remainders)
```

In the preceding code, I highlighted two things: `n > 0`, which is the condition to keep looping, and `remainders[::-1]` which is a nice and easy way to get the reversed version of a list (missing `start` and `end` parameters, `step = -1`, produces the same list, from `end` to `start`, in reverse order). We can make the code a little shorter (and more Pythonic), by using the `divmod` function, which is called with a number and a divisor, and returns a tuple with the result of the integer division and its remainder. For example, `divmod(13, 5)` would return `(2, 3)`, and indeed *5 * 2 + 3 = 13*.

`binary.2.py`

```
n = 39
remainders = []
while n > 0:
    n, remainder = divmod(n, 2)
    remainders.append(remainder)

# reassign the list to its reversed copy and print it
remainders = remainders[::-1]
print(remainders)
```

In the preceding code, we have reassigned n to the result of the division by 2, and the remainder, in one single line.

Notice that the condition in a `while` loop is a condition to continue looping. If it evaluates to `True`, then the body is executed and then another evaluation follows, and so on, until the condition evaluates to `False`. When that happens, the loop is exited immediately without executing its body.

If the condition never evaluates to `False`, the loop becomes a so called **infinite loop**. Infinite loops are used for example when polling from network devices: you ask the socket if there is any data, you do something with it if there is any, then you sleep for a small amount of time, and then you ask the socket again, over and over again, without ever stopping.

Having the ability to loop over a condition, or to loop indefinitely, is the reason why the `for` loop alone is not enough, and therefore Python provides the `while` loop.

By the way, if you need the binary representation of a number, checkout the `bin` function.

Just for fun, let's adapt one of the examples (`multiple.sequences.py`) using the while logic.

`multiple.sequences.while.py`

```
people = ['Jonas', 'Julio', 'Mike', 'Mez']
ages = [25, 30, 31, 39]
position = 0
while position < len(people):
    person = people[position]
    age = ages[position]
    print(person, age)
    position += 1
```

In the preceding code, I have highlighted the *initialization*, *condition*, and *update* of the variable `position`, which makes it possible to simulate the equivalent `for` loop code by handling the iteration variable manually. Everything that can be done with a `for` loop can also be done with a `while` loop, even though you can see there's a bit of boilerplate you have to go through in order to achieve the same result. The opposite is also true, but simulating a never ending `while` loop using a `for` loop requires some real trickery, so why would you do that? Use the right tool for the job, and 99.9% of the times you'll be fine.

So, to recap, use a `for` loop when you need to iterate over one (or a combination of) iterable, and a `while` loop when you need to loop according to a condition being satisfied or not. If you keep in mind the difference between the two purposes, you will never choose the wrong looping construct.

Let's now see how to alter the normal flow of a loop.

The break and continue statements

According to the task at hand, sometimes you will need to alter the regular flow of a loop. You can either skip a single iteration (as many times you want), or you can break out of the loop entirely. A common use case for skipping iterations is for example when you're iterating over a list of items and you need to work on each of them only if some condition is verified. On the other hand, if you're iterating over a collection of items, and you have found one of them that satisfies some need you have, you may decide not to continue the loop entirely and therefore break out of it. There are countless possible scenarios, so it's better to see a couple of examples.

Let's say you want to apply a 20% discount to all products in a basket list for those which have an expiration date of today. The way you achieve this is to use the **continue** statement, which tells the looping construct (`for` or `while`) to immediately stop execution of the body and go to the next iteration, if any. This example will take us a little deeper down the rabbit whole, so be ready to jump.

`discount.py`

```
from datetime import date, timedelta

today = date.today()
tomorrow = today + timedelta(days=1)  # today + 1 day is tomorrow
products = [
    {'sku': '1', 'expiration_date': today, 'price': 100.0},
    {'sku': '2', 'expiration_date': tomorrow, 'price': 50},
    {'sku': '3', 'expiration_date': today, 'price': 20},
]
for product in products:
    if product['expiration_date'] != today:
        continue
    product['price'] *= 0.8  # equivalent to applying 20% discount
    print(
        'Price for sku', product['sku'],
        'is now', product['price'])
```

You see we start by importing the `date` and `timedelta` objects, then we set up our products. Those with sku `1` and `3` have an expiration date of `today`, which means we want to apply 20% discount on them. We loop over each `product` and we inspect the expiration date. If it is not (inequality operator, `!=`) `today`, we don't want to execute the rest of the body suite, so we `continue`.

Notice that is not important where in the body suite you place the `continue` statement (you can even use it more than once). When you reach it, execution stops and goes back to the next iteration. If we run the `discount.py` module, this is the output:

```
$ python discount.py
Price for sku 1 is now 80.0
Price for sku 3 is now 16.0
```

Which shows you that the last two lines of the body haven't been executed for sku number 2.

Let's now see an example of breaking out of a loop. Say we want to tell if at least any of the elements in a list evaluates to `True` when fed to the `bool` function. Given that we need to know if there is at least one, when we find it we don't need to keep scanning the list any further. In Python code, this translates to using the **break** statement. Let's write this down into code:

any.py

```
items = [0, None, 0.0, True, 0, 7]  # True and 7 evaluate to True
found = False  # this is called "flag"
for item in items:
    print('scanning item', item)
    if item:
        found = True  # we update the flag
        break

if found:  # we inspect the flag
    print('At least one item evaluates to True')
else:
    print('All items evaluate to False')
```

The preceding code is such a common pattern in programming, you will see it a lot. When you inspect items this way, basically what you do is to set up a `flag` variable, then start the inspection. If you find one element that matches your criteria (in this example, that evaluates to `True`), then you update the flag and stop iterating. After iteration, you inspect the flag and take action accordingly. Execution yields:

```
$ python any.py
scanning item 0
scanning item None
scanning item 0.0
scanning item True
At least one item evaluates to True
```

See how execution stopped after `True` was found?

The `break` statement acts exactly like the `continue` one, in that it stops executing the body of the loop immediately, but also, prevents any other iteration to run, effectively breaking out of the loop.

The `continue` and `break` statements can be used together with no limitation in their number, both in the `for` and `while` looping constructs.

By the way, there is no need to write code to detect if there is at least one element in a sequence that evaluates to `True`. Just check out the `any` built-in function.

A special else clause

One of the features I've seen only in the Python language is the ability to have `else` clauses after `while` and `for` loops. It's very rarely used, but it's definitely nice to have. In short, you can have an `else` suite after a `for` or `while` loop. If the loop ends normally, because of exhaustion of the iterator (`for` loop) or because the condition is finally not met (`while` loop), then the `else` suite (if present) is executed. In case execution is interrupted by a `break` statement, the `else` clause is not executed. Let's take an example of a `for` loop that iterates over a group of items, looking for one that would match some condition. In case we don't find at least one that satisfies the condition, we want to raise an **exception**. This means we want to arrest the regular execution of the program and signal that there was an error, or exception, that we cannot deal with. Exceptions will be the subject of *Chapter 7, Testing, Profiling, and Dealing with Exceptions*, so don't worry if you don't fully understand them now. Just bear in mind that they will alter the regular flow of the code. Let me now show you two examples that do exactly the same thing, but one of them is using the special `for ... else` syntax. Say that we want to find among a collection of people one that could drive a car.

`for.no.else.py`

```
class DriverException(Exception):
    pass

people = [('James', 17), ('Kirk', 9), ('Lars', 13), ('Robert', 8)]
driver = None
for person, age in people:
    if age >= 18:
```

```
            driver = (person, age)
            break

    if driver is None:
        raise DriverException('Driver not found.')
```

Notice the `flag` pattern again. We set driver to be `None`, then if we find one we update the `driver` flag, and then, at the end of the loop, we inspect it to see if one was found. I kind of have the feeling that those kids would drive a very metallic car, but anyway, notice that if a driver is not found, a `DriverException` is raised, signaling the program that execution cannot continue (we're lacking the driver).

The same functionality can be rewritten a bit more elegantly using the following code:

`for.else.py`

```
    class DriverException(Exception):
        pass

    people = [('James', 17), ('Kirk', 9), ('Lars', 13), ('Robert', 8)]
    for person, age in people:
        if age >= 18:
            driver = (person, age)
            break
    else:
        raise DriverException('Driver not found.')
```

Notice that we aren't forced to use the `flag` pattern any more. The exception is raised as part of the `for` loop logic, which makes good sense because the `for` loop is checking on some condition. All we need is to set up a `driver` object in case we find one, because the rest of the code is going to use that information somewhere. Notice the code is shorter and more elegant, because the logic is now correctly grouped together where it belongs.

Putting this all together

Now that you have seen all there is to see about conditionals and loops, it's time to spice things up a little, and see those two examples I anticipated at the beginning of this chapter. We'll mix and match here, so you can see how one can use all these concepts together. Let's start by writing some code to generate a list of prime numbers up to some limit. Please bear in mind that I'm going to write a very inefficient and rudimentary algorithm to detect primes. The important thing for you is to concentrate on those bits in the code that belong to this chapter's subject.

Example 1 – a prime generator

According to Wikipedia:

> "*A prime number (or a prime) is a natural number greater than 1 that has no positive divisors other than 1 and itself. A natural number greater than 1 that is not a prime number is called a composite number.*"

Based on this definition, if we consider the first 10 natural numbers, we can see that 2, 3, 5, and 7 are primes, while 1, 4, 6, 8, 9, 10 are not. In order to have a computer tell you if a number *N* is prime, you can divide that number by all natural numbers in the range [2, *N*). If any of those divisions yields zero as a remainder, then the number is not a prime. Enough chatter, let's get down to business. I'll write two versions of this, the second of which will exploit the `for ... else` syntax.

`primes.py`

```
primes = []  # this will contain the primes in the end
upto = 100  # the limit, inclusive
for n in range(2, upto + 1):
    is_prime = True  # flag, new at each iteration of outer for
    for divisor in range(2, n):
        if n % divisor == 0:
            is_prime = False
            break
    if is_prime:  # check on flag
        primes.append(n)
print(primes)
```

Lots of things to notice in the preceding code. First of all we set up an empty list `primes`, which will contain the primes at the end. The limit is 100, and you can see it's inclusive in the way we call `range()` in the outer loop. If we wrote `range(2, upto)` that would be [2, upto), right? Therefore `range(2, upto + 1)` gives us *[2, upto + 1) == [2, upto]*.

So, two `for` loops. In the outer one we loop over the candidate primes, that is, all natural numbers from 2 to `upto`. Inside each iteration of this outer loop we set up a flag (which is set to `True` at each iteration), and then start dividing the current `n` by all numbers from 2 to *n* – 1. If we find a proper divisor for `n`, it means `n` is composite, and therefore we set the flag to `False` and break the loop. Notice that when we break the inner one, the outer one keeps on going normally. The reason why we break after having found a proper divisor for `n` is that we don't need any further information to be able to tell that `n` is not a prime.

When we check on the `is_prime` flag, if it is still `True`, it means we couldn't find any number in [2, *n*) that is a proper divisor for `n`, therefore `n` is a prime. We append `n` to the `primes` list, and hop! Another iteration, until *n* equals 100.

Running this code yields:

```
$ python primes.py
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67,
71, 73, 79, 83, 89, 97]
```

Before we proceed, one question: of all iterations of the outer loop, one of them is different than all the others. Could you tell which one, and why? Think about it for a second, go back to the code and try to figure it out for yourself, and then keep reading on.

Did you figure it out? If not, don't feel bad, it's perfectly normal. I asked you to do it as a small exercise because it's what coders do all the time. The skill to understand what the code does by simply looking at it is something you build over time. It's very important, so try to exercise it whenever you can. I'll tell you the answer now: the iteration that behaves differently from all others is the first one. The reason is because in the first iteration, `n` is 2. Therefore the innermost `for` loop won't even run, because it's a `for` loop which iterates over `range(2, 2)`, and what is that if not [2, 2)? Try it out for yourself, write a simple `for` loop with that iterable, put a `print` in the body suite, and see if anything happens (it won't...).

Now, from an algorithmic point of view this code is inefficient so let's at least make it more beautiful:

primes.else.py

```
primes = []
upto = 100
for n in range(2, upto + 1):
    for divisor in range(2, n):
        if n % divisor == 0:
            break
    else:
        primes.append(n)
print(primes)
```

Much nicer, right? The `is_prime` flag is completely gone, and we append `n` to the `primes` list when we know the inner `for` loop hasn't encountered any `break` statements. See how the code looks cleaner and reads better?

Example 2 – applying discounts

In this example, I want to show you a technique I like a lot. In many programming languages, other than the `if/elif/else` constructs, in whatever form or syntax they may come, you can find another statement, usually called `switch/case`, that in Python is missing. It is the equivalent of a cascade of `if/elif/.../elif/else` clauses, with a syntax similar to this (warning! JavaScript code!):

`switch.js`

```
switch (day_number) {
    case 1:
    case 2:
    case 3:
    case 4:
    case 5:
        day = "Weekday";
        break;
    case 6:
        day = "Saturday";
        break;
    case 0:
        day = "Sunday";
        break;
    default:
        day = "";
        alert(day_number + ' is not a valid day number.')
}
```

In the preceding code, we `switch` on a variable called `day_number`. This means we get its value and then we decide what case it fits in (if any). From 1 to 5 there is a cascade, which means no matter the number, [1, 5] all go down to the bit of logic that sets `day` as `"Weekday"`. Then we have single cases for 0 and 6 and a `default` case to prevent errors, which alerts the system that `day_number` is not a valid day number, that is, not in [0, 6]. Python is perfectly capable of realizing such logic using `if/elif/else` statements:

`switch.py`

```
if 1 <= day_number <= 5:
    day = 'Weekday'
elif day_number == 6:
    day = 'Saturday'
elif day_number == 0:
    day = 'Sunday'
```

```
    else:
        day = ''
        raise ValueError(
            str(day_number) + ' is not a valid day number.')
```

In the preceding code, we reproduce the same logic of the JavaScript snippet, in Python, using `if/elif/else` statements. I raised `ValueError` exception just as an example at the end, if `day_number` is not in [0, 6]. This is one possible way of translating the `switch/case` logic, but there is also another one, sometimes called dispatching, which I will show you in the last version of the next example.

By the way, did you notice the first line of the previous snippet? Have you noticed that Python can make double (actually, even multiple) comparisons? It's just wonderful!

Let's start the new example by simply writing some code that assigns a discount to customers based on their coupon value. I'll keep the logic down to a minimum here, remember that all we really care about is conditionals and loops.

coupons.py

```
    customers = [
        dict(id=1, total=200, coupon_code='F20'),  # F20: fixed, £20
        dict(id=2, total=150, coupon_code='P30'),  # P30: percent, 30%
        dict(id=3, total=100, coupon_code='P50'),  # P50: percent, 50%
        dict(id=4, total=110, coupon_code='F15'),  # F15: fixed, £15
    ]
    for customer in customers:
        code = customer['coupon_code']
        if code == 'F20':
            customer['discount'] = 20.0
        elif code == 'F15':
            customer['discount'] = 15.0
        elif code == 'P30':
            customer['discount'] = customer['total'] * 0.3
        elif code == 'P50':
            customer['discount'] = customer['total'] * 0.5
        else:
            customer['discount'] = 0.0

    for customer in customers:
        print(customer['id'], customer['total'], customer['discount'])
```

We start by setting up some customers. They have an order total, a coupon code, and an id. I made up four different types of coupon, two are fixed and two are percentage based. You can see that in the `if/elif/else` cascade I apply the discount accordingly, and I set it as a `'discount'` key in the `customer` dict.

At the end I just print out part of the data to see if my code is working properly.

```
$ python coupons.py
1 200 20.0
2 150 45.0
3 100 50.0
4 110 15.0
```

This code is simple to understand, but all those clauses are kind of cluttering the logic. It's not easy to see what's going on at a first glance, and I don't like it. In cases like this, you can exploit a dictionary to your advantage, like this:

coupons.dict.py

```
customers = [
    dict(id=1, total=200, coupon_code='F20'),  # F20: fixed, £20
    dict(id=2, total=150, coupon_code='P30'),  # P30: percent, 30%
    dict(id=3, total=100, coupon_code='P50'),  # P50: percent, 50%
    dict(id=4, total=110, coupon_code='F15'),  # F15: fixed, £15
]
discounts = {
    'F20': (0.0, 20.0),  # each value is (percent, fixed)
    'P30': (0.3, 0.0),
    'P50': (0.5, 0.0),
    'F15': (0.0, 15.0),
}
for customer in customers:
    code = customer['coupon_code']
    percent, fixed = discounts.get(code, (0.0, 0.0))
    customer['discount'] = percent * customer['total'] + fixed

for customer in customers:
    print(customer['id'], customer['total'], customer['discount'])
```

Running the preceding code yields exactly the same result we had from the snippet before it. We spared two lines, but more importantly, we gained a lot in readability, as the body of the `for` loop now is just three lines long, and very easy to understand. The concept here is to use a dictionary as **dispatcher**. In other words, we try to fetch something from the dictionary based on a code (our `coupon_code`), and by using `dict.get(key, default)`, we make sure we also cater for when the `code` is not in the dictionary and we need a default value.

Notice that I had to apply some very simple linear algebra in order to calculate the discount properly. Each discount has a percentage and fixed part in the dictionary, represented by a 2-tuple. By applying `percent * total + fixed`, we get the correct discount. When `percent` is `0`, the formula just gives the fixed amount, and it gives `percent * total` when fixed is `0`. Simple but effective.

This technique is important because it is also used in other contexts, with functions, where it actually becomes much more powerful than what we've seen in the preceding snippet. If it's not completely clear to you how it works, I suggest you to take your time and experiment with it. Change values and add print statements to see what's going on while the program is running.

A quick peek at the itertools module

A chapter about iterables, iterators, conditional logic, and looping wouldn't be complete without spending a few words about the `itertools` module. If you are into iterating, this is a kind of heaven.

According to the Python official documentation, the `itertools` module is:

> *"A module which implements a number of iterator building blocks inspired by constructs from APL, Haskell, and SML. Each has been recast in a form suitable for Python. The module standardizes a core set of fast, memory efficient tools that are useful by themselves or in combination. Together, they form an "iterator algebra" making it possible to construct specialized tools succinctly and efficiently in pure Python."*

By no means do I have the room here to show you all the goodies you can find in this module, so I encourage you to go and check it out for yourself, I promise you'll enjoy it.

In a nutshell, it provides you with three broad categories of iterators. I will give you a very small example of one iterator taken from each one of them, just to make your mouth water a little.

Infinite iterators

Infinite iterators allow you to work with a `for` loop in a different fashion, like if it was a `while` loop.

`infinite.py`

```
from itertools import count
for n in count(5, 3):
    if n > 20:
        break
    print(n, end=', ')  # instead of newline, comma and space
```

Running the code gives this:

```
$ python infinite.py
5, 8, 11, 14, 17, 20,
```

The `count` factory class makes an iterator that just goes on and on counting. It starts from 5 and keeps adding 3 to it. We need to manually break it if we don't want to get stuck in an infinite loop.

Iterators terminating on the shortest input sequence

This category is very interesting. It allows you to create an iterator based on multiple iterators, combining their values according to some logic. The key point here is that among those iterators, in case any of them are shorter than the rest, the resulting iterator won't break, it will simply stop as soon as the shortest iterator is exhausted. This is very theoretical, I know, so let me give you an example using `compress`. This iterator gives you back the data according to a corresponding item in a selector being `True` or `False`:

`compress('ABC', (1, 0, 1))` would give back `'A'` and `'C'`, because they correspond to the `1's`. Let's see a simple example:

`compress.py`

```
from itertools import compress
data = range(10)
even_selector = [1, 0] * 10
odd_selector = [0, 1] * 10

even_numbers = list(compress(data, even_selector))
odd_numbers = list(compress(data, odd_selector))
```

```
print(odd_selector)
print(list(data))
print(even_numbers)
print(odd_numbers)
```

Notice that `odd_selector` and `even_selector` are 20 elements long, while `data` is just 10 elements long. `compress` will stop as soon as `data` has yielded its last element. Running this code produces the following:

```
$ python compress.py
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 2, 4, 6, 8]
[1, 3, 5, 7, 9]
```

It's a very fast and nice way of selecting elements out of an iterable. The code is very simple, just notice that instead of using a `for` loop to iterate over each value that is given back by the compress calls, we used `list()`, which does the same, but instead of executing a body of instructions, puts all the values into a list and returns it.

Combinatoric generators

Last but not least, combinatoric generators. These are really fun, if you are into this kind of thing. Let's just see a simple example on permutations.

According to Wolfram Mathworld:

> *"A permutation, also called an "arrangement number" or "order", is a rearrangement of the elements of an ordered list S into a one-to-one correspondence with S itself."*

For example, the permutations of ABC are 6: ABC, ACB, BAC, BCA, CAB, and CBA.

If a set has *N* elements, then the number of permutations of them is *N!* (*N* factorial). For the string ABC the permutations are *3! = 3 * 2 * 1 = 6*. Let's do it in Python:

`permutations.py`

```
from itertools import permutations
print(list(permutations('ABC')))
```

This very short snippet of code produces the following result:

```
$ python permutations.py
[('A', 'B', 'C'), ('A', 'C', 'B'), ('B', 'A', 'C'), ('B', 'C', 'A'),
('C', 'A', 'B'), ('C', 'B', 'A')]
```

Be very careful when you play with permutation. Their number grows at a rate that is proportional to the factorial of the number of the elements you're permuting, and that number can get really big, really fast.

Summary

In this chapter, we've taken another step forward to expand our coding vocabulary. We've seen how to drive the execution of the code by evaluating conditions, and we've seen how to loop and iterate over sequences and collections of objects. This gives us the power to control what happens when our code is run, which means we are getting an idea on how to shape it so that it does what we want and it reacts to data that changes dynamically.

We've also seen how to combine everything together in a couple of simple examples, and in the end we have taken a brief look at the `itertools` module, which is full of interesting iterators which can enrich our abilities with Python even more.

Now it's time to switch gears, to take another step forward and talk about functions. The next chapter is all about them because they are extremely important. Make sure you're comfortable with what has been done up to now: I want to provide you with interesting examples, so I'll have to go a little faster. Ready? Turn the page.

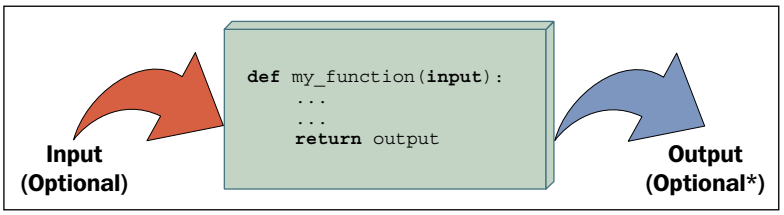
4
Functions, the Building Blocks of Code

"To create architecture is to put in order. Put what in order? Function and objects."

- Le Corbusier

In this chapter, we're going to explore functions. We already said that everything is an object in Python, and functions are no exception to this. But, what exactly is a function? A **function** is a sequence of instructions that perform a task, bundled as a unit. This unit can then be imported and used wherever it's needed. There are many advantages to using functions in your code, as we'll see shortly.

I believe the saying, *a picture is worth one thousand words*, is particularly true when explaining functions to someone who is new to this concept, so please take a look at the following image:



As you can see, a function is a block of instructions, packaged as a whole, like a box. Functions can accept input arguments and produce output values. Both of these are optional, as we'll see in the examples in this chapter.

A function in Python is defined by using the `def` keyword, after which the name of the function follows, terminated by a pair of braces (which may or may not contain input parameters) and, finally, a colon (`:`) signals the end of the function definition line. Immediately afterwards, indented by four spaces, we find the body of the function, which is the set of instructions that the function will execute when called.

Note that the indentation by four spaces is not mandatory, but it is the amount of spaces suggested by **PEP8**, and, in practice, it is the most widely used spacing measure.

A function may or may not return output. If a function wants to return output, it does so by using the `return` keyword, followed by the desired output. If you have an eagle eye, you may have noticed the little ***** after **Optional** in the output section of the preceding picture. This is because a function always returns something in Python, even if you don't explicitly use the `return` clause. If the function has no `return` statement in its body, it's return value is `None`. The reasons behind this design choice are out of the scope of an introductory chapter, so all you need to know is that this behavior will make your life easier, as always, thank you Python.

Why use functions?

Functions are among the most important concepts and constructs of any language, so let me give you a few reasons why we need them:

- They reduce code duplication in a program. By having a specific task taken care of by a nice block of packaged code that we can import and call whenever we want, we don't need to duplicate its implementation.
- They help in splitting a complex task or procedure into smaller blocks, each of which becomes a function.
- They hide the implementation details from their users.
- They improve traceability.
- They improve readability.

Let's look at a few examples to get a better understanding of each point.

Reduce code duplication

Imagine that you are writing a piece of scientific software, and you need to calculate primes up to a limit, as we did in the previous chapter. You write several algorithms and prime numbers, being the basis of many different types of calculations, keep creeping into your code. Well, you have a nice algorithm to calculate them, so you copy and paste it to wherever you need. One day, though, your friend *Mister Smarty* gives you a better algorithm to calculate prime numbers, and this will save you a lot of time. At this point, you need to go over your whole codebase and replace the old code with the new code.

This is actually a very bad way to go about it. It's error-prone, you never know what lines you are chopping out or leaving there by mistake when you cut and paste code in other code, and you may also risk missing one of the places where prime calculation was done, leaving your software with different versions. Can you imagine if you discovered that the old way was buggy? You would have an undetected bug in your code, and bugs like this are quite hard to spot, especially in big codebases.

So, what should you do? Simple! You write a function, `get_prime_numbers(upto)`, and use it anywhere you need a list of primes. When *Mister Smarty* comes to you and gives you the new code, all you have to do is replace the body of that function with the new implementation, and you're done! The rest of the software will automatically adapt, since it's just calling the function.

Your code will be shorter, it will not suffer from inconsistencies between old and new ways of performing a task, or undetected bugs due to copy and paste failures or oversights. Use functions, and you'll only gain from it, I promise.

Splitting a complex task

Functions are very useful also to split a long or complex task into smaller pieces. The end result is that the code benefits from it in several ways, for example, readability, testability, and reuse. To give you a simple example, imagine that you're preparing a report. Your code needs to fetch data from a data source, parse it, filter it, polish it, and then a whole series of algorithms needs to be run against it, in order to produce the results which will feed the `Report` class. It's not uncommon to read procedures like this that are just one big function `do_report(data_source)`. There are tens or hundreds of lines of code which end with `return report`.

Situations like this are common in code produced by scientists. They have brilliant minds and they care about the correctness of the end result but, unfortunately, sometimes they have no training in programming theory. It is not their fault, one cannot know everything. Now, picture in your head something like a few hundred lines of code. It's very hard to follow through, to find the places where things are changing context (like finishing one task and starting the next one). Do you have the picture in your mind? Good. Don't do it! Instead, look at this code:

`data.science.example.py`

```
    def do_report(data_source):
        # fetch and prepare data
        data = fetch_data(data_source)
        parsed_data = parse_data(data)
        filtered_data = filter_data(parsed_data)
        polished_data = polish_data(filtered_data)

        # run algorithms on data
        final_data = analyse(polished_data)

        # create and return report
        report = Report(final_data)
        return report
```

The previous example is fictitious, of course, but can you see how easy it would be to go through the code? If the end result looks wrong, it would be very easy to debug each of the single data outputs in the `do_report` function. Moreover, it's even easier to exclude part of the process temporarily from the whole procedure (you just need to comment out the parts you need to suspend). Code like this is easier to deal with.

Hide implementation details

Let's stay with the preceding example to talk about this point as well. You can see that, by going through the code of the `do_report` function, you can get a pretty good understanding without reading one single line of implementation. This is because functions hide the implementation details. This feature means that, if you don't need to delve into details, you are not forced to, in the way you would if `do_report` was just one big fat function. In order to understand what was going on, you would have to read the implementation details. You don't need to with functions. This reduces the time you spend reading the code and since, in a professional environment, reading code takes much more time than actually writing it, it's very important to reduce it as much as we can.

Improve readability

Coders sometimes don't see the point in writing a function with a body of one or two lines of code, so let's look at an example that shows you why you should do it.

Imagine that you need to multiply two matrices:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} 5 & 1 \\ 2 & 1 \end{pmatrix} = \begin{pmatrix} 9 & 3 \\ 23 & 7 \end{pmatrix}$$

Would you prefer to have to read this code:

`matrix.multiplication.nofunc.py`

```
a = [[1, 2], [3, 4]]
b = [[5, 1], [2, 1]]
c = [[sum(i * j for i, j in zip(r, c)) for c in zip(*b)]
     for r in a]
```

Or would you prefer this one:

`matrix.multiplication.func.py`

```
# this function could also be defined in another module
def matrix_mul(a, b):
    return [[sum(i * j for i, j in zip(r, c)) for c in zip(*b)]
            for r in a]

a = [[1, 2], [3, 4]]
b = [[5, 1], [2, 1]]
c = matrix_mul(a, b)
```

It's much easier to understand that `c` is the result of the multiplication between `a` and `b` in the second example. It's much easier to read through the code and, if you don't need to modify that part, you don't even need to go into the implementation details.

Therefore, readability is improved here while, in the first snippet, you would have to spend time trying to understand what that complicated list comprehension was doing.

Don't worry if you don't understand *list comprehensions*, we'll study them in the next chapter.

Improve traceability

Imagine that you have written an e-commerce website. You have displayed the product prices all over the pages. Imagine that the prices in your database are stored with no VAT, but you want to display them on the website with VAT at 20%. Here's a few ways of calculating the VAT-inclusive price from the VAT-exclusive price.

`vat.py`

```
price = 100  # GBP, no VAT
final_price1 = price * 1.2
final_price2 = price + price / 5.0
final_price3 = price * (100 + 20) / 100.0
final_price4 = price + price * 0.2
```

All these four different ways of calculating a VAT-inclusive price are perfectly acceptable, and I promise you I have found them all in my colleagues' code, over the years. Now, imagine that you have started selling your products in different countries and some of them have different VAT rates so you need to refactor your code (throughout the website) in order to make that VAT calculation dynamic.

How do you trace all the places in which you are performing a VAT calculation? Coding today is a collaborative task and you cannot be sure the VAT has been calculated using only one of those forms. It's going to be hell, believe me.

So, let's write a function that takes the input values, `vat` and `price` (VAT-exclusive), and returns a VAT-inclusive price.

`vat.function.py`

```
def calculate_price_with_vat(price, vat):
    return price * (100 + vat) / 100
```

Now you can import that function and apply it in any place of your website where you need to calculate a VAT-inclusive price and when you need to trace those calls, you can search for `calculate_price_with_vat`.

Note that, in the preceding example, `price` is assumed to be VAT-exclusive, and `vat` has a percentage value (for example, 19, 20, 23, and so on).

Scopes and name resolution

Do you remember when we talked about scopes and namespaces in the first chapter? We're going to expand on that concept now. Finally, we can talk about functions and this will make everything easier to understand. Let's start with a very simple example.

scoping.level.1.py

```
def my_function():
    test = 1 # this is defined in the local scope of the function
    print('my_function:', test)

test = 0  # this is defined in the global scope
my_function()
print('global:', test)
```

I have defined the name `test` in two different places in the previous example. It is actually in two different scopes. One is the global scope (`test = 0`), and the other is the local scope of the function `my_function` (`test = 1`). If you execute the code, you'll see this:

```
$ python scoping.level.1.py
my_function: 1
global: 0
```

It's clear that `test = 1` shadows the assignment `test = 0` in `my_function`. In the global context, `test` is still `0`, as you can see from the output of the program but we define the name `test` again in the function body, and we set it to point to an integer of value `1`. Both the two `test` names therefore exist, one in the global scope, pointing to an `int` object with value 0, the other in the `my_function` scope, pointing to an `int` object with value 1. Let's comment out the line with `test = 1`. Python goes and searches for the name `test` in the next enclosing namespace (recall the *LEGB* rule: *Local*, *Enclosing*, *Global*, *Built-in* described in *Chapter 1*, *Introduction and First Steps – Take a Deep Breath*) and, in this case, we will see the value `0` printed twice. Try it in your code.

Now, let's raise the stakes here and level up:

scoping.level.2.py

```
def outer():
    test = 1  # outer scope

    def inner():
        test = 2  # inner scope
```

```
            print('inner:', test)
        inner()
        print('outer:', test)
test = 0  # global scope
outer()
print('global:', test)
```

In the preceding code, we have two levels of shadowing. One level is in the function `outer`, and the other one is in the function `inner`. It is far from rocket science, but it can be tricky. If we run the code, we get:

```
$ python scoping.level.2.py
inner: 2
outer: 1
global: 0
```

Try commenting out the line `test = 1`. What do you think the result will be? Well, when reaching the line `print('outer:', test)`, Python will have to look for `test` in the next enclosing scope, therefore it will find and print `0`, instead of `1`. Make sure you comment out `test = 2` as well, to see if you understand what happens, and if the LEGB rule is clear, before proceeding.

Another thing to note is that Python gives you the ability to define a function in another function. The inner function's name is defined within the namespace of the outer function, exactly as would happen with any other name.

The global and nonlocal statements

Going back to the preceding example, we can alter what happens to the shadowing of the test name by using one of these two special statements: `global` and `nonlocal`. As you can see from the previous example, when we define `test = 2` in the function `inner`, we overwrite `test` neither in the function `outer`, nor in the global scope. We can get read access to those names if we use them in a nested scope that doesn't define them, but we cannot modify them because, when we write an assignment instruction, we're actually defining a new name in the current scope.

How do we change this behavior? Well, we can use the `nonlocal` statement. According to the official documentation:

> "*The* `nonlocal` *statement causes the listed identifiers to refer to previously bound variables in the nearest enclosing scope excluding globals.*"

Let's introduce it in the function `inner`, and see what happens:

`scoping.level.2.nonlocal.py`

```
def outer():
    test = 1  # outer scope

    def inner():
        nonlocal test
        test = 2  # nearest enclosing scope
        print('inner:', test)
    inner()
    print('outer:', test)

test = 0  # global scope
outer()
print('global:', test)
```

Notice how in the body of the function `inner` I have declared the `test` name to be `nonlocal`. Running this code produces the following result:

```
$ python scoping.level.2.nonlocal.py
inner: 2
outer: 2
global: 0
```

Wow, look at that result! It means that, by declaring `test` to be `nonlocal` in the function `inner`, we actually get to bind the name `test` to that declared in the function `outer`. If we removed the `nonlocal test` line from the function `inner` and tried the same trick in the function `outer`, we would get a `SyntaxError`, because the `nonlocal` statement works on enclosing scopes excluding the global one.

Is there a way to get to that `test = 0` in the global namespace then? Of course, we just need to use the `global` statement. Let's try it.

`scoping.level.2.global.py`

```
def outer():
    test = 1  # outer scope

    def inner():
        global test
        test = 2  # global scope
        print('inner:', test)
    inner()
```

```
        print('outer:', test)

    test = 0  # global scope
    outer()
    print('global:', test)
```

Note that we have now declared the name `test` to be `global`, which will basically bind it to the one we defined in the global namespace (`test = 0`). Run the code and you should get the following:

```
$ python scoping.level.2.global.py
inner: 2
outer: 1
global: 2
```

This shows that the name affected by the assignment `test = 2` is now the `global` one. This trick would also work in the `outer` function because, in this case, we're referring to the global scope. Try it for yourself and see what changes, get comfortable with scopes and name resolution, it's very important.

Input parameters

At the beginning of this chapter, we saw that a function can take input parameters. Before we delve into all possible type of parameters, let's make sure you have a clear understanding of what passing a parameter to a function means. There are three key points to keep in mind:

- Argument passing is nothing more than assigning an object to a local variable name
- Assigning an object to an argument name inside a function doesn't affect the caller
- Changing a mutable object argument in a function affects the caller

Let's look at an example for each of these points.

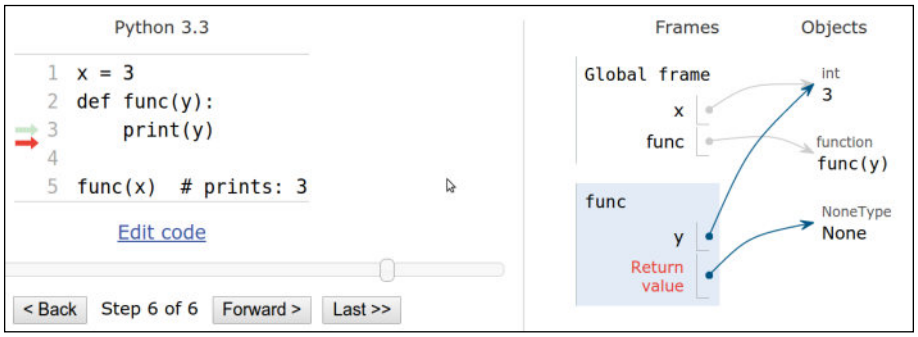
Argument passing

Take a look at the following code. We declare a name `x` in the global scope, then we declare a function `func(y)` and we call it, passing `x`. I highlighted the call in the code.

`key.points.argument.passing.py`

```
x = 3
def func(y):
    print(y)
func(x)  # prints: 3
```

When `func` is called with `x`, what happens is that within its local scope, a name `y` is created, and it's pointed to the same object `x` is pointing to. This is better clarified by the following picture:



The right part of the preceding picture depicts the state of the program when execution has reached the end, after `func` has returned (`None`). Take a look at the **Frames** column, and note that we have two names, **x** and **func**, in the global namespace (**Global frame)**, pointing to an **int** (with a value of three) and to a function object, respectively. Right below it, in the rectangle titled **func**, we can see the function's local namespace, in which only one name has been defined: **y**. Because we have called **func** with **x** (line 5 in the left part of the picture), **y** is pointing to the same object that **x** is pointing to. This is what happens under the hood when an argument is passed to a function. If we had used the name **x** instead of **y** in the function definition, things would have been exactly the same (only maybe a bit confusing at first), there would be a local **x** in the function, and a global **x** outside, as we saw in the *Scopes and name resolution* section.

So, in a nutshell, what really happens is that the function creates in its local scope the names defined as arguments and, when we call it, we basically tell Python which objects those names must be pointed towards.

Assignment to argument names don't affect the caller

This is something that can be tricky to understand at first, so let's look at an example.

`key.points.assignment.py`

```
x = 3
def func(x):
    x = 7  # defining a local x, not changing the global one

func(x)
print(x)  # prints: 3
```

In the preceding code, when the line `x = 7` is executed, what happens is that within the local scope of the function `func`, the name `x` is pointed to an integer with value 7, leaving the global `x` unaltered.

Changing a mutable affects the caller

This is the final point, and it's very important because Python apparently behaves differently with mutables (just apparently though). Let's look at an example:

`key.points.mutable.py`

```
x = [1, 2, 3]
def func(x):
    x[1] = 42  # this affects the caller!

func(x)
print(x)  # prints: [1, 42, 3]
```

Wow, we actually changed the original object! If you think about it, there is nothing weird in this behavior. The name `x` in the function is set to point to the caller object by the function call and within the body of the function, we're not changing `x`, in that we're not changing its reference, or, in other words, we are not changing the object `x` is pointing to. What we're doing is accessing that object's element at position 1, and changing its value.

Remember point #2: "*Assigning an object to an argument name within a function doesn't affect the caller*". If that is clear to you, the following code should not be surprising.

key.points.mutable.assignment.py

```
x = [1, 2, 3]
def func(x):
    x[1] = 42  # this changes the caller!
    x = 'something else'  # this points x to a new string object

func(x)
print(x)  # still prints: [1, 42, 3]
```

Take a look at the two lines I have highlighted. At first, we just access the caller object again, at position 1, and change its value to number 42. Then, we reassign `x` to point to the string `'something else'`. This leaves the caller unaltered, according to point #2, and, in fact, the output is the same as that of the previous snippet.

Take your time to play around with this concept and experiment with prints and calls to the `id` function until everything is clear in your mind. This is one of the key aspects of Python and it must be very clear, otherwise you risk introducing subtle bugs into your code.

Now that we have a good understanding of input parameters and how they behave, let's see how we can specify them.

How to specify input parameters

There are five different ways of specifying input parameters. Let's look at them one by one.

Positional arguments

Positional arguments are read from left to right and they are the most common type of arguments.

arguments.positional.py

```
def func(a, b, c):
    print(a, b, c)
func(1, 2, 3)  # prints: 1 2 3
```

There is not much else to say. They can be as numerous as you want and they are assigned by position. In the function call, `1` comes first, `2` comes second and `3` comes third, therefore they are assigned to `a`, `b` and `c` respectively.

Keyword arguments and default values

Keyword arguments are assigned by keyword using the `name=value` syntax.

`arguments.keyword.py`

```
def func(a, b, c):
    print(a, b, c)
func(a=1, c=2, b=3)  # prints: 1 3 2
```

Keyword arguments act when calling the function instead of respecting the left-to-right positional assignment, k. Keyword arguments are matched by name, even when they don't respect the definition's original position (we'll see that there is a limitation to this behavior later, when we mix and match different types of arguments).

The counterpart of keyword arguments, on the definition side, is **default values**. The syntax is the same, `name=value`, and allows us to not have to provide an argument if we are happy with the given default.

`arguments.default.py`

```
def func(a, b=4, c=88):
    print(a, b, c)

func(1)              # prints: 1 4 88
func(b=5, a=7, c=9)  # prints: 7 5 9
func(42, c=9)        # prints: 42 4 9
```

The are two things to notice, which are very important. First of all, you cannot specify a default argument on the left of a positional one. Second, note how in the examples, when an argument is passed without using the `argument_name=value` syntax, it must be the first one in the list,, and it is always assigned to `a`. Try and scramble those arguments and see what happens. Python error messages are very good at telling you what's wrong. So, for example, if you tried something like this:

```
func(b=1, c=2, 42)  # positional argument after keyword one
```

You would get the following error:

```
SyntaxError: non-keyword arg after keyword arg
```

This informs you that you've called the function incorrectly.

Variable positional arguments

Sometimes you may want to pass a variable number of positional arguments to a function and Python provides you with the ability to do it. Let's look at a very common use case, the `minimum` function. This is a function that calculates the minimum of its input values.

`arguments.variable.positional.py`

```
def minimum(*n):
    # print(n)  # n is a tuple
    if n:  # explained after the code
        mn = n[0]
        for value in n[1:]:
            if value < mn:
                mn = value
        print(mn)

minimum(1, 3, -7, 9)  # n = (1, 3, -7, 9) - prints: -7
minimum()             # n = () - prints: nothing
```

As you can see, when we specify a parameter prepending a `*` to its name, we are telling Python that that parameter will be collecting a variable number of positional arguments, according to how the function is called. Within the function, `n` is a tuple. Uncomment the `print(n)` to see for yourself and play around with it for a bit.

Have you noticed how we checked if `n` wasn't empty with a simple `if n:`? This is due to the fact that collection objects evaluate to `True` when non-empty, and otherwise `False` in Python. This is true for tuples, sets, lists, dictionaries, and so on.

One other thing to note is that we may want to throw an error when we call the function with no arguments, instead of silently doing nothing. In this context, we're not concerned about making this function robust, but in understanding variable positional arguments.

Let's make another example to show you two things that, in my experience, are confusing to those who are new to this.

`arguments.variable.positional.unpacking.py`

```
def func(*args):
    print(args)

values = (1, 3, -7, 9)
func(values)   # equivalent to: func((1, 3, -7, 9))
func(*values)  # equivalent to: func(1, 3, -7, 9)
```

Take a good look at the last two lines of the preceding example. In the first one, we call `func` with one argument, a four elements tuple. In the second example, by using the `*` syntax, we're doing something called **unpacking**, which means that the four elements tuple is unpacked, and the function is called with four arguments: `1, 3, -7, 9`.

This behavior is part of the magic Python does to allow you to do amazing things when calling functions dynamically.

Variable keyword arguments

Variable keyword arguments are very similar to variable positional arguments. The only difference is the syntax (`**` instead of `*`) and that they are collected in a dictionary. Collection and unpacking work in the same way, so let's look at an example:

`arguments.variable.keyword.py`

```
def func(**kwargs):
    print(kwargs)
# All calls equivalent. They print: {'a': 1, 'b': 42}
func(a=1, b=42)
func(**{'a': 1, 'b': 42})
func(**dict(a=1, b=42))
```

All the calls are equivalent in the preceding example. You can see that adding a `**` in front of the parameter name in the function definition tells Python to use that name to collect a variable number of keyword parameters. On the other hand, when we call the function, we can either pass `name=value` arguments explicitly, or unpack a dictionary using the same `**` syntax.

The reason why being able to pass a variable number of keyword parameters is so important may not be evident at the moment, so, how about a more realistic example? Let's define a function that connects to a database. We want to connect to a default database by simply calling this function with no parameters. We also want to connect to any other database by passing the function the appropriate arguments. Before you read on, spend a couple of minutes figuring out a solution by yourself.

`arguments.variable.db.py`

```
def connect(**options):
    conn_params = {
        'host': options.get('host', '127.0.0.1'),
        'port': options.get('port', 5432),
        'user': options.get('user', ''),
        'pwd': options.get('pwd', ''),
```

```
        }
        print(conn_params)
        # we then connect to the db (commented out)
        # db.connect(**conn_params)

    connect()
    connect(host='127.0.0.42', port=5433)
    connect(port=5431, user='fab', pwd='gandalf')
```

Note in the function we can prepare a dictionary of connection parameters (`conn_params`) in the function using default values as fallback, allowing them to be overwritten if they are provided in the function call. There are better ways to do this with fewer lines of code but we're not concerned with that now. Running the preceding code yields the following result:

```
$ python arguments.variable.db.py
{'host': '127.0.0.1', 'pwd': '', 'user': '', 'port': 5432}
{'host': '127.0.0.42', 'pwd': '', 'user': '', 'port': 5433}
{'host': '127.0.0.1', 'pwd': 'gandalf', 'user': 'fab', 'port': 5431}
```

Note the correspondence between the function calls and the output. Note how default values are either there or overridden, according to what was passed to the function.

Keyword-only arguments

Python 3 allows for a new type of parameter: the **keyword-only** parameter. We are going to study them only briefly as their use cases are not that frequent. There are two ways of specifying them, either after the variable positional arguments, or after a bare *. Let's see an example of both.

`arguments.keyword.only.py`

```
    def kwo(*a, c):
        print(a, c)

    kwo(1, 2, 3, c=7)  # prints: (1, 2, 3) 7
    kwo(c=4)           # prints: () 4
    # kwo(1, 2)  # breaks, invalid syntax, with the following error
    # TypeError: kwo() missing 1 required keyword-only argument: 'c'

    def kwo2(a, b=42, *, c):
        print(a, b, c)

    kwo2(3, b=7, c=99)  # prints: 3 7 99
```

```
kwo2(3, c=13)       # prints: 3 42 13
# kwo2(3, 23)  # breaks, invalid syntax, with the following error
# TypeError: kwo2() missing 1 required keyword-only argument: 'c'
```

As anticipated, the function, `kwo`, takes a variable number of positional arguments (`a`) and a keyword-only function, `c`. The results of the calls are straightforward and you can uncomment the third call to see what error Python returns.

The same applies to the function, `kwo2`, which differs from `kwo` in that it takes a positional argument `a`, a keyword argument `b`, and then a keyword-only argument, `c`. You can uncomment the third call to see the error.

Now that you know how to specify different types of input parameters, let's see how you can combine them in function definitions.

Combining input parameters

You can combine input parameters, as long as you follow these ordering rules:

- When defining a function, normal positional arguments come first (`name`), then any default arguments (`name=value`), then the variable positional arguments (`*name`, or simply `*`), then any keyword-only arguments (either `name` or `name=value` form is good), then any variable keyword arguments (`**name`).
- On the other hand, when calling a function, arguments must be given in the following order: positional arguments first (`value`), then any combination of keyword arguments (`name=value`), variable positional arguments (`*name`), then variable keyword arguments (`**name`).

Since this can be a bit tricky when left hanging in the theoretical world, let's look at a couple of quick examples.

`arguments.all.py`

```
def func(a, b, c=7, *args, **kwargs):
    print('a, b, c:', a, b, c)
    print('args:', args)
    print('kwargs:', kwargs)

func(1, 2, 3, *(5, 7, 9), **{'A': 'a', 'B': 'b'})
func(1, 2, 3, 5, 7, 9, A='a', B='b')  # same as previous one
```

Note the order of the parameters in the function definition, and that the two calls are equivalent. In the first one, we're using the unpacking operators for iterables and dictionaries, while in the second one we're using a more explicit syntax. The execution of this yields (I printed only the result of one call):

```
$ python arguments.all.py
a, b, c: 1 2 3
args: (5, 7, 9)
kwargs: {'A': 'a', 'B': 'b'}
```

Let's now look at an example with keyword-only arguments.

arguments.all.kwonly.py

```
def func_with_kwonly(a, b=42, *args, c, d=256, **kwargs):
    print('a, b:', a, b)
    print('c, d:', c, d)
    print('args:', args)
    print('kwargs:', kwargs)

# both calls equivalent
func_with_kwonly(3, 42, c=0, d=1, *(7, 9, 11), e='E', f='F')
func_with_kwonly(3, 42, *(7, 9, 11), c=0, d=1, e='E', f='F')
```

Note that I have highlighted the keyword-only arguments in the function declaration. They come after the variable positional argument `*args`, and it would be the same if they came right after a single `*` (in which case there wouldn't be a variable positional argument). The execution of this yields (I printed only the result of one call):

```
$ python arguments.all.kwonly.py
a, b: 3 42
c, d: 0 1
args: (7, 9, 11)
kwargs: {'f': 'F', 'e': 'E'}
```

One other thing to note are the names I gave to the variable positional and keyword arguments. You're free to choose differently, but be aware that `args` and `kwargs` are the conventional names given to these parameters, at least generically. Now that you know how to define a function in all possible flavors, let me show you something tricky: mutable defaults.

Avoid the trap! Mutable defaults

One thing to be very aware of with Python is that default values are created at `def` time, therefore, subsequent calls to the same function will possibly behave differently according to the mutability of their default values. Let's look at an example:

`arguments.defaults.mutable.py`

```
def func(a=[], b={}):
    print(a)
    print(b)
    print('#' * 12)
    a.append(len(a))  # this will affect a's default value
    b[len(a)] = len(a)  # and this will affect b's one

func()
func()
func()
```

The parameters both have mutable default values. This means that, if you affect those objects, any modification will stick around in subsequent function calls. See if you can understand the output of those calls:

```
$ python arguments.defaults.mutable.py
[]
{}
############
[0]
{1: 1}
############
[0, 1]
{1: 1, 2: 2}
############
```

It's interesting, isn't it? While this behavior may seem very weird at first, it actually makes sense, and it's very handy, for example, when using memoization techniques (Google an example of that, if you're interested).

Even more interesting is what happens when, between the calls, we introduce one that doesn't use defaults, like this:

`arguments.defaults.mutable.intermediate.call.py`

```
func()
func(a=[1, 2, 3], b={'B': 1})
func()
```

When we run this code, this is the output:

```
$ python arguments.defaults.mutable.intermediate.call.py
[]
{}
############
[1, 2, 3]
{'B': 1}
############
[0]
{1: 1}
############
```

This output shows us that the defaults are retained even if we call the function with other values. One question that comes to mind is, how do I get a fresh empty value every time? Well, the convention is the following:

arguments.defaults.mutable.no.trap.py

```
def func(a=None):
    if a is None:
        a = []
    # do whatever you want with `a` ...
```

Note that, by using the preceding technique, if `a` isn't passed when calling the function, you always get a brand new empty list.

Okay, enough with the input, let's look at the other side of the coin, the output.

Return values

Return values of functions are one of those things where Python is light years ahead of most other languages. Functions are usually allowed to return one object (one value) but, in Python, you can return a tuple, and this implies that you can return whatever you want. This feature allows a coder to write software that would be much harder to write in any other language, or certainly more tedious. We've already said that to return something from a function we need to use the `return` statement, followed by what we want to return. There can be as many return statements as needed in the body of a function.

On the other hand, if within the body of a function we don't return anything, the function will return `None`. This behavior is harmless and, even though I don't have the room here to go into detail explaining why Python was designed like this, let me just tell you that this feature allows for several interesting patterns, and confirms Python as a very consistent language.

I say it's harmless because you are never forced to collect the result of a function call. I'll show you what I mean with an example:

`return.none.py`

```
def func():
    pass
func()  # the return of this call won't be collected. It's lost.
a = func()  # the return of this one instead is collected into `a`
print(a)  # prints: None
```

Note that the whole body of the function is comprised only of the `pass` statement. As the official documentation tells us, `pass` is a null operation. When it is executed, nothing happens. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed. In other languages, we would probably just indicate that with a pair of curly braces ({}), which define an *empty scope* but in Python a scope is defined by indenting code, therefore a statement such as `pass` is necessary.

Notice also that the first call of the function `func` returns a value (`None`) which we don't collect. As I said before, collecting the return value of a function call is not mandatory.

Now, that's good but not very interesting so, how about we write an interesting function? Remember that in *Chapter 1*, *Introduction and First Steps – Take a Deep Breath*, we talked about the factorial of a function. Let's write our own here (for simplicity, I will assume the function is always called correctly with appropriate values so I won't sanity-check on the input argument):

`return.single.value.py`

```
def factorial(n):
    if n in (0, 1):
        return 1
    result = n
    for k in range(2, n):
        result *= k
    return result

f5 = factorial(5)  # f5 = 120
```

Note that we have two points of return. If `n` is either `0` or `1` (in Python it's common to use the `in` type of check as I did instead of the more verbose `if n ==0 or n == 1:`), we return `1`. Otherwise, we perform the required calculation, and we return `result`. Can we write this function a little bit more Pythonically? Yes, but I'll let you figure out that for yourself, as an exercise.

`return.single.value.2.py`

```
from functools import reduce
from operator import mul

def factorial(n):
    return reduce(mul, range(1, n + 1), 1)
f5 = factorial(5)  # f5 = 120
```

I know what you're thinking, one line? Python is elegant, and concise! I think this function is readable even if you have never seen `reduce` or `mul`, but if you can't read it or understand it, set aside a few minutes and do some research on the Python documentation until its behavior is clear to you. Being able to look up functions in the documentation and understand code written by someone else is a task every developer needs to be able to perform, so think of this as a good exercise, and good luck!

To this end, make sure you look up the `help` function, which comes in very handy exploring with the console.

Returning multiple values

Unlike in most other languages, in Python it's very easy to return multiple objects from a function. This feature opens up a whole world of possibilities and allows you to code in a style that is hard to reproduce with other languages. Our thinking is limited by the tools we use, therefore when Python gives you more freedom than other languages, it is actually boosting your own creativity as well. To return multiple values is very easy, you just use tuples (either explicitly or implicitly). Let's look at a simple example that mimics the `divmod` built-in function:

`return.multiple.py`

```
def moddiv(a, b):
    return a // b, a % b

print(moddiv(20, 7))  # prints (2, 6)
```

I could have wrapped the highlighted part in the preceding code in braces, making it an explicit tuple, but there's no need for that. The preceding function returns both the result and the remainder of the division, at the same time.

A few useful tips

When writing functions, it's very useful to follow guidelines so that you write them well. I'll quickly point some of them out here:

- **Functions should do one thing**: Functions that do one thing are easy to describe in one short sentence. Functions which do multiple things can be split into smaller functions which do one thing. These smaller functions are usually easier to read and understand. Remember the data science example we saw a few pages ago.
- **Functions should be small**: The smaller they are, the easier it is to test them and to write them so that they do one thing.
- **The fewer input parameters, the better**: Functions which take a lot of arguments quickly become harder to manage (among other issues).
- **Functions should be consistent in their return values**: Returning `False` or `None` is not the same thing, even if within a Boolean context they both evaluate to `False`. `False` means that we have information (`False`), while `None` means that there is no information. Try writing functions which return in a consistent way, no matter what happens in their body.
- **Functions shouldn't have side effects**: In other words, functions should not affect the values you call them with. This is probably the hardest statement to understand at this point, so I'll give you an example using lists. In the following code, note how `numbers` is not sorted by the `sorted` function, which actually returns a sorted copy of `numbers`. Conversely, the `list.sort()` method is acting on the `numbers` object itself, and that is fine because it is a method (a function that belongs to an object and therefore has the rights to modify it):

```
>>> numbers = [4, 1, 7, 5]
>>> sorted(numbers)  # won't sort the original `numbers` list
[1, 4, 5, 7]
>>> numbers  # let's verify
[4, 1, 7, 5]  # good, untouched
>>> numbers.sort()  # this will act on the list
>>> numbers
[1, 4, 5, 7]
```

Follow these guidelines and you'll write better functions, which will serve you well.

Chapter 3, *Functions* in *Clean Code* by Robert C. Martin, Prentice Hall is dedicated to functions and it's probably the best set of guidelines I've ever read on the subject.

Recursive functions

When a function calls itself to produce a result, it is said to be **recursive**. Sometimes recursive functions are very useful in that they make it easier to write code. Some algorithms are very easy to write using the recursive paradigm, while others are not. There is no recursive function that cannot be rewritten in an iterative fashion, so it's usually up to the programmer to choose the best approach for the case at hand.

A recursive function usually has a set of base cases for which the return value doesn't depend on a subsequent call to the function itself and a set of recursive cases, for which the return value is calculated with one or more calls to the function itself.

As an example, we can consider the (hopefully familiar by now) `factorial` function *N!*. The base case is when *N* is either 0 or 1. The function returns 1 with no need for further calculation. On the other hand, in the general case, *N!* returns the product *1 * 2 * ... * (N-1) * N*. If you think about it, *N!* can be rewritten like this: *N! = (N-1)! * N*. As a practical example, consider *5! = 1 * 2 * 3 * 4 * 5 = (1 * 2 * 3 * 4) * 5 = 4! * 5*.

Let's write this down in code:

`recursive.factorial.py`

```
def factorial(n):
    if n in (0, 1):  # base case
        return 1
    return factorial(n - 1) * n  # recursive case
```

When writing recursive functions, always consider how many nested calls you make, there is a limit. For further information on this, check out `sys.getrecursionlimit()` and `sys.setrecursionlimit()`.

Recursive functions are used a lot when writing algorithms and they can be really fun to write. As a good exercise, try to solve a couple of simple problems using both a recursive and an iterative approach.

Anonymous functions

One last type of functions that I want to talk about are **anonymous** functions. These functions, which are called **lambdas** in Python, are usually used when a fully-fledged function with its own name would be overkill, and all we want is a quick, simple one-liner that does the job.

Imagine that you want a list of all the numbers up to *N* which are multiples of five. Imagine that you want to filter those out using the `filter` function, which takes a function and an iterable and constructs a filter object which you can iterate on, from those elements of iterable for which the function returns `True`. Without using an anonymous function, you would do something like this:

`filter.regular.py`

```
def is_multiple_of_five(n):
    return not n % 5
def get_multiples_of_five(n):
    return list(filter(is_multiple_of_five, range(n)))
print(get_multiples_of_five(50))
```

I have highlighted the main logic of `get_multiples_of_five`. Note how the filter uses `is_multiple_of_five` to filter the first *n* natural numbers. This seems a bit excessive, the task is simple and we don't need to keep the `is_multiple_of_five` function around for anything else. Let's rewrite it using a lambda function:

`filter.lambda.py`

```
def get_multiples_of_five(n):
    return list(filter(lambda k: not k % 5, range(n)))
print(get_multiples_of_five(50))
```

The logic is exactly the same but the filtering function is now a lambda. Defining a lambda is very easy and follows this form: `func_name = lambda [parameter_list]: expression`. A function object is returned, which is equivalent to this: `def func_name([parameter_list]): return expression`.

Note that optional parameters are indicated following the common syntax of wrapping them in square brackets.

Let's look at another couple of examples of equivalent functions defined in the two forms:

lambda.explained.py

```
# example 1: adder
def adder(a, b):
    return a + b
# is equivalent to:
adder_lambda = lambda a, b: a + b

# example 2: to uppercase
def to_upper(s):
    return s.upper()
# is equivalent to:
to_upper_lambda = lambda s: s.upper()
```

The preceding examples are very simple. The first one adds two numbers, and the second one produces the uppercase version of a string. Note that I assigned what is returned by the `lambda` expressions to a name (`adder_lambda`, `to_upper_lambda`), but there is no need for that when you use lambdas in the way we did in the `filter` example before.

Function attributes

Every function is a fully-fledged object and, as such, they have many attributes. Some of them are special and can be used in an introspective way to inspect the function object at runtime. The following script is an example that shows all of them and how to display their value for an example function:

func.attributes.py

```
def multiplication(a, b=1):
    """Return a multiplied by b. """
    return a * b

special_attributes = [
    "__doc__", "__name__", "__qualname__", "__module__",
    "__defaults__", "__code__", "__globals__", "__dict__",
    "__closure__", "__annotations__", "__kwdefaults__",
]

for attribute in special_attributes:
    print(attribute, '->', getattr(multiplication, attribute))
```

I used the built-in `getattr` function to get the value of those attributes. `getattr(obj, attribute)` is equivalent to `obj.attribute` and comes in handy when we need to get an attribute at runtime using its string name. Running this script yields:

```
$ python func.attributes.py
__doc__ -> Return a multiplied by b.
__name__ -> multiplication
__qualname__ -> multiplication
__module__ -> __main__
__defaults__ -> (1,)
__code__ -> <code object multiplication at 0x7ff529e79300, file "ch4/
func.attributes.py", line 1>
__globals__ -> {... omitted ...}
__dict__ -> {}
__closure__ -> None
__annotations__ -> {}
__kwdefaults__ -> None
```

I have omitted the value of the `__globals__` attribute, it was too big. An explanation of the meaning of this attribute can be found in the *types* section of the *Python Data Model* documentation page.

Built-in functions

Python comes with a lot of built-in functions. They are available anywhere and you can get a list of them by inspecting the `builtins` module with `dir(__builtins__)`, or by going to the official Python documentation. Unfortunately, I don't have the room to go through all of them here. Some of them we've already seen, such as `any`, `bin`, `bool`, `divmod`, `filter`, `float`, `getattr`, `id`, `int`, `len`, `list`, `min`, `print`, `set`, `tuple`, `type`, and `zip`, but there are many more, which you should read at least once.

Get familiar with them, experiment, write a small piece of code for each of them, make sure you have them at the tip of your fingers so that you can use them when you need them.

One final example

Before we finish off this chapter, how about a final example? I was thinking we could write a function to generate a list of prime numbers up to a limit. We've already seen the code for this so let's make it a function and, to keep it interesting, let's optimize it a bit.

It turns out that you don't need to divide it by all numbers from 2 to *N*-1 to decide if a number *N* is prime. You can stop at $\sqrt{N}$. Moreover, you don't need to test the division for all numbers from 2 to $\sqrt{N}$, you can just use the primes in that range. I'll leave it to you to figure out why this works, if you're interested. Let's see how the code changes:

primes.py

```
from math import sqrt, ceil

def get_primes(n):
    """Calculate a list of primes up to n (included). """
    primelist = []
    for candidate in range(2, n + 1):
        is_prime = True
        root = int(ceil(sqrt(candidate)))  # division limit
        for prime in primelist:  # we try only the primes
            if prime > root:  # no need to check any further
                break
            if candidate % prime == 0:
                is_prime = False
                break
        if is_prime:
            primelist.append(candidate)
    return primelist
```

The code is the same as in the previous chapter. We have changed the division algorithm so that we only test divisibility using the previously calculated primes and we stopped once the testing divisor was greater than the root of the candidate. We used the result list `primelist` to get the primes for the division. We calculated the root value using a fancy formula, the integer value of the ceiling of the root of the candidate. While a simple `int(k ** 0.5) + 1` would have served our purpose as well, the formula I chose is cleaner and requires me to use a couple of imports, which I wanted to show you. Check out the functions in the `math` module, they are very interesting!

Documenting your code

I'm a big fan of code that doesn't need documentation. When you program correctly, choose the right names and take care of the details, your code should come out as self-explanatory and documentation should not be needed. Sometimes a comment is very useful though, and so is some documentation. You can find the guidelines for documenting Python in *PEP257 – Docstring conventions*, but I'll show you the basics here.

Python is documented with strings, which are aptly called **docstrings**. Any object can be documented, and you can use either one-line or multi-line docstrings. One-liners are very simple. They should not provide another signature for the function, but clearly state its purpose.

`docstrings.py`

```
    def square(n):
        """Return the square of a number n. """
        return n ** 2

    def get_username(userid):
        """Return the username of a user given their id. """
        return db.get(user_id=userid).username
```

Using triple double-quoted strings allows you to expand easily later on. Use sentences that end in a period, and don't leave blank lines before or after.

Multi-line comments are structured in a similar way. There should be a one-liner that briefly gives you the gist of what the object is about, and then a more verbose description. As an example, I have documented a fictitious `connect` function, using the Sphinx notation, in the following example.

Sphinx is probably the most widely used tool for creating Python documentation. In fact, the official Python documentation was written with it. It's definitely worth spending some time checking it out.

`docstrings.py`

```
    def connect(host, port, user, password):
        """Connect to a database.

        Connect to a PostgreSQL database directly, using the given
        parameters.
```

```
        :param host: The host IP.
        :param port: The desired port.
        :param user: The connection username.
        :param password: The connection password.
        :return: The connection object.
        """
        # body of the function here...
        return connection
```

Importing objects

Now that you know a lot about functions, let's see how to use them. The whole point of writing functions is to be able to later reuse them, and this in Python translates to importing them into the namespace in which you need them. There are many different ways to import objects into a namespace, but the most common ones are just two: `import module_name` and `from module_name import function_name`. Of course, these are quite simplistic examples, but bear with me for the time being.

The form `import module_name` finds the module `module_name` and defines a name for it in the local namespace where the `import` statement is executed.

The form `from module_name import identifier` is a little bit more complicated than that, but basically does the same thing. It finds `module_name` and searches for an attribute (or a submodule) and stores a reference to `identifier` in the local namespace.

Both forms have the option to change the name of the imported object using the `as` clause, like this:

```
from mymodule import myfunc as better_named_func
```

Just to give you a flavor of what importing looks like, here's an example from a test module of a number theory library I wrote some years ago (it's available on Bitbucket):

karma/test_nt.py

```
import unittest  # imports the unittest module
from math import sqrt  # imports one function from math
from random import randint, sample  # two imports at once

from mock import patch
from nose.tools import (  # multiline import
    assert_equal,
    assert_list_equal,
```

```
        assert_not_in,
    )

    from karma import nt, utils
```

I commented some of them and I hope it's easy to follow. When you have a structure of files starting in the root of your project, you can use the dot notation to get to the object you want to import into your current namespace, be it a package, a module, a class, a function, or anything else. The `from module import` syntax also allows a catch-all clause `from module import *`, which is sometimes used to get all the names from a module into the current namespace at once, but it's frowned upon for several reasons: performances, the risk of silently shadowing other names, and so on. You can read all that there is to know about imports in the official Python documentation but, before we leave the subject, let me give you a better example.

Imagine that you have defined a couple of functions: `square(n)` and `cube(n)` in a module, `funcdef.py`, which is in the `lib` folder. You want to use them in a couple of modules which are at the same level of the `lib` folder, called `func_import.py`, and `func_from.py`. Showing the tree structure of that project produces something like this:

```
├── func_from.py
├── func_import.py
├── lib
    ├── funcdef.py
    └── __init__.py
```

Before I show you the code of each module, please remember that in order to tell Python that it is actually a package, we need to put a `__init__.py` module in it.

There are two things to note about the `__init__.py` file. First of all, it is a fully fledged Python module so you can put code into it as you would with any other module. Second, as of Python 3.3, its presence is no longer required to make a folder be interpreted as a Python package.

The code is as follows:

`funcdef.py`

```
    def square(n):
        return n ** 2
    def cube(n):
        return n ** 3
```

`func_import.py`

```
    import lib.funcdef
    print(lib.funcdef.square(10))
    print(lib.funcdef.cube(10))
```

`func_from.py`

```
    from lib.funcdef import square, cube
    print(square(10))
    print(cube(10))
```

Both these files, when executed, print `100` and `1000`. You can see how differently we then access the `square` and `cube` functions, according to how and what we imported in the current scope.

Relative imports

The imports we've seen until now are called absolute, that is to say they define the whole path of the module that we want to import, or from which we want to import an object. There is another way of importing objects into Python, which is called relative import. It's helpful in situations in which we want to rearrange the structure of large packages without having to edit sub-packages, or when we want to make a module inside a package able to import itself. Relative imports are done by adding as many leading dots in front of the module as the number of folders we need to backtrack, in order to find what we're searching for. Simply put, it is something like this:

```
    from .mymodule import myfunc
```

For a complete explanation of relative imports, refer to **PEP328** (`https://www.python.org/dev/peps/pep-0328`).

In later chapters, we'll create projects using different libraries and we'll use several different types of imports, including relative ones, so make sure you take a bit of time to read up about it in the official Python documentation.

Summary

In this chapter, finally we explored the world of functions. They are extremely important and, from now on, we'll use them basically everywhere. We talked about the main reasons for using them, the most important of which are code reuse and implementation hiding.

We saw that a function object is like a box that takes optional input and produces output. We can feed input values to a function in many different ways, using positional and keyword arguments, and using variable syntax for both types.

Now you should know how to write a function, how to document it, import it into your code, and call it.

The next chapter will force me to push my foot down on the throttle even more so I suggest you take any opportunity you get to consolidate and enrich the knowledge you've gathered until now by putting your nose into the Python official documentation.

Ready for the cool stuff? Let's go!

5
Saving Time and Memory

"It's not the daily increase but daily decrease. Hack away at the unessential."

-Bruce Lee

I love this quote from Bruce Lee, he was such a wise man! Especially, the second part, *hack away at the unessential*, is to me what makes a computer program elegant. After all, if there is a better way of doing things so that we don't waste time or memory, why not?

Sometimes, there are valid reasons for not pushing our code up to the maximum limit: for example, sometimes to achieve a negligible improvement, we have to sacrifice on readability or maintainability. Does it make any sense to have a web page served in 1 second with unreadable, complicated code, when we can serve it in 1.05 seconds with readable, clean code? No, it makes no sense.

On the other hand, sometimes it's perfectly licit to try and shave off a millisecond from a function, especially when the function is meant to be called thousands of times. Every millisecond you save there means one second saved per thousand of calls, and this could be meaningful for your application.

In light of these considerations, the focus of this chapter will not be to give you the tools to push your code to the absolute limits of performance and optimization "no matter what", but rather, to give you the tools to write efficient, elegant code that reads well, runs fast, and doesn't waste resources in an obvious way.

In this chapter, I will perform several measurements and comparisons, and cautiously draw some conclusions. Please do keep in mind that on a different box with a different setup or a different operating system, results may vary. Take a look at this code:

`squares.py`

```
def square1(n):
    return n ** 2  # squaring through the power operator

def square2(n):
    return n * n  # squaring through multiplication
```

Both functions return the square of *n*, but which is faster? From a simple benchmark I ran on them, it looks like the second is slightly faster. If you think about it, it makes sense: calculating the power of a number involves multiplication and therefore, whatever algorithm you may use to perform the power operation, it's not likely to beat a simple multiplication like the one in `square2`.

Do we care about this result? In most cases no. If you're coding an e-commerce website, chances are you won't ever even need to raise a number to the second power, and if you do, you probably will have to do it a few times per page. You don't need to concern yourself on saving a few microseconds on a function you call a few times.

So, when does optimization become important? One very common case is when you have to deal with huge collections of data. If you're applying the same function on a million `customer` objects, then you want your function to be tuned up to its best. Gaining 1/10 of a second on a function called one million times saves you 100,000 seconds, which are about 27.7 hours. That's not the same, right? So, let's focus on collections, and let's see which tools Python gives you to handle them with efficiency and grace.

Many of the concepts we will see in this chapter are based on those of **iterator** and **iterable**. Simply put, the ability for an object to return its next element when asked, and to raise a `StopIteration` exception when exhausted. We'll see how to code a custom iterator and iterable objects in the next chapter.

map, zip, and filter

We'll start by reviewing `map`, `filter`, and `zip`, which are the main built-in functions one can employ when handling collections, and then we'll learn how to achieve the same results using two very important constructs: **comprehensions** and **generators**. Fasten your seat belt!

map

According to the official Python documentation:

> `map(function, iterable, ...)` *returns an iterator that applies function to every item of iterable, yielding the results. If additional iterable arguments are passed, function must take that many arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted.*

We will explain the concept of yielding later on in the chapter. For now, let's translate this into code: we'll use a *lambda* function that takes a variable number of positional arguments, and just returns them as a tuple. Also, as `map` returns an iterator, we'll need to wrap each call to it within a `list` constructor so that we exhaust the iterable by putting all of its elements into a list (you'll see an example of this in the code):

`map.example.py`

```
>>> map(lambda *a: a, range(3))  # without wrapping in list...
<map object at 0x7f563513b518>  # we get the iterator object
>>> list(map(lambda *a: a, range(3)))  # wrapping in list...
[(0,), (1,), (2,)]  # we get a list with its elements
>>> list(map(lambda *a: a, range(3), 'abc'))  # 2 iterables
[(0, 'a'), (1, 'b'), (2, 'c')]
>>> list(map(lambda *a: a, range(3), 'abc', range(4, 7)))  # 3
[(0, 'a', 4), (1, 'b', 5), (2, 'c', 6)]
>>> # map stops at the shortest iterator
>>> list(map(lambda *a: a, (), 'abc'))  # empty tuple is shortest
[]
>>> list(map(lambda *a: a, (1, 2), 'abc'))  # (1, 2) shortest
[(1, 'a'), (2, 'b')]
>>> list(map(lambda *a: a, (1, 2, 3, 4), 'abc'))  # 'abc' shortest
[(1, 'a'), (2, 'b'), (3, 'c')]
```

In the preceding code you can see why, in order to present you with the results, I have to wrap the calls to `map` within a `list` constructor, otherwise I get the string representation of a `map` object, which is not really useful in this context, is it?

You can also notice how the elements of each iterable are applied to the function: at first, the first element of each iterable, then the second one of each iterable, and so on. Notice also that map stops when the shortest of the iterables we called it with is exhausted. This is actually a very nice behavior: it doesn't force us to level off all the iterables to a common length, and it doesn't break if they aren't all the same length.

`map` is very useful when you have to apply the same function to one or more collections of objects. As a more interesting example, let's see the **decorate-sort-undecorate** idiom (also known as **Schwartzian transform**). It's a technique that was extremely popular when Python sorting wasn't providing *key-functions*, and therefore today is less used, but it's a cool trick that still comes at hand once in a while.

Let's see a variation of it in the next example: we want to sort in descending order by the sum of credits accumulated by students, so to have the best student at position 0. We write a function to produce a decorated object, we sort, and then we undecorate. Each student has credits in three (possibly different) subjects. To decorate an object means to transform it, either adding extra data to it, or putting it into another object, in a way that allows us to be able to sort the original objects the way we want. After the sorting, we revert the decorated objects to get the original ones from them. This is called to undecorate.

`decorate.sort.undecorate.py`

```
students = [
    dict(id=0, credits=dict(math=9, physics=6, history=7)),
    dict(id=1, credits=dict(math=6, physics=7, latin=10)),
    dict(id=2, credits=dict(history=8, physics=9, chemistry=10)),
    dict(id=3, credits=dict(math=5, physics=5, geography=7)),
]

def decorate(student):
    # create a 2-tuple (sum of credits, student) from student dict
    return (sum(student['credits'].values()), student)

def undecorate(decorated_student):
    # discard sum of credits, return original student dict
    return decorated_student[1]

students = sorted(map(decorate, students), reverse=True)
students = list(map(undecorate, students))
```

In the preceding code, I highlighted the tricky and important parts. Let's start by understanding what each student object is. In fact, let's print the first one:

```
{'credits': {'history': 7, 'math': 9, 'physics': 6}, 'id': 0}
```

You can see that it's a dictionary with two keys: `id` and `credit`. The value of `credit` is also a dictionary in which there are three subject/grade key/value pairs. As I'm sure you recall from our visit in the data structures world, calling `dict.values()` returns an object similar to an `iterable`, with only the values. Therefore, `sum(student['credits'].values())`, for the first student is equivalent to `sum(9, 6, 7)` (or any permutation of those numbers because dictionaries don't retain order, but luckily for us, addition is commutative).

With that out of the way, it's easy to see what is the result of calling decorate with any of the students. Let's print the result of `decorate(students[0])`:

```
(22, {'credits': {'history': 7, 'math': 9, 'physics': 6}, 'id': 0})
```

That's nice! If we decorate all the students like this, we can sort them on their total amount of credits but just sorting the list of tuples. In order to apply the decoration to each item in students, we call `map(decorate, students)`. Then we sort the result, and then we undecorate in a similar fashion. If you have gone through the previous chapters correctly, understanding this code shouldn't be too hard.

Printing students after running the whole code yields:

```
$ python decorate.sort.undecorate.py
[{'credits': {'chemistry': 10, 'history': 8, 'physics': 9}, 'id': 2},
 {'credits': {'latin': 10, 'math': 6, 'physics': 7}, 'id': 1},
 {'credits': {'history': 7, 'math': 9, 'physics': 6}, 'id': 0},
 {'credits': {'geography': 7, 'math': 5, 'physics': 5}, 'id': 3}]
```

And you can see, by the order of the student objects, that they have indeed been sorted by the sum of their credits.

For more on the *decorate-sort-undecorate* idiom, there's a very nice introduction in the sorting how-to section of the official Python documentation (`https://docs.python.org/3.4/howto/sorting.html#the-old-way-using-decorate-sort-undecorate`).

One thing to notice about the sorting part: what if two or more students share the same total sum? The sorting algorithm would then proceed sorting the tuples by comparing the `student` objects with each other. This doesn't make any sense, and in more complex cases could lead to unpredictable results, or even errors. If you want to be sure to avoid this issue, one simple solution is to create a 3-tuple instead of a 2-tuple, having the sum of credits in the first position, the position of the `student` object in the `students` list in the second one, and the `student` object itself in the third one. This way, if the sum of credits is the same, the tuples will be sorted against the position, which will always be different and therefore enough to resolve the sorting between any pair of tuples. For more considerations on this topic, please check out the sorting how-to section on the official Python documentation.

zip

We've already covered `zip` in the previous chapters, so let's just define it properly and then I want to show you how you could combine it with `map`.

According to the Python documentation:

> `zip(*iterables)` *returns an iterator of tuples, where the i-th tuple contains the i-th element from each of the argument sequences or iterables. The iterator stops when the shortest input iterable is exhausted. With a single iterable argument, it returns an iterator of 1-tuples. With no arguments, it returns an empty iterator.*

Let's see an example:

`zip.grades.py`

```
>>> grades = [18, 23, 30, 27, 15, 9, 22]
>>> avgs = [22, 21, 29, 24, 18, 18, 24]
>>> list(zip(avgs, grades))
[(22, 18), (21, 23), (29, 30), (24, 27), (18, 15), (18, 9), (24, 22)]
>>> list(map(lambda *a: a, avgs, grades))  # equivalent to zip
[(22, 18), (21, 23), (29, 30), (24, 27), (18, 15), (18, 9), (24, 22)]
```

In the preceding code, we're zipping together the average and the grade for the last exam, per each student. Notice how the code inside the two list calls produces exactly the same result, showing how easy it is to reproduce `zip` using `map`. Notice also that, as we do for `map`, we have to feed the result of the `zip` call to a `list` constructor.

A simple example on the combined use of `map` and `zip` could be a way of calculating the element-wise maximum amongst sequences, that is, the maximum of the first element of each sequence, then the maximum of the second one, and so on:

`maxims.py`

```
>>> a = [5, 9, 2, 4, 7]
>>> b = [3, 7, 1, 9, 2]
>>> c = [6, 8, 0, 5, 3]
>>> maxs = map(lambda n: max(*n), zip(a, b, c))
>>> list(maxs)
[6, 9, 2, 9, 7]
```

Notice how easy it is to calculate the max values of three sequences. `zip` is not strictly needed of course, we could just use `map`, but this would require us to write a much more complicated function to feed `map` with. Sometimes we may be in a situation where changing the function we feed to `map` is not even possible. In cases like these, being able to massage the data (like we're doing in this example with `zip`) is very helpful.

filter

According to the Python documentation:

> `filter(function, iterable)` *construct an iterator from those elements of iterable for which function returns True. iterable may be either a sequence, a container which supports iteration, or an iterator. If function is* `None`*, the identity function is assumed, that is, all elements of iterable that are false are removed.*

Let's see a very quick example:

`filter.py`

```
>>> test = [2, 5, 8, 0, 0, 1, 0]
>>> list(filter(None, test))
[2, 5, 8, 1]
>>> list(filter(lambda x: x, test))  # equivalent to previous one
[2, 5, 8, 1]
>>> list(filter(lambda x: x > 4, test))  # keep only items > 4
[5, 8]
```

In the preceding code, notice how the second call to filter is equivalent to the first one. If we pass a function that takes one argument and returns the argument itself, only those arguments that are `True` will make the function return `True`, therefore this behavior is exactly the same as passing `None`. It's often a very good exercise to mimic some of the built-in Python behaviors. When you succeed you can say you fully understand how Python behaves in a specific situation.

Armed with `map`, `zip`, and `filter` (and several other functions from the Python standard library) we can massage sequences very effectively. But those functions are not the only way to do it. So let's see one of the nicest features of Python: comprehensions.

Comprehensions

Python offers you different types of comprehensions: `list`, `dict`, and `set`.

We'll concentrate on the first one for now, and then it will be easy to explain the other two.

A `list` comprehension is a quick way of making a list. Usually the list is the result of some operation that may involve applying a function, filtering, or building a different data structure.

Let's start with a very simple example I want to calculate a list with the squares of the first 10 natural numbers. How would you do it? There are a couple of equivalent ways:

`squares.map.py`

```
# If you code like this you are not a Python guy! ;)
>>> squares = []
>>> for n in range(10):
...     squares.append(n ** 2)
...
>>> list(squares)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]


# This is better, one line, nice and readable
>>> squares = map(lambda n: n**2, range(10))
>>> list(squares)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

The preceding example should be nothing new for you. Let's see how to achieve the same result using a list comprehension:

squares.comprehension.py

```
>>> [n ** 2 for n in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

As simple as that. Isn't it elegant? Basically we have put a `for` loop within square brackets. Let's now filter out the odd squares. I'll show you how to do it with `map` and `filter`, and then using a `list` comprehension again.

even.squares.py

```
# using map and filter
sq1 = list(
    filter(lambda n: not n % 2, map(lambda n: n ** 2, range(10)))
)
# equivalent, but using list comprehensions
sq2 = [n ** 2 for n in range(10) if not n % 2]

print(sq1, sq1 == sq2)  # prints: [0, 4, 16, 36, 64] True
```

I think that now the difference in readability is evident. The list comprehension reads much better. It's almost English: give me all squares (*n* ** 2) for *n* between 0 and 9 if n is even.

According to the Python documentation:

> *A list comprehension consists of brackets containing an expression followed by a* `for` *clause, then zero or more* `for` *or* `if` *clauses. The result will be a new list resulting from evaluating the expression in the context of the* `for` *and* `if` *clauses which follow it*".

Nested comprehensions

Let's see an example of nested loops. It's very common when dealing with algorithms to have to iterate on a sequence using two placeholders. The first one runs through the whole sequence, left to right. The second one as well, but it starts from the first one, instead of 0. The concept is that of testing all pairs without duplication. Let's see the classical `for` loop equivalent.

pairs.for.loop.py

```
items = 'ABCDE'
pairs = []
for a in range(len(items)):
```

```
        for b in range(a, len(items)):
            pairs.append((items[a], items[b]))
```

If you print pairs at the end, you get:

```
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('A', 'D'), ('A', 'E'), ('B', 'B'),
('B', 'C'), ('B', 'D'), ('B', 'E'), ('C', 'C'), ('C', 'D'), ('C', 'E'),
('D', 'D'), ('D', 'E'), ('E', 'E')]
```

All the tuples with the same letter are those for which `b` is at the same position as `a`. Now, let's see how we can translate this in a list comprehension:

`pairs.list.comprehension.py`

```
items = 'ABCDE'
pairs = [(items[a], items[b])
    for a in range(len(items)) for b in range(a, len(items))]
```

This version is just two lines long and achieves the same result. Notice that in this particular case, because the `for` loop over `b` has a dependency on `a`, it must follow the `for` loop over `a` in the comprehension. If you swap them around, you'll get a name error.

Filtering a comprehension

We can apply filtering to a comprehension. Let's first do it with `filter`. Let's find all Pythagorean triples whose short sides are numbers smaller than 10. We obviously don't want to test a combination twice, and therefore we'll use a trick like the one we saw in the previous example.

A **Pythagorean triple** is a triple (*a*, *b*, *c*) of integer numbers satisfying the equation $a^2 + b^2 = c^2$.

`pythagorean.triple.py`

```
from math import sqrt
# this will generate all possible pairs
mx = 10
legs = [(a, b, sqrt(a**2 + b**2))
    for a in range(1, mx) for b in range(a, mx)]
# this will filter out all non pythagorean triples
legs = list(
    filter(lambda triple: triple[2].is_integer(), legs))
print(legs)  # prints: [(3, 4, 5.0), (6, 8, 10.0)]
```

In the preceding code, we generated a list of *3-tuples*, legs. Each tuple contains two integer numbers (the legs) and the hypotenuse of the Pythagorean triangle whose legs are the first two numbers in the tuple. For example, when a = 3 and b = 4, the tuple will be (3, 4, 5.0), and when a = 5 and b = 7, the tuple will be (5, 7, 8.602325267042627).

After having all the triples done, we need to filter out all those that don't have a hypotenuse that is an integer number. In order to do this, we filter based on `float_number.is_integer()` being `True`. This means that of the two example tuples I showed you before, the one with hypotenuse 5.0 will be retained, while the one with hypotenuse 8.602325267042627 will be discarded.

This is good, but I don't like that the triple has two integer numbers and a float. They are supposed to be all integers, so let's use map to fix this:

`pythagorean.triple.int.py`

```
from math import sqrt
mx = 10
legs = [(a, b, sqrt(a**2 + b**2))
    for a in range(1, mx) for b in range(a, mx)]
legs = filter(lambda triple: triple[2].is_integer(), legs)
# this will make the third number in the tuples integer
legs = list(
    map(lambda triple: triple[:2] + (int(triple[2]), ), legs))
print(legs)  # prints: [(3, 4, 5), (6, 8, 10)]
```

Notice the step we added. We take each element in legs and we slice it, taking only the first two elements in it. Then, we concatenate the slice with a 1-tuple, in which we put the integer version of that float number that we didn't like.

Seems like a lot of work, right? Indeed it is. Let's see how to do all this with a list comprehension:

`pythagorean.triple.comprehension.py`

```
from math import sqrt
# this step is the same as before
mx = 10
legs = [(a, b, sqrt(a**2 + b**2))
    for a in range(1, mx) for b in range(a, mx)]
# here we combine filter and map in one CLEAN list comprehension
legs = [(a, b, int(c)) for a, b, c in legs if c.is_integer()]
print(legs)  # prints: [(3, 4, 5), (6, 8, 10)]
```

I know. It's much better, isn't it? It's clean, readable, shorter. In other words, elegant.

I'm going quite fast here, as anticipated in the summary of the last chapter. Are you playing with this code? If not, I suggest you do. It's very important that you play around, break things, change things, see what happens. Make sure you have a clear understanding of what is going on. You want to become a ninja, right?

dict comprehensions

Dictionary and set comprehensions work exactly like the list ones, only there is a little difference in the syntax. The following example will suffice to explain everything you need to know:

`dictionary.comprehensions.py`

```
from string import ascii_lowercase
lettermap = dict((c, k) for k, c in enumerate(ascii_lowercase, 1))
```

If you print `lettermap`, you will see the following (I omitted the middle results, you get the gist):

```
{'a': 1,
 'b': 2,
 'c': 3,
 ... omitted results ...
 'x': 24,
 'y': 25,
 'z': 26}
```

What happens in the preceding code is that we're feeding the `dict` constructor with a comprehension (technically, a generator expression, we'll see it in a bit). We tell the `dict` constructor to make *key/value* pairs from each tuple in the comprehension. We enumerate the sequence of all lowercase ASCII letters, starting from *1*, using `enumerate`. Piece of cake. There is also another way to do the same thing, which is closer to the other dictionary syntax:

```
lettermap = {c: k for k, c in enumerate(ascii_lowercase, 1)}
```

It does exactly the same thing, with a slightly different syntax that highlights a bit more of the *key: value* part.

Dictionaries do not allow duplication in the keys, as shown in the following example:

dictionary.comprehensions.duplicates.py

```
word = 'Hello'
swaps = {c: c.swapcase() for c in word}
print(swaps)  # prints: {'o': 'O', 'l': 'L', 'e': 'E', 'H': 'h'}
```

We create a dictionary with keys, the letters in the string `'Hello'`, and values of the same letters, but with the case swapped. Notice there is only one `'l': 'L'` pair. The constructor doesn't complain, simply reassigns duplicates to the latest value. Let's make this clearer with another example; let's assign to each key its position in the string:

dictionary.comprehensions.positions.py

```
word = 'Hello'
positions = {c: k for k, c in enumerate(word)}
print(positions)  # prints: {'l': 3, 'o': 4, 'e': 1, 'H': 0}
```

Notice the value associated to the letter `'l': 3`. The pair `'l': 2` isn't there, it has been overridden by `'l': 3`.

set comprehensions

Set comprehensions are very similar to list and dictionary ones. Python allows both the `set()` constructor to be used, or the explicit `{}` syntax. Let's see one quick example:

set.comprehensions.py

```
word = 'Hello'
letters1 = set(c for c in word)
letters2 = {c for c in word}
print(letters1)  # prints: {'l', 'o', 'H', 'e'}
print(letters1 == letters2)  # prints: True
```

Notice how for set comprehensions, as for dictionaries, duplication is not allowed and therefore the resulting set has only four letters. Also, notice that the expressions assigned to `letters1` and `letters2` produce equivalent sets.

The syntax used to create `letters2` is very similar to the one we can use to create a dictionary comprehension. You can spot the difference only by the fact that dictionaries require keys and values, separated by columns, while sets don't.

Generators

Generators are one very powerful tool that Python gifts us with. They are based on the concepts of *iteration*, as we said before, and they allow for coding patterns that combine elegance with efficiency.

Generators are of two types:

- **Generator functions**: These are very similar to regular functions, but instead of returning results through return statements, they use yield, which allows them to suspend and resume their state between each call
- **Generator expressions**: These are very similar to the list comprehensions we've seen in this chapter, but instead of returning a list they return an object that produces results one by one

Generator functions

Generator functions come under all aspects like regular functions, with one difference: instead of collecting results and returning them at once, they can start the computation, yield one value, suspend their state saving everything they need to be able to resume and, if called again, resume and perform another step. Generator functions are automatically turned into their own iterators by Python, so you can call `next` on them.

This is all very theoretical so, let's make it clear why such a mechanism is so powerful, and then let's see an example.

Say I asked you to count out loud from 1 to a million. You start, and at some point I ask you to stop. After some time, I ask you to resume. At this point, what is the minimum information you need to be able to resume correctly? Well, you need to remember the last number you called. If I stopped you after 31415, you will just go on with 31416, and so on.

The point is, you don't need to remember all the numbers you said before 31415, nor do you need them to be written down somewhere. Well, you may not know it, but you're behaving like a generator already!

Take a good look at the following code:

`first.n.squares.py`

```
def get_squares(n):  # classic function approach
    return [x ** 2 for x in range(n)]
print(get_squares(10))
```

```
def get_squares_gen(n):  # generator approach
    for x in range(n):
        yield x ** 2  # we yield, we don't return
print(list(get_squares_gen(10)))
```

The result of the prints will be the same: `[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]`. But there is a huge difference between the two functions. `get_squares` is a classic function that collects all the squares of numbers in [0, *n*) in a list, and returns it. On the other hand, `get_squares_gen` is a generator, and behaves very differently. Each time the interpreter reaches the `yield` line, its execution is suspended. The only reason those prints return the same result is because we fed `get_squares_gen` to the `list` constructor, which when called like that exhausts the generator completely by asking the next element until a `StopIteration` is raised. Let's see this in detail:

`first.n.squares.manual.py`

```
def get_squares_gen(n):
    for x in range(n):
        yield x ** 2

squares = get_squares_gen(4)  # this creates a generator object
print(squares)  # <generator object get_squares_gen at 0x7f158...>
print(next(squares))  # prints: 0
print(next(squares))  # prints: 1
print(next(squares))  # prints: 4
print(next(squares))  # prints: 9
# the following raises StopIteration, the generator is exhausted,
# any further call to next will keep raising StopIteration
print(next(squares))
```

In the preceding code, each time we call `next` on the generator object, we either start it (first `next`) or make it resume from the last suspension point (any other `next`).

The first time we call `next` on it, we get `0`, which is the square of `0`, then `1`, then `4`, then `9` and since the `for` loop stops after that (`n` is `4`), then the generator naturally ends. A classic function would at that point just return `None`, but in order to comply with the iteration protocol, a generator will instead raise a `StopIteration` exception.

This explains how a `for` loop works for example. When you call `for k in range(n)`, what happens under the hood is that the `for` loop gets an iterator out of `range(n)` and starts calling `next` on it, until `StopIteration` is raised, which tells the `for` loop that the iteration has reached its end.

Having this behavior built-in in every iteration aspect of Python makes generators even more powerful because once we write them, we'll be able to plug them in whatever iteration mechanism we want.

At this point, you're probably asking yourself why would you want to use a generator instead of a regular function. Well, the title of this chapter should suggest the answer. I'll talk about performances later, so for now let's concentrate on another aspect: sometimes generators allow you to do something that wouldn't be possible with a simple list. For example, say you want to analyze all permutations of a sequence. If the sequence has length *N*, then the number of its permutations is *N!*. This means that if the sequence is 10 elements long, the number of permutations is 3628800. But a sequence of 20 elements would have 2432902008176640000 permutations. They grow factorially.

Now imagine you have a classic function that is attempting to calculate all permutations, put them in a list, and return it to you. With 10 elements, it would require probably a few tens of seconds, but for 20 elements there is simply no way that it can be done.

On the other hand, a generator function will be able to start the computation and give you back the first permutation, then the second, and so on. Of course you won't have the time to parse them all, they are too many, but at least you'll be able to work with some of them.

Remember when we were talking about the `break` statement in `for` loops? When we found a number dividing a *candidate prime* we were breaking the loop, no need to go on.

Sometimes it's exactly the same, only the amount of data you have to iterate over is so huge that you cannot keep it all in memory in a list. In this case, generators are invaluable: they make possible what wouldn't be possible otherwise.

So, in order to save memory (and time), use generator functions whenever possible.

It's also worth noting that you can use the return statement in a generator function. It will produce a `StopIteration` exception to be raised, effectively ending the iteration. This is extremely important. If a `return` statement were actually to make the function return something, it would break the iteration protocol. Python consistency prevents this, and allows us great ease when coding. Let's see a quick example:

`gen.yield.return.py`

```
def geometric_progression(a, q):
    k = 0
    while True:
        result = a * q**k
```

```
            if result <= 100000:
                yield result
            else:
                return
            k += 1

    for n in geometric_progression(2, 5):
        print(n)
```

The preceding code yields all terms of the geometric progression a, aq, aq^2, aq^3, ... When the progression produces a term that is greater than 100,000, the generator stops (with a `return` statement). Running the code produces the following result:

```
$ python gen.yield.return.py
2
10
50
250
1250
6250
31250
```

The next term would have been 156250, which is too big.

Going beyond next

At the beginning of this chapter, I told you that generator objects are based on the iteration protocol. We'll see in the next chapter a complete example of how to write a custom iterator/iterable object. For now, I just want you to understand how `next()` works.

What happens when you call `next(generator)` is that you're calling the `generator.__next__()` method. Remember, a **method** is just a function that belongs to an object, and objects in Python can have special methods. Our friend `__next__()` is just one of these and its purpose is to return the next element of the iteration, or to raise `StopIteration` when the iteration is over and there are no more elements to return.

In Python, an object's special methods are also called **magic methods**, or **dunder** (from "double underscore") **methods**.

When we write a generator function, Python automatically transforms it into an object that is very similar to an iterator, and when we call `next(generator)`, that call is transformed in `generator.__next__()`. Let's revisit the previous example about generating squares:

`first.n.squares.manual.method.py`

```
def get_squares_gen(n):
    for x in range(n):
        yield x ** 2

squares = get_squares_gen(3)
print(squares.__next__())  # prints: 0
print(squares.__next__())  # prints: 1
print(squares.__next__())  # prints: 4
# the following raises StopIteration, the generator is exhausted,
# any further call to next will keep raising StopIteration
print(squares.__next__())
```

The result is exactly as the previous example, only this time instead of using the proxy call `next(squares)`, we're directly calling `squares.__next__()`.

Generator objects have also three other methods that allow controlling their behavior: `send`, `throw`, and `close`. `send` allows us to communicate a value back to the generator object, while `throw` and `close` respectively allow raising an exception within the generator and closing it. Their use is quite advanced and I won't be covering them here in detail, but I want to spend a few words at least about `send`, with a simple example.

Take a look at the following code:

`gen.send.preparation.py`

```
def counter(start=0):
    n = start
    while True:
        yield n
        n += 1

c = counter()
print(next(c))  # prints: 0
print(next(c))  # prints: 1
print(next(c))  # prints: 2
```

The preceding iterator creates a generator object that will run forever. You can keep calling it, it will never stop. Alternatively, you can put it in a `for` loop, for example, `for n in counter(): ...` and it will go on forever as well.

Now, what if you wanted to stop it at some point? One solution is to use a variable to control the `while` loop. Something like this:

`gen.send.preparation.stop.py`

```
stop = False
def counter(start=0):
    n = start
    while not stop:
        yield n
        n += 1

c = counter()
print(next(c))  # prints: 0
print(next(c))  # prints: 1
stop = True
print(next(c))  # raises StopIteration
```

This will do it. We start with `stop = False`, and until we change it to `True`, the generator will just keep going, like before. The moment we change stop to `True` though, the `while` loop will exit, and the next call will raise a `StopIteration` exception. This trick works, but I don't like it. We depend on an external variable, and this can lead to issues: what if another function changes that `stop`? Moreover, the code is scattered. In a nutshell, this isn't good enough.

We can make it better by using `generator.send()`. When we call `generator.send()`, the value that we feed to `send` will be passed in to the generator, execution is resumed, and we can fetch it via the `yield` expression. This is all very complicated when explained with words, so let's see an example:

`gen.send.py`

```
def counter(start=0):
    n = start
    while True:
        result = yield n              # A
        print(type(result), result)  # B
        if result == 'Q':
            break
        n += 1
```

```
    c = counter()
    print(next(c))          # C
    print(c.send('Wow!'))   # D
    print(next(c))          # E
    print(c.send('Q'))      # F
```

Execution of the preceding code produces the following:

```
$ python gen.send.py
0
<class 'str'> Wow!
1
<class 'NoneType'> None
2
<class 'str'> Q
Traceback (most recent call last):
  File "gen.send.py", line 14, in <module>
    print(c.send('Q'))      # F
StopIteration
```

I think it's worth going through this code line by line, like if we were executing it, and see if we can understand what's going on.

We start the generator execution with a call to `next` (`#C`). Within the generator, `n` is set to the same value of `start`. The `while` loop is entered, execution stops (`#A`) and `n` (0) is yielded back to the caller. 0 is printed on the console.

We then call `send` (`#D`), execution resumes and `result` is set to `'Wow!'` (still `#A`), then its type and value are printed on the console (`#B`). `result` is not `'Q'`, therefore `n` is incremented by 1 and execution goes back to the `while` condition, which, being `True`, evaluates to `True` (that wasn't hard to guess, right?). Another loop cycle begins, execution stops again (`#A`), and `n` (1) is yielded back to the caller. 1 is printed on the console.

At this point, we call `next` (`#E`), execution is resumed again (`#A`), and because we are not sending anything to the generator explicitly, Python behaves exactly like functions that are not using the `return` statement: the `yield n` expression (`#A`) returns `None`. `result` therefore is set to `None`, and its type and value are yet again printed on the console (`#B`). Execution continues, `result` is not `'Q'` so `n` is incremented by 1, and we start another loop again. Execution stops again (`#A`) and `n` (2) is yielded back to the caller. 2 is printed on the console.

And now for the grand finale: we call `send` again (`#F`), but this time we pass in `'Q'`, therefore when execution is resumed, `result` is set to `'Q'` (`#A`). Its type and value are printed on the console (`#B`), and then finally the `if` clause evaluates to `True` and the `while` loop is stopped by the `break` statement. The generator naturally terminates and this means a `StopIteration` exception is raised. You can see the print of its traceback on the last few lines printed on the console.

This is not at all simple to understand at first, so if it's not clear to you, don't be discouraged. You can keep reading on and then you can come back to this example after some time.

Using `send` allows for interesting patterns, and it's worth noting that `send` can only be used to resume the execution, not to start it. Only `next` starts the execution of a generator.

The yield from expression

Another interesting construct is the `yield from` expression. This expression allows you to yield values from a subiterator. Its use allows for quite advanced patterns, so let's just see a very quick example of it:

`gen.yield.for.py`

```
def print_squares(start, end):
    for n in range(start, end):
        yield n ** 2

for n in print_squares(2, 5):
    print(n)
```

The previous code prints the numbers `4`, `9`, `16` on the console (on separate lines). By now, I expect you to be able to understand it by yourself, but let's quickly recap what happens. The `for` loop outside the function gets an iterator from `print_squares(2, 5)` and calls `next` on it until iteration is over. Every time the generator is called, execution is suspended (and later resumed) on `yield n ** 2`, which returns the square of the current `n`.

Let's see how we can transform this code benefiting from the `yield from` expression:

`gen.yield.from.py`

```
def print_squares(start, end):
    yield from (n ** 2 for n in range(start, end))

for n in print_squares(2, 5):
    print(n)
```

This code produces the same result, but as you can see the `yield from` is actually running a subiterator `(n ** 2 …)`. The `yield from` expression returns to the caller each value the subiterator is producing. It's shorter and it reads better.

Generator expressions

Let's now talk about the other techniques to generate values one at a time.

The syntax is exactly the same as list comprehensions, only, instead of wrapping the comprehension with square brackets, you wrap it with round braces. That is called a **generator expression**.

In general, generator expressions behave like equivalent list comprehensions, but there is one very important thing to remember: generators allow for one iteration only, then they will be exhausted. Let's see an example:

`generator.expressions.py`

```
>>> cubes = [k**3 for k in range(10)]  # regular list
>>> cubes
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
>>> type(cubes)
<class 'list'>
>>> cubes_gen = (k**3 for k in range(10))  # create as generator
>>> cubes_gen
<generator object <genexpr> at 0x7ff26b5db990>
>>> type(cubes_gen)
<class 'generator'>
>>> list(cubes_gen)  # this will exhaust the generator
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
>>> list(cubes_gen)  # nothing more to give
[]
```

Look at the line in which the generator expression is created and assigned the name `cubes_gen`. You can see it's a generator object. In order to see its elements, we can use a `for` loop, a manual set of calls to `next`, or simply, feed it to a `list` constructor, which is what I did.

Notice how, once the generator has been exhausted, there is no way to recover the same elements from it again. We need to recreate it, if we want to use it from scratch again.

In the next few examples, let's see how to reproduce `map` and `filter` using generator expressions:

`gen.map.py`

```
def adder(*n):
    return sum(n)
s1 = sum(map(lambda n: adder(*n), zip(range(100), range(1, 101))))
s2 = sum(adder(*n) for n in zip(range(100), range(1, 101)))
```

In the previous example, `s1` and `s2` are exactly the same: they are the sum of `adder(0, 1), adder(1, 2), adder(2, 3),` and so on, which translates to `sum(1, 3, 5, ...)`. The syntax is different though, I find the generator expression to be much more readable:

`gen.filter.py`

```
cubes = [x**3 for x in range(10)]
odd_cubes1 = filter(lambda cube: cube % 2, cubes)
odd_cubes2 = (cube for cube in cubes if cube % 2)
```

In the previous example, `odd_cubes1` and `odd_cubes2` are the same: they generate a sequence of odd cubes. Yet again, I prefer the generator syntax. This should be evident when things get a little more complicated:

`gen.map.filter.py`

```
N = 20
cubes1 = map(
    lambda n: (n, n**3),
    filter(lambda n: n % 3 == 0 or n % 5 == 0, range(N))
)
cubes2 = (
    (n, n**3) for n in range(N) if n % 3 == 0 or n % 5 == 0)
```

The preceding code creates to generators `cubes1` and `cubes2`. They are exactly the same, and return 2-tuples (n, n^3) when n is a multiple of 3 or 5.

If you print the list (`cubes1`), you get: `[(0, 0), (3, 27), (5, 125), (6, 216), (9, 729), (10, 1000), (12, 1728), (15, 3375), (18, 5832)]`.

See how much better the generator expression reads? It may be debatable when things are very simple, but as soon as you start nesting functions a bit, like we did in this example, the superiority of the generator syntax is evident. Shorter, simpler, more elegant.

Now, let me ask you a question: what is the difference between the following lines of code?

`sum.example.py`

```
s1 = sum([n**2 for n in range(10**6)])
s2 = sum((n**2 for n in range(10**6)))
s3 = sum(n**2 for n in range(10**6))
```

Strictly speaking, they all produce the same sum. The expressions to get `s2` and `s3` are exactly the same because the braces in `s2` are redundant. They are both generator expressions inside the `sum` function. The expression to get `s1` is different though. Inside `sum`, we find a list comprehension. This means that in order to calculate `s1`, the `sum` function has to call `next` on a list, a million times.

Do you see where we're loosing time and memory? Before `sum` can start calling `next` on that list, the list needs to have been created, which is a waste of time and space. It's much better for `sum` to call `next` on a simple generator expression. There is no need to have all the numbers from `range(10**6)` stored in a list.

So, *watch out for extra parentheses when you write your expressions*: sometimes it's easy to skip on these details, which makes our code much different. Don't believe me?

`sum.example.2.py`

```
s = sum([n**2 for n in range(10**8)])  # this is killed
# s = sum(n**2 for n in range(10**8))  # this succeeds
print(s)
```

Try running the preceding example. If I run the first line, this is what I get:

```
$ python sum.example.2.py
Killed
```

On the other hand, if I comment out the first line, and uncomment the second one, this is the result:

```
$ python sum.example.2.py
333333328333333350000000
```

Sweet generator expressions. The difference between the two lines is that in the first one, a list with the squares of the first hundred million numbers must be made before being able to sum them up. That list is huge, and we run out of memory (at least, my box did, if yours doesn't try a bigger number), therefore Python kills the process for us. Sad face.

But when we remove the square brackets, we don't make a list any more. The sum function receives 0, 1, 4, 9, and so on until the last one, and sums them up. No problems, happy face.

Some performance considerations

So, we've seen that we have many different ways to achieve the same result. We can use any combination of `map`, `zip`, `filter`, or choose to go with a comprehension, or maybe choose to use a generator, either function or expression. We may even decide to go with `for` loops: when the logic to apply to each running parameter isn't simple, they may be the best option.

Other than readability concerns though, let's talk about performances. When it comes to performances, usually there are two factors which play a major role: **space** and **time**.

Space means the size of the memory that a data structure is going to take up. The best way to choose is to ask yourself if you really need a list (or tuple) or if a simple generator function would work as well. If the answer is yes, go with the generator, it'll save a lot of space. Same goes with functions: if you don't actually need them to return a list or tuple, then you can transform them in generator functions as well.

Sometimes, you will have to use lists (or tuples), for example there are algorithms that scan sequences using multiple pointers or maybe they run over the sequence more than once. A generator function (or expression) can be iterated over only once and then it's exhausted, so in these situations, it wouldn't be the right choice.

Time is a bit harder than space because it depends on more variables and therefore it isn't possible to state that *X is faster than Y* with absolute certainty for all cases. However, based on tests run on Python today, we can say that `map` calls can be twice as fast as equivalent `for` loops, and list comprehensions can be (always generally speaking) even faster than equivalent `map` calls.

In order to fully appreciate the reason behind these statements, we need to understand how Python works, and this is a bit outside the scope of this book, for it's too technical in detail. Let's just say that `map` and `list` comprehensions run at C language speed within the interpreter, while a Python `for` loop is run as Python bytecode within the Python Virtual Machine, which is often much slower.

There are several different implementations of Python. The original one, and still the most common one, is the one written in C. C is one of the most powerful and popular programming languages still used today.

These claims I made come from books and articles that you can find on the Web, but how about we do a small exercise and try to find out for ourselves? I will write a small piece of code that collects the results of `divmod(a, b)` for a certain set of integer pairs `(a, b)`. I will use the `time` function from the `time` module to calculate the elapsed time of the operations that I will perform. Let's go!

`performances.py`

```
from time import time
mx = 5500  # this is the max I could reach with my computer...

t = time()  # start time for the for loop
dmloop = []
for a in range(1, mx):
    for b in range(a, mx):
        dmloop.append(divmod(a, b))
print('for loop: {:.4f} s'.format(time() - t))  # elapsed time

t = time()  # start time for the list comprehension
dmlist = [
    divmod(a, b) for a in range(1, mx) for b in range(a, mx)]
print('list comprehension: {:.4f} s'.format(time() - t))

t = time()  # start time for the generator expression
dmgen = list(
    divmod(a, b) for a in range(1, mx) for b in range(a, mx))
print('generator expression: {:.4f} s'.format(time() - t))

# verify correctness of results and number of items in each list
print(dmloop == dmlist == dmgen, len(dmloop))
```

As you can see, we're creating three lists: `dmloop`, `dmlist`, `dmgen` (`divmod`-`for` loop, `divmod`-`list` comprehension, `divmod`-generator expression). We start with the slowest option, the `for` loops. Then we have a `list` comprehension, and finally a generator expression. Let's see the output:

```
$ python performances.py
for loop: 4.3433 s
list comprehension: 2.7238 s
generator expression: 3.1380 s
True 15122250
```

The `list` comprehension runs in 63% of the time taken by the `for` loop. That's impressive. The generator expression came quite close to that, with a good 72%. The reason the generator expression is slower is that we need to feed it to the `list()` constructor and this has a little bit more overhead compared to a sheer list comprehension.

I would never go with a generator expression in a similar case though, there is no point if at the end we want a list. I would just use a list comprehension, and the result of the previous example proves me right. On the other hand, if I just had to do those `divmod` calculations without retaining the results, then a generator expression would be the way to go because in such a situation a list comprehension would unnecessarily consume a lot of space.

So, to recap: generators are very fast and allow you to save on space. List comprehensions are in general even faster, but don't save on space. Pure Python `for` loops are the slowest option. Let's see a similar example that compares a `for` loop and a `map` call:

`performances.map.py`

```
from time import time
mx = 2 * 10 ** 7

t = time()
absloop = []
for n in range(mx):
    absloop.append(abs(n))
print('for loop: {:.4f} s'.format(time() - t))

t = time()
abslist = [abs(n) for n in range(mx)]
print('list comprehension: {:.4f} s'.format(time() - t))

t = time()
absmap = list(map(abs, range(mx)))
print('map: {:.4f} s'.format(time() - t))

print(absloop == abslist == absmap)
```

This code is conceptually very similar to the previous example. The only thing that has changed is that we're applying the `abs` function instead of the `divmod` one, and we have only one loop instead of two nested ones. Execution gives the following result:

```
$ python performances.map.py
for loop: 3.1283 s
list comprehension: 1.3966 s
map: 1.2319 s
True
```

And `map` wins the race! As I told you before, giving a statement of *what is faster than what* is very tricky. In this case, the `map` call is faster than the list comprehension.

Apart from the case by case little differences though, it's quite clear that the `for` loop option is the slowest one, so let's see what are the reasons we still want to use it.

Don't overdo comprehensions and generators

We've seen how powerful list comprehensions and generator expressions can be. And they are, don't get me wrong, but the feeling that I have when I deal with them is that their complexity grows exponentially. The more you try to do within a single comprehension or a generator expression, the harder it becomes to read, understand, and therefore to maintain or change.

Open a Python console and type in `import this`, let's read the Zen of Python again, in particular, there are a few lines that I think are very important to keep in mind:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.  #
Simple is better than complex.  #
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.  #
Special cases aren't special enough to break the rules.
```

```
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.  #
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

I have put a comment sign on the right of the main focus points here. Comprehensions and generator expressions become hard to read, more implicit than explicit, complex, and they can be hard to explain. Sometimes you have to break them apart using the inside-out technique, to understand why they produce the result they produce.

To give you an example, let's talk a bit more about Pythagorean triples. Just to remind you, a Pythagorean triple is a tuple of positive integers (*a*, *b*, *c*) such that $a^2 + b^2 = c^2$.

We saw earlier in this chapter how to calculate them, but we did it in a very inefficient way because we were scanning all pairs of numbers below a certain threshold, calculating the hypotenuse, and filtering out those that were not producing a triple.

A better way to get a list of Pythagorean triples is to directly generate them. There are many different formulas to do this and we'll use one of them: the **Euclidean formula**.

This formula says that any triple (*a*, *b*, *c*), where $a = m^2 - n^2$, $b = 2mn$, $c = m^2 + n^2$, with *m* and *n* positive integers such that $m > n$, is a Pythagorean triple. For example, when $m = 2$ and $n = 1$, we find the smallest triple: (3, 4, 5).

There is one catch though: consider the triple (6, 8, 10), that is just like (3, 4, 5) with all the numbers multiplied by 2. This triple is definitely Pythagorean, since $6^2 + 8^2 = 10^2$, but we can derive it from (3, 4, 5) simply by multiplying each of its elements by 2. Same goes for (9, 12, 15), (12, 16, 20), and in general for all the triples that we can write as (3k, 4k, 5k), with *k* being a positive integer greater than 1.

A triple that cannot be obtained by multiplying the elements of another one by some factor *k*, is called **primitive**. Another way of stating this is: if the three elements of a triple are **coprime**, then the triple is primitive. Two numbers are coprime when they don't share any prime factor amongst their divisors, that is, their **greatest common divisor** (**GCD**) is 1. For example, 3 and 5 are coprime, while 3 and 6 are not, because they are both divisible by 3.

So, the Euclidean formula tells us that if *m* and *n* are coprime, and *m – n* is odd, the triple they generate is *primitive*. In the following example, we will write a generator expression to calculate all the primitive Pythagorean triples whose hypotenuse (*c*) is less than or equal to some integer *N*. This means we want all triples for which $m^2 + n^2 \leq N$. When *n* is *1*, the formula looks like this: $m^2 \leq N - 1$, which means we can approximate the calculation with an upper bound of $m \leq N^{1/2}$.

So, to recap: *m* must be greater than *n*, they must also be coprime, and their difference *m - n* must be odd. Moreover, in order to avoid useless calculations we'll put the upper bound for *m* at *floor(sqrt(N)) + 1*.

The function `floor` for a real number *x* gives the maximum integer *n* such that *n* < *x*, for example, *floor(3.8) = 3*, *floor(13.1) = 13*. Taking the *floor(sqrt(N)) + 1* means taking the integer part of the square root of *N* and adding a minimal margin just to make sure we don't miss out any number.

Let's put all of this into code, step by step. Let's start by writing a simple `gcd` function that uses **Euclid's algorithm**:

`functions.py`

```
def gcd(a, b):
    """Calculate the Greatest Common Divisor of (a, b). """
    while b != 0:
        a, b = b, a % b
    return a
```

The explanation of Euclid's algorithm is available on the Web, so I won't spend any time here talking about it; we need to concentrate on the generator expression. The next step is to use the knowledge we gathered before to generate a list of primitive Pythagorean triples:

`pythagorean.triple.generation.py`

```
from functions import gcd
N = 50

triples = sorted(                                    # 1
```

```
    ((a, b, c) for a, b, c in (                          # 2
        ((m**2 - n**2), (2 * m * n), (m**2 + n**2))      # 3
        for m in range(1, int(N**.5) + 1)                # 4
        for n in range(1, m)                             # 5
        if (m - n) % 2 and gcd(m, n) == 1                # 6
    ) if c <= N), key=lambda *triple: sum(*triple)       # 7
)

print(triples)
```

There you go. It's not easy to read, so let's go through it line by line. At `#3`, we start a generator expression that is creating triples. You can see from `#4` and `#5` that we're looping on `m` in *[1, M]* with *M* being the integer part of *sqrt(N)*, plus 1. On the other hand, `n` loops within *[1, m)*, to respect the $m > n$ rule. Worth noting how I calculated *sqrt(N)*, that is, `N**.5`, which is just another way to do it that I wanted to show you.

At `#6`, you can see the filtering conditions to make the triples primitive: `(m - n) % 2` evaluates to `True` when `(m - n)` is odd, and `gcd(m, n) == 1` means `m` and `n` are coprime. With these in place, we know the triples will be primitive. This takes care of the innermost generator expression. The outermost one starts at `#2`, and finishes at `#7`. We take the triples (*a*, *b*, *c*) in (...innermost generator...) such that `c <= N`. This is necessary because $m \leq N^{1/2}$ is the lowest upper bound that we can apply, but it doesn't guarantee that *c* will actually be less than or equal to *N*.

Finally, at `#1` we apply sorting, to present the list in order. At `#7`, after the outermost generator expression is closed, you can see that we specify the sorting key to be the sum $a + b + c$. This is just my personal preference, there is no mathematical reason behind it.

So, what do you think? Was it straightforward to read? I don't think so. And believe me, this is still a simple example; I have seen expressions way more complicated than this one.

Unfortunately some programmers think that writing code like this is cool, that it's some sort of demonstration of their superior intellectual powers, of their ability to quickly read and digest intricate code.

Within a professional environment though, I find myself having much more respect for those who write efficient, clean code, and manage to keep ego out the door. Conversely, those who don't, will produce lines at which you will stare for a long time while swearing in three languages (at least this is what I do).

Now, let's see if we can rewrite this code into something easier to read:

pythagorean.triple.generation.for.py

```
from functions import gcd

def gen_triples(N):
    for m in range(1, int(N**.5) + 1):            # 1
        for n in range(1, m):                     # 2
            if (m - n) % 2 and gcd(m, n) == 1:    # 3
                c = m**2 + n**2                   # 4
                if c <= N:                        # 5
                    a = m**2 - n**2               # 6
                    b = 2 * m * n                 # 7
                    yield (a, b, c)               # 8

triples = sorted(
    gen_triples(50), key=lambda *triple: sum(*triple))  # 9
print(triples)
```

I feel so much better already. Let's go through this code as well, line by line. You'll see how easier it is to understand.

We start looping at `#1` and `#2`, in exactly the same way we were looping in the previous example. On line `#3`, we have the filtering for primitive triples. On line `#4`, we deviate a bit from what we were doing before: we calculate `c`, and on line `#5`, we filter on `c` being less than or equal to `N`. Only when `c` satisfies that condition, we calculate `a` and `b`, and yield the resulting tuple. It's always good to delay all calculations for as much as possible so that we don't waste time, in case eventually we have to discard those results.

On the last line, before printing the result, we apply sorting with the same key we were using in the generator expression example.

I hope you agree, this example is easier to understand. And I promise you, if you have to modify the code one day, you'll find that modifying this one is easy, while to modify the other version will take much longer (and it will be more error prone).

Both examples, when run, print the following:

```
$ python pythagorean.triple.generation.py
[(3, 4, 5), (5, 12, 13), (15, 8, 17), (7, 24, 25), (21, 20, 29), (35, 12,
37), (9, 40, 41)]
```

The moral of the story is, try and use comprehensions and generator expressions as much as you can, but if the code starts to be complicated to modify or to read, you may want to refactor into something more readable. There is nothing wrong with this.

Name localization

Now that we are familiar with all types of comprehensions and generator expression, let's talk about name localization within them. Python 3.* localizes loop variables in all four forms of comprehensions: `list`, `dict`, `set`, and generator expressions. This behavior is therefore different from that of the `for` loop. Let's see a simple example to show all the cases:

`scopes.py`

```
A = 100
ex1 = [A for A in range(5)]
print(A)  # prints: 100

ex2 = list(A for A in range(5))
print(A)  # prints: 100

ex3 = dict((A, 2 * A) for A in range(5))
print(A)  # prints: 100

ex4 = set(A for A in range(5))
print(A)  # prints: 100

s = 0
for A in range(5):
    s += A
print(A)  # prints: 4
```

In the preceding code, we declare a global name `A = 100`, and then we exercise the four comprehensions: list, generator expression, dictionary, and set. None of them alter the global name `A`. Conversely, you can see at the end that the `for` loop modifies it. The last print statement prints 4.

Let's see what happens if `A` wasn't there:

`scopes.noglobal.py`

```
ex1 = [A for A in range(5)]
print(A)  # breaks: NameError: name 'A' is not defined
```

The preceding code would work the same with any of the four types of comprehensions. After we run the first line, `A` is not defined in the global namespace.

Once again, the `for` loop behaves differently:

`scopes.for.py`

```
s = 0
for A in range(5):
    s += A
print(A)  # prints: 4
print(globals())
```

The preceding code shows that after a `for` loop, if the loop variable wasn't defined before it, we can find it in the global frame. To make sure of it, let's take a peek at it by calling the `globals()` built-in function:

```
$ python scopes.for.py
4
{'__spec__': None, '__name__': '__main__', 's': 10, 'A': 4, '__doc__':
None, '__cached__': None, '__package__': None, '__file__': 'scopes.
for.py', '__loader__': <_frozen_importlib.SourceFileLoader object at
0x7f05a5a183c8>, '__builtins__': <module 'builtins' (built-in)>}
```

Together with a lot of other boilerplate stuff, we can spot `'A': 4`.

Generation behavior in built-ins

Amongst the built-in types, the generation behavior is now quite common. This is a major difference between Python 2 and Python 3. A lot of functions such as `map`, `zip`, and `filter` have been transformed so that they return objects that behave like iterables. The idea behind this change is that if you need to make a list of those results you can always wrap the call in a `list()` class, and you're done. On the other hand, if you just need to iterate and want to keep the impact on memory as light as possible, you can use those functions safely.

Another notable example is the `range` function. In Python 2 it returns a list, and there is another function called `xrange` that returns an object that you can iterate on, which generates the numbers on the fly. In Python 3 this function has gone, and `range` now behaves like it.

But this concept in general is now quite widespread. You can find it in the `open()` function, which is used to operate on file objects (we'll see it in one of the next chapters), but also in `enumerate`, in the dictionary `keys`, `values`, and `items` methods, and several other places.

It all makes sense: Python's aim is to try and reduce the memory footprint by avoiding wasting space wherever is possible, especially in those functions and methods that are used extensively in most situations.

Do you remember at the beginning of this chapter? I said that it makes more sense to optimize the performances of code that has to deal with a lot of objects, rather than shaving off a few milliseconds from a function that we call twice a day.

One last example

Before we part from this chapter, I'll show you a simple problem that I submitted to candidates for a Python developer role in a company I used to work for.

The problem is the following: given the sequence `0 1 1 2 3 5 8 13 21 ...` write a function that would return the terms of this sequence up to some limit `N`.

If you haven't recognized it, that is the Fibonacci sequence, which is defined as *F(0) = 0*, *F(1) = 1* and, for any *n > 1*, *F(n) = F(n-1) + F(n-2)*. This sequence is excellent to test knowledge about recursion, memoization techniques and other technical details, but in this case it was a good opportunity to check whether the candidate knew about generators (and too many so called Python coders didn't, when I was interviewing them).

Let's start from a rudimentary version of a function, and then improve on it:

fibonacci.first.py

```
def fibonacci(N):
    """Return all fibonacci numbers up to N. """
    result = [0]
    next_n = 1
    while next_n <= N:
        result.append(next_n)
        next_n = sum(result[-2:])
    return result

print(fibonacci(0))  # [0]
print(fibonacci(1))  # [0, 1, 1]
print(fibonacci(50))  # [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

From the top: we set up the `result` list to a starting value of `[0]`. Then we start the iteration from the next element (`next_n`), which is `1`. While the next element is not greater than `N`, we keep appending it to the list and calculating the next. We calculate the next element by taking a slice of the last two elements in the `result` list and passing it to the `sum` function. Add some `print` statements here and there if this is not clear to you, but by now I would expect it not to be an issue.

When the condition of the `while` loop evaluates to `False`, we exit the loop and return `result`. You can see the result of those `print` statements in the comments next to each of them.

At this point, I would ask the candidate the following question: "What if I just wanted to iterate over those numbers?" A good candidate would then change the code like the next listing (an excellent candidate would have started with it!):

`fibonacci.second.py`

```
def fibonacci(N):
    """Return all fibonacci numbers up to N. """
    yield 0
    if N == 0:
        return
    a = 0
    b = 1
    while b <= N:
        yield b
        a, b = b, a + b


print(list(fibonacci(0)))  # [0]
print(list(fibonacci(1)))  # [0, 1, 1]
print(list(fibonacci(50))) # [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

This is actually one of the solutions I was given. I don't know why I kept it, but I'm glad I did so I can show it to you. Now, the `fibonacci` function is a *generator function*. First we yield `0`, then if `N` is `0` we return (this will cause a `StopIteration` exception to be raised). If that's not the case, we start iterating, yielding `b` at every loop cycle, and then updating `a` and `b`. All we need in order to be able to produce the next element of the sequence is the past two: `a` and `b`, respectively.

This code is much better, has a lighter memory footprint and all we have to do to get a list of Fibonacci numbers is to wrap the call with `list()`, as usual.

But what about elegance? I cannot leave the code like that. It was decent for an interview, where the focus is more on functionality than elegance, but here I'd like to show you a nicer version:

`fibonacci.elegant.py`

```
def fibonacci(N):
    """Return all fibonacci numbers up to N. """
    a, b = 0, 1
    while a <= N:
        yield a
        a, b = b, a + b
```

Much better. The whole body of the function is four lines, five if you count the docstring. Notice how in this case using tuple assignment (`a, b = 0, 1` and `a, b = b, a + b`) helps in making the code shorter, and more readable. It's one of the features of Python I like a lot.

Summary

In this chapter, we explored the concept of iteration and generation a bit more deeply. We saw the `map`, `zip` and `filter` functions quite in detail, and how to use them as an alternative to a regular `for` loop approach.

Then we saw the concept of comprehensions, for lists, dictionaries, and sets. We saw their syntax and how to use them as an alternative to both the classic `for` loop approach and also to the use of `map`, `zip`, and `filter` functions.

Finally, we talked about the concept of generation, in two forms: generator functions and expressions. We learned how to save time and space by using generation techniques and saw how they can make possible what wouldn't normally be if we used a conventional approach based on lists.

We talked about performances, and saw that `for` loops are last in terms of speed, but they provide the best readability and flexibility to change. On the other hand, functions such as `map` and `filter` can be much faster, and comprehensions may be even better.

The complexity of the code written using these techniques grows exponentially so, in order to favor readability and ease of maintainability, we still need to use the classic `for` loop approach at times. Another difference is in the name localization, where the `for` loop behaves differently from all other types of comprehensions.

The next chapter will be all about objects and classes. Structurally similar to this one, in that we won't explore many different subjects, rather, just a few of them, but we'll try to dive a little bit more deeply.

Make sure you understand well the concepts of this chapter before jumping to the next one. We're building a wall brick by brick, and if the foundation is not solid, we won't get very far.

6
Advanced Concepts – OOP, Decorators, and Iterators

"La classe non è acqua. (Class will out)"

- Italian saying

I could probably write a small book about **object-oriented programming** (referred to as **OOP** henceforth) and classes. In this chapter, I'm facing the hard challenge of finding the balance between breadth and depth. There are simply too many things to tell, and there's plenty of them that would take more than this whole chapter if I described them alone in depth. Therefore, I will try to give you what I think is a good panoramic view of the fundamentals, plus a few things that may come in handy in the next chapters. Python's official documentation will help in filling the gaps.

We're going to explore three important concepts in this chapter: decorators, OOP, and iterators.

Decorators

In the previous chapter, I measured the execution time of various expressions. If you recall, I had to initialize a variable to the start time, and subtract it from the current time after execution in order to calculate the elapsed time. I also printed it on the console after each measurement. That was very tedious.

Every time you find yourself repeating things, an alarm bell should go off. Can you put that code in a function and avoid repetition? The answer most of the time is *yes*, so let's look at an example.

decorators/time.measure.start.py

```
from time import sleep, time

def f():
    sleep(.3)

def g():
    sleep(.5)

t = time()
f()
print('f took: ', time() - t)  # f took: 0.3003859519958496

t = time()
g()
print('g took:', time() - t)  # g took: 0.5005719661712646
```

In the preceding code, I defined two functions, `f` and `g`, which do nothing but sleep (by 0.3 and 0.5 seconds respectively). I used the `sleep` function to suspend the execution for the desired amount of time. I also highlighted how we calculate the time elapsed by setting `t` to the current time and then subtracting it when the task is done. You can see that the measure is pretty accurate.

Now, how do we avoid repeating that code and those calculations? One first potential approach could be the following:

decorators/time.measure.dry.py

```
from time import sleep, time

def f():
    sleep(.3)

def g():
    sleep(.5)

def measure(func):
    t = time()
    func()
```

```
        print(func.__name__, 'took:', time() - t)

    measure(f)  # f took: 0.30041074752807617
    measure(g)  # g took: 0.5006198883056641
```

Ah, much better now. The whole timing mechanism has been encapsulated into a function so we don't repeat code. We print the function name dynamically and it's easy enough to code. What if we need to pass arguments to the function we measure? This code would get just a bit more complicated, so let's see an example.

decorators/time.measure.arguments.py

```
    from time import sleep, time

    def f(sleep_time=0.1):
        sleep(sleep_time)

    def measure(func, *args, **kwargs):
        t = time()
        func(*args, **kwargs)
        print(func.__name__, 'took:', time() - t)

    measure(f, sleep_time=0.3)  # f took: 0.3004162311553955
    measure(f, 0.2)  # f took: 0.20028162002563477
```

Now, `f` is expecting to be fed `sleep_time` (with a default value of `0.1`). I also had to change the `measure` function so that it is now accepting a function, any variable positional arguments, and any variable keyword arguments. In this way, whatever we call `measure` with, we redirect those arguments to the call to `f` we do inside.

This is very good, but we can push it a little bit further. Let's say we want to somehow have that timing behavior built-in in the `f` function, so that we could just call it and have that measure taken. Here's how we could do it:

decorators/time.measure.deco1.py

```
    from time import sleep, time

    def f(sleep_time=0.1):
        sleep(sleep_time)

    def measure(func):
        def wrapper(*args, **kwargs):
            t = time()
            func(*args, **kwargs)
            print(func.__name__, 'took:', time() - t)
```

```
        return wrapper

    f = measure(f)  # decoration point

    f(0.2)  # f took: 0.2002875804901123
    f(sleep_time=0.3)  # f took: 0.3003721237182617
    print(f.__name__)  # wrapper  <- ouch!
```

The preceding code is probably not so straightforward. I confess that, even today, it sometimes requires me some serious concentration to understand some decorators, they can be pretty nasty. Let's see what happens here. The magic is in the **decoration point**. We basically reassign `f` with whatever is returned by `measure` when we call it with `f` as an argument. Within `measure`, we define another function, `wrapper`, and then we return it. So, the net effect is that after the decoration point, when we call `f`, we're actually calling `wrapper`. Since the wrapper inside is calling `func`, which is `f`, we are actually closing the loop like that. If you don't believe me, take a look at the last line.

`wrapper` is actually... a wrapper. It takes variable and positional arguments, and calls `f` with them. It also does the time measurement trick around the call.

This technique is called **decoration**, and `measure` is, at all effects, a **decorator**. This paradigm became so popular and widely used that at some point, Python added a special syntax for it (check **PEP 318**). Let's explore three cases: one decorator, two decorators, and one decorator that takes arguments.

`decorators/syntax.py`

```
    def func(arg1, arg2, ...):
        pass
    func = decorator(func)

    # is equivalent to the following:

    @decorator
    def func(arg1, arg2, ...):
        pass
```

Basically, instead of manually reassigning the function to what was returned by the decorator, we prepend the definition of the function with the special syntax `@decorator_name`.

We can apply multiple decorators to the same function in the following way:

`decorators/syntax.py`

```
def func(arg1, arg2, ...):
    pass
func = deco1(deco2(func))

# is equivalent to the following:

@deco1
@deco2
def func(arg1, arg2, ...):
    pass
```

When applying multiple decorators, pay attention to the order, should it matter. In the preceding example, `func` is decorated with `deco2` first, and the result is decorated with `deco1`. A good rule of thumb is: *the closer the decorator to the function, the sooner it is applied*.

Some decorators can take arguments. This technique is generally used to produce other decorators. Let's look at the syntax, and then we'll see an example of it.

`decorators/syntax.py`

```
def func(arg1, arg2, ...):
    pass
func = decoarg(argA, argB)(func)

# is equivalent to the following:

@decoarg(argA, argB)
def func(arg1, arg2, ...):
    pass
```

As you can see, this case is a bit different. First `decoarg` is called with the given arguments, and then its return value (the actual decorator) is called with `func`. Before I give you another example, let's fix one thing that is bothering me. I don't want to lose the original function name and docstring (and the other attributes as well, check the documentation for the details) when I decorate it. But because inside our decorator we return `wrapper`, the original attributes from `func` are lost and `f` ends up being assigned the attributes of `wrapper`. There is an easy fix for that from `functools`, a wonderful module from the Python standard library. I will fix the last example, and I will also rewrite its syntax to use the `@` operator.

`decorators/time.measure.deco2.py`

```
from time import sleep, time
from functools import wraps
```

```
def measure(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        t = time()
        func(*args, **kwargs)
        print(func.__name__, 'took:', time() - t)
    return wrapper

@measure
def f(sleep_time=0.1):
    """I'm a cat. I love to sleep! """
    sleep(sleep_time)

f(sleep_time=0.3)  # f took: 0.30039525032043457
print(f.__name__, ':', f.__doc__)
# f : I'm a cat. I love to sleep!
```

Now we're talking! As you can see, all we need to do is to tell Python that `wrapper` actually wraps `func` (by means of the `wraps` function), and you can see that the original name and docstring are now maintained.

Let's see another example. I want a decorator that prints an error message when the result of a function is greater than a threshold. I will also take this opportunity to show you how to apply two decorators at once.

`decorators/two.decorators.py`

```
from time import sleep, time
from functools import wraps

def measure(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        t = time()
        result = func(*args, **kwargs)
        print(func.__name__, 'took:', time() - t)
        return result
    return wrapper

def max_result(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        if result > 100:
```

```
                print('Result is too big ({0}). Max allowed is 100.'
                    .format(result))
            return result
        return wrapper

@measure
@max_result
def cube(n):
    return n ** 3

print(cube(2))
print(cube(5))
```

Take your time in studying the preceding example until you are sure you understand it well. If you do, I don't think there is any decorator you won't be able to write afterwards.

I had to enhance the `measure` decorator, so that its `wrapper` now returns the result of the call to `func`. The `max_result` decorator does that as well, but before returning, it checks that `result` is not greater than 100, which is the maximum allowed.

I decorated `cube` with both of them. First, `max_result` is applied, then `measure`. Running this code yields this result:

```
$ python two.decorators.py
cube took: 7.62939453125e-06  #
8  #
Result is too big (125). Max allowed is 100.
cube took: 1.1205673217773438e-05
125
```

For your convenience, I put a `#` to the right of the results of the first call: `print(cube(2))`. The result is 8, and therefore it passes the threshold check silently. The running time is measured and printed. Finally, we print the result (`8`).

On the second call, the result is `125`, so the error message is printed, the result returned, and then it's the turn of `measure`, which prints the running time again, and finally, we print the result (`125`).

Had I decorated the `cube` function with the same two decorators but in a different order, the error message would follow the line that prints the running time, instead of preceding it.

A decorator factory

Let's simplify this example now, going back to a single decorator: `max_result`. I want to make it so that I can decorate different functions with different thresholds, and I don't want to write one decorator for each threshold. Let's amend `max_result` so that it allows us to decorate functions specifying the threshold dynamically.

`decorators/decorators.factory.py`

```
from functools import wraps

def max_result(threshold):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            result = func(*args, **kwargs)
            if result > threshold:
                print(
                    'Result is too big ({0}). Max allowed is {1}.'
                    .format(result, threshold))
            return result
        return wrapper
    return decorator

@max_result(75)
def cube(n):
    return n ** 3

print(cube(5))
```

This preceding code shows you how to write a **decorator factory**. If you recall, decorating a function with a decorator that takes arguments is the same as writing `func = decorator(argA, argB)(func)`, so when we decorate `cube` with `max_result(75)`, we're doing `cube = max_result(75)(cube)`.

Let's go through what happens, step by step. When we call `max_result(75)`, we enter its body. A `decorator` function is defined inside, which takes a function as its only argument. Inside that function, the usual decorator trick is performed. We define a wrapper, inside of which we check the result of the original function's call. The beauty of this approach is that from the innermost level, we can still refer to both `func` and `threshold`, which allows us to set the threshold dynamically.

`wrapper` returns `result`, `decorator` returns `wrapper`, and `max_result` returns `decorator`. This means that our call `cube = max_result(75)(cube)`, actually becomes `cube = decorator(cube)`. Not just any `decorator` though, but one for which `threshold` has the value `75`. This is achieved by a mechanism called **closure**, which is outside of the scope of this chapter but nonetheless very interesting, so I mentioned it for you to do some research on it.

Running the last example produces the following result:

```
$ python decorators.factory.py
Result is too big (125). Max allowed is 75.
125
```

The preceding code allows me to use the `max_result` decorator with different thresholds at my own will, like this:

`decorators/decorators.factory.py`

```
@max_result(75)
def cube(n):
    return n ** 3

@max_result(100)
def square(n):
    return n ** 2

@max_result(1000)
def multiply(a, b):
    return a * b
```

Note that every decoration uses a different `threshold` value.

Decorators are very popular in Python. They are used quite often and they simplify (and beautify, I dare say) the code a lot.

Object-oriented programming

It's been quite a long and hopefully nice journey and, by now, we should be ready to explore object-oriented programming. I'll use the definition from *Kindler, E.; Krivy, I. (2011). Object-Oriented Simulation of systems with sophisticated control. International Journal of General Systems*, and adapt it to Python:

> *Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which are data structures that contain data, in the form of attributes, and code, in the form of functions known as methods. A distinguishing feature of objects is that an object's method can access and often modify the data attributes of the object with which they are associated (objects have a notion of "self"). In OO programming, computer programs are designed by making them out of objects that interact with one another.*

Python has full support for this paradigm. Actually, as we have already said, *everything in Python is an object*, so this shows that OOP is not just supported by Python, but it's part of its very core.

The two main players in OOP are **objects** and **classes**. Classes are used to create objects (objects are instances of the classes with which they were created), so we could see them as instance factories. When objects are created by a class, they inherit the class attributes and methods. They represent concrete items in the program's domain.

The simplest Python class

I will start with the simplest class you could ever write in Python.

oop/simplest.class.py

```
class Simplest():  # when empty, the braces are optional
    pass

print(type(Simplest))  # what type is this object?

simp = Simplest()  # we create an instance of Simplest: simp
print(type(simp))  # what type is simp?
# is simp an instance of Simplest?
print(type(simp) == Simplest)  # There's a better way for this
```

Let's run the preceding code and explain it line by line:

```
$ python oop/simplest.class.py
<class 'type'>
<class '__main__.Simplest'>
True
```

The `Simplest` class I defined only has the `pass` instruction for its body, which means it doesn't have any custom attributes or methods. I will print its type (`__main__` is the name of the scope in which top-level code executes), and I am aware that, in the comment, I wrote *object* instead of *class*. It turns out that, as you can see by the result of that `print`, *classes are actually objects*. To be precise, they are instances of `type`. Explaining this concept would lead to a talk about **metaclasses** and **metaprogramming**, advanced concepts that require a solid grasp of the fundamentals to be understood and alas this is beyond the scope of this chapter. As usual, I mentioned it to leave a pointer for you, for when you'll be ready to dig deeper.

Let's go back to the example: I used `Simplest` to create an instance, `simp`. You can see that the *syntax to create an instance is the same we use to call a function*.

Then we print what type `simp` belongs to and we verify that `simp` is in fact an instance of `Simplest`. I'll show you a better way of doing this later on in the chapter.

Up to now, it's all very simple. What happens when we write `class ClassName() : pass`, though? Well, what Python does is create a class object and assign it a name. This is very similar to what happens when we declare a function using `def`.

Class and object namespaces

After the class object has been created (which usually happens when the module is first imported), it basically represents a namespace. We can call that class to create its instances. Each instance inherits the class attributes and methods and is given its own namespace. We already know that, to walk a namespace, all we need to do is to use the dot (`.`) operator.

Let's look at another example:

oop/class.namespaces.py

```
class Person():
    species = 'Human'

print(Person.species)  # Human
```

```
Person.alive = True  # Added dynamically!
print(Person.alive)  # True

man = Person()
print(man.species)  # Human (inherited)
print(man.alive)  # True (inherited)

Person.alive = False
print(man.alive)  # False (inherited)

man.name = 'Darth'
man.surname = 'Vader'
print(man.name, man.surname)  # Darth Vader
```

In the preceding example, I have defined a class attribute called `species`. Any variable defined in the body of a class is an attribute that belongs to that class. In the code, I have also defined `Person.alive`, which is another class attribute. You can see that there is no restriction on accessing that attribute from the class. You can see that `man`, which is an instance of `Person`, inherits both of them, and reflects them instantly when they change.

`man` has also two attributes which belong to its own namespace and therefore are called **instance attributes**: `name` and `surname`.

Class attributes are shared amongst all instances, while instance attributes are not; therefore, you should use class attributes to provide the states and behaviors to be shared by all instances, and use instance attributes for data that belongs just to one specific object.

Attribute shadowing

When you search for an attribute in an object, if it is not found, Python keeps searching in the class that was used to create that object (and keeps searching until it's either found or the end of the inheritance chain is reached). This leads to an interesting shadowing behavior. Let's look at an example:

`oop/class.attribute.shadowing.py`

```
class Point():
    x = 10
    y = 7

p = Point()
```

```
print(p.x)  # 10 (from class attribute)
print(p.y)  # 7 (from class attribute)

p.x = 12  # p gets its own 'x' attribute
print(p.x)  # 12 (now found on the instance)
print(Point.x)  # 10 (class attribute still the same)

del p.x  # we delete instance attribute
print(p.x)  # 10 (now search has to go again to find class attr)

p.z = 3  # let's make it a 3D point
print(p.z)  # 3

print(Point.z)
# AttributeError: type object 'Point' has no attribute 'z'
```

The preceding code is very interesting. We have defined a class called `Point` with two class attributes, `x` and `y`. When we create an instance, `p`, you can see that we can print both `x` and `y` from `p`'s namespace (`p.x` and `p.y`). What happens when we do that is that Python doesn't find any `x` or `y` attributes on the instance, and therefore searches the class, and finds them there.

Then we give `p` its own `x` attribute by assigning `p.x = 12`. This behavior may appear a bit weird at first, but if you think about it, it's exactly the same as what happens in a function that declares `x = 12` when there is a global `x = 10` outside. We know that `x = 12` won't affect the global one, and for classes and instances, it is exactly the same.

After assigning `p.x = 12`, when we print it, the search doesn't need to read the class attributes, because `x` is found on the instance, therefore we get `12` printed out.

We also print `Point.x` which refers to `x` in the class namespace.

And then, we delete `x` from the namespace of `p`, which means that, on the next line, when we print it again, Python will go again and search for it in the class, because it won't be found in the instance any more.

The last three lines show you that assigning attributes to an instance doesn't mean that they will be found in the class. Instances get whatever is in the class, but the opposite is not true.

What do you think about putting the `x` and `y` coordinates as class attributes? Do you think it was a good idea?

I, me, and myself – using the self variable

From within a class method we can refer to an instance by means of a special argument, called `self` by convention. `self` is always the first attribute of an instance method. Let's examine this behavior together with how we can share, not just attributes, but methods with all instances.

oop/class.self.py

```
class Square():
    side = 8
    def area(self):  # self is a reference to an instance
        return self.side ** 2

sq = Square()
print(sq.area())  # 64 (side is found on the class)
print(Square.area(sq))  # 64 (equivalent to sq.area())

sq.side = 10
print(sq.area())  # 100 (side is found on the instance)
```

Note how the `area` method is used by `sq`. The two calls, `Square.area(sq)` and `sq.area()`, are equivalent, and teach us how the mechanism works. Either you pass the instance to the method call (`Square.area(sq)`), which within the method will be called `self`, or you can use a more comfortable syntax: `sq.area()` and Python will translate that for you behind the curtains.

Let's look at a better example:

oop/class.price.py

```
class Price():
    def final_price(self, vat, discount=0):
        """Returns price after applying vat and fixed discount."""
        return (self.net_price * (100 + vat) / 100) - discount

p1 = Price()
p1.net_price = 100
print(Price.final_price(p1, 20, 10))  # 110 (100 * 1.2 - 10)
print(p1.final_price(20, 10))  # equivalent
```

The preceding code shows you that nothing prevents us from using arguments when declaring methods. We can use the exact same syntax as we used with the function, but we need to remember that the first argument will always be the instance.

Initializing an instance

Have you noticed how, before calling `p1.final_price(...)`, we had to assign `net_price` to `p1`? There is a better way to do it. In other languages, this would be called a **constructor**, but in Python, it's not. It is actually an **initializer**, since it works on an already created instance, and therefore it's called `__init__`. It's a *magic method*, which is run right after the object is created. Python objects also have a `__new__` method, which is the actual constructor. In practice, it's not so common to have to override it though, it's a practice that is mostly used when coding metaclasses, which is a fairly advanced topic that we won't explore in the book.

`oop/class.init.py`

```
class Rectangle():
    def __init__(self, sideA, sideB):
        self.sideA = sideA
        self.sideB = sideB

    def area(self):
        return self.sideA * self.sideB

r1 = Rectangle(10, 4)
print(r1.sideA, r1.sideB)  # 10 4
print(r1.area())  # 40

r2 = Rectangle(7, 3)
print(r2.area())  # 21
```

Things are finally starting to take shape. When an object is created, the `__init__` method is automatically run for us. In this case, I coded it so that when we create an object (by calling the class name like a function), we pass arguments to the creation call, like we would on any regular function call. The way we pass parameters follows the signature of the `__init__` method, and therefore, in the two creation statements, `10` and `7` will be `sideA` for `r1` and `r2` respectively, while `4` and `3` will be `sideB`. You can see that the call to `area()` from `r1` and `r2` reflects that they have different instance arguments.

Setting up objects in this way is much nicer and convenient.

OOP is about code reuse

By now it should be pretty clear: *OOP is all about code reuse*. We define a class, we create instances, and those instances use methods that are defined only in the class. They will behave differently according to how the instances have been set up by the initializer.

Inheritance and composition

But this is just half of the story, *OOP is much more powerful*. We have two main design constructs to exploit: inheritance and composition.

Inheritance means that two objects are related by means of an *Is-A* type of relationship. On the other hand, **composition** means that two objects are related by means of a *Has-A* type of relationship. It's all very easy to explain with an example:

oop/class.inheritance.py

```
class Engine():
    def start(self):
        pass

    def stop(self):
        pass

class ElectricEngine(Engine):  # Is-A Engine
    pass

class V8Engine(Engine):  # Is-A Engine
    pass

class Car():
    engine_cls = Engine

    def __init__(self):
        self.engine = self.engine_cls()  # Has-A Engine

    def start(self):
        print(
            'Starting engine {0} for car {1}... Wroom, wroom!'
            .format(
                self.engine.__class__.__name__,
                self.__class__.__name__)
        )
        self.engine.start()

    def stop(self):
        self.engine.stop()

class RaceCar(Car):  # Is-A Car
    engine_cls = V8Engine
```

```
class CityCar(Car):  # Is-A Car
    engine_cls = ElectricEngine

class F1Car(RaceCar):  # Is-A RaceCar and also Is-A Car
    engine_cls = V8Engine

car = Car()
racecar = RaceCar()
citycar = CityCar()
f1car = F1Car()
cars = [car, racecar, citycar, f1car]

for car in cars:
    car.start()

""" Prints:
Starting engine Engine for car Car... Wroom, wroom!
Starting engine V8Engine for car RaceCar... Wroom, wroom!
Starting engine ElectricEngine for car CityCar... Wroom, wroom!
Starting engine V8Engine for car F1Car... Wroom, wroom!
"""
```

The preceding example shows you both the *Is-A* and *Has-A* types of relationships between objects. First of all, let's consider `Engine`. It's a simple class that has two methods, `start` and `stop`. We then define `ElectricEngine` and `V8Engine`, which both inherit from `Engine`. You can see that by the fact that when we define them, we put `Engine` within the braces after the class name.

This means that both `ElectricEngine` and `V8Engine` inherit attributes and methods from the `Engine` class, which is said to be their **base class**.

The same happens with cars. `Car` is a base class for both `RaceCar` and `CityCar`. `RaceCar` is also the base class for `F1Car`. Another way of saying this is that `F1Car` inherits from `RaceCar`, which inherits from `Car`. Therefore, `F1Car` *Is-A* `RaceCar` and `RaceCar` *Is-A* `Car`. Because of the transitive property, we can say that `F1Car` *Is-A* `Car` as well. `CityCar` too, *Is-A* `Car`.

When we define `class A(B): pass`, we say `A` is the *child* of `B`, and `B` is the *parent* of `A`. *parent* and *base* are synonyms, as well as *child* and *derived*. Also, we say that a class inherits from another class, or that it extends it.

This is the inheritance mechanism.

On the other hand, let's go back to the code. Each class has a class attribute, `engine_cls`, which is a reference to the engine class we want to assign to each type of car. `Car` has a generic `Engine`, while the two race cars have a powerful V8 engine, and the city car has an electric one.

When a car is created in the initializer method `__init__`, we create an instance of whatever engine class is assigned to the car, and set it as `engine` instance attribute.

It makes sense to have `engine_cls` shared amongst all class instances because it's quite likely that the same instances of a car will have the same kind of engine. On the other hand, it wouldn't be good to have one single engine (an instance of any `Engine` class) as a class attribute, because we would be sharing one engine amongst all instances, which is incorrect.

The type of relationship between a car and its engine is a *Has-A* type. A car *Has-A* engine. This is called **composition**, and reflects the fact that objects can be made of many other objects. A car *Has-A* engine, gears, wheels, a frame, doors, seats, and so on.

When designing OOP code, it is of vital importance to describe objects in this way so that we can use inheritance and composition correctly to structure our code in the best way.

Before we leave this paragraph, let's check if I told you the truth with another example:

`oop/class.issubclass.isinstance.py`

```
car = Car()
racecar = RaceCar()
f1car = F1Car()
cars = [(car, 'car'), (racecar, 'racecar'), (f1car, 'f1car')]
car_classes = [Car, RaceCar, F1Car]

for car, car_name in cars:
    for class_ in car_classes:
        belongs = isinstance(car, class_)
        msg = 'is a' if belongs else 'is not a'
        print(car_name, msg, class_.__name__)

""" Prints:
car is a Car
car is not a RaceCar
```

```
car is not a F1Car
racecar is a Car
racecar is a RaceCar
racecar is not a F1Car
f1car is a Car
f1car is a RaceCar
f1car is a F1Car
"""
```

As you can see, `car` is just an instance of `Car`, while `racecar` is an instance of `RaceCar` (and of `Car` by extension) and `f1car` is an instance of `F1Car` (and of both `RaceCar` and `Car`, by extension). A *banana* is an instance of *Banana*. But, also, it is a *Fruit*. Also, it is *Food*, right? This is the same concept.

To check if an object is an instance of a class, use the `isinstance` method. It is recommended over sheer type comparison (`type(object) == Class`).

Let's also check inheritance, same setup, with different `for` loops:

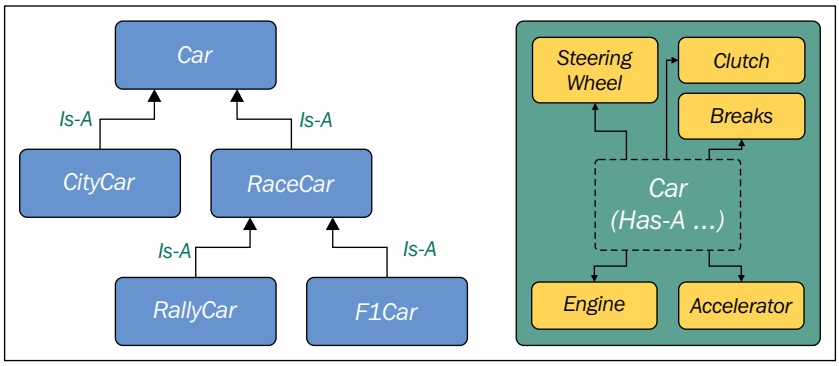
`oop/class.issubclass.isinstance.py`

```
for class1 in car_classes:
    for class2 in car_classes:
        is_subclass = issubclass(class1, class2)
        msg = '{0} a subclass of'.format(
            'is' if is_subclass else 'is not')
        print(class1.__name__, msg, class2.__name__)

""" Prints:
Car is a subclass of Car
Car is not a subclass of RaceCar
Car is not a subclass of F1Car
RaceCar is a subclass of Car
RaceCar is a subclass of RaceCar
RaceCar is not a subclass of F1Car
F1Car is a subclass of Car
F1Car is a subclass of RaceCar
F1Car is a subclass of F1Car
"""
```

Interestingly, we learn that *a class is a subclass of itself*. Check the output of the preceding example to see that it matches the explanation I provided.

One thing to notice about conventions is that class names are always written using *CapWords*, which means *ThisWayIsCorrect*, as opposed to functions and methods, which are written *this_way_is_correct*. Also, when in the code you want to use a name which is a Python-reserved keyword or built-in function or class, the convention is to add a trailing underscore to the name. In the first `for` loop example, I'm looping through the class names using `for class_ in ...`, because `class` is a reserved word. But you already knew all this because you have thoroughly studied PEP8, right?

To help you picture the difference between *Is-A* and *Has-A*, take a look at the following diagram:



Accessing a base class

We've already seen class declarations like `class ClassA: pass` and `class ClassB(BaseClassName): pass`. When we don't specify a base class explicitly, Python will set the special **object** class as the base class for the one we're defining. Ultimately, all classes derive from `object`. Note that, if you don't specify a base class, braces are optional.

Therefore, writing `class A: pass` or `class A(): pass` or `class A(object): pass` is exactly the same thing. *object* is a special class in that it has the methods that are common to all Python classes, and it doesn't allow you to set any attributes on it.

Let's see how we can access a base class from within a class.

oop/super.duplication.py

```
class Book:
    def __init__(self, title, publisher, pages):
        self.title = title
        self.publisher = publisher
        self.pages = pages

class Ebook(Book):
    def __init__(self, title, publisher, pages, format_):
        self.title = title
        self.publisher = publisher
        self.pages = pages
        self.format_ = format_
```

Take a look at the preceding code. I highlighted the part of `Ebook` initialization that is duplicated from its base class `Book`. This is quite bad practice because we now have two sets of instructions that are doing the same thing. Moreover, any change in the signature of `Book.__init__` will not reflect in `Ebook`. We know that `Ebook` *Is-A* `Book`, and therefore we would probably want changes to be reflected in the children classes.

Let's see one way to fix this issue:

oop/super.explicit.py

```
class Book:
    def __init__(self, title, publisher, pages):
        self.title = title
        self.publisher = publisher
        self.pages = pages

class Ebook(Book):
    def __init__(self, title, publisher, pages, format_):
        Book.__init__(self, title, publisher, pages)
        self.format_ = format_

ebook = Ebook('Learning Python', 'Packt Publishing', 360, 'PDF')
print(ebook.title)  # Learning Python
print(ebook.publisher)  # Packt Publishing
print(ebook.pages)  # 360
print(ebook.format_)  # PDF
```

Now, that's better. We have removed that nasty duplication. Basically, we tell Python to call the `__init__` method of the `Book` class, and we feed `self` to the call, making sure that we bind that call to the present instance.

If we modify the logic within the `__init__` method of `Book`, we don't need to touch `Ebook`, it will auto adapt to the change.

This approach is good, but we can still do a bit better. Say that we change `Book`'s name to `Liber`, because we've fallen in love with Latin. We have to change the `__init__` method of `Ebook` to reflect the change. This can be avoided by using `super`.

`oop/super.implicit.py`

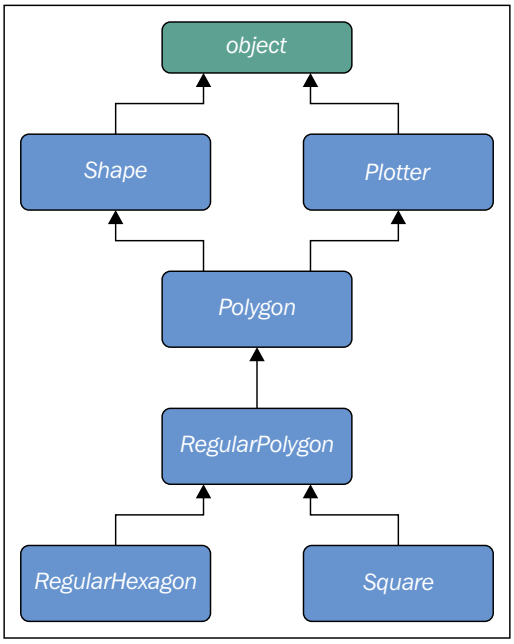
```
class Book:
    def __init__(self, title, publisher, pages):
        self.title = title
        self.publisher = publisher
        self.pages = pages

class Ebook(Book):
    def __init__(self, title, publisher, pages, format_):
        super().__init__(title, publisher, pages)
        # Another way to do the same thing is:
        # super(Ebook, self).__init__(title, publisher, pages)
        self.format_ = format_

ebook = Ebook('Learning Python', 'Packt Publishing', 360, 'PDF')
print(ebook.title)  # Learning Python
print(ebook.publisher)  # Packt Publishing
print(ebook.pages)  # 360
print(ebook.format_)  # PDF
```

`super` is a function that returns a proxy object that delegates method calls to a parent or sibling class. In this case, it will delegate that call to `__init__` to the `Book` class, and the beauty of this method is that now we're even free to change `Book` to `Liber` without having to touch the logic in the `__init__` method of `Ebook`.

Now that we know how to access a base class from a child, let's explore Python's multiple inheritance.

Multiple inheritance

Apart from composing a class using more than one base class, what is of interest here is how an attribute search is performed. Take a look at the following diagram:



As you can see, `Shape` and `Plotter` act as base classes for all the others. `Polygon` inherits directly from them, `RegularPolygon` inherits from `Polygon`, and both `RegularHexagon` and `Square` inherit from `RegulaPolygon`. Note also that `Shape` and `Plotter` implicitly inherit from `object`, therefore we have what is called a **diamond** or, in simpler terms, more than one path to reach a base class. We'll see why this matters in a few moments. Let's translate it into some simple code:

`oop/multiple.inheritance.py`

```
class Shape:
    geometric_type = 'Generic Shape'

    def area(self):  # This acts as placeholder for the interface
        raise NotImplementedError

    def get_geometric_type(self):
        return self.geometric_type

class Plotter:

    def plot(self, ratio, topleft):
        # Imagine some nice plotting logic here...
```

```
            print('Plotting at {}, ratio {}.'.format(
                topleft, ratio))

    class Polygon(Shape, Plotter):  # base class for polygons
        geometric_type = 'Polygon'

    class RegularPolygon(Polygon):  # Is-A Polygon
        geometric_type = 'Regular Polygon'

        def __init__(self, side):
            self.side = side

    class RegularHexagon(RegularPolygon): # Is-A RegularPolygon
        geometric_type = 'RegularHexagon'

        def area(self):
            return 1.5 * (3 ** .5 * self.side ** 2)

    class Square(RegularPolygon):  # Is-A RegularPolygon
        geometric_type = 'Square'

        def area(self):
            return self.side * self.side

    hexagon = RegularHexagon(10)
    print(hexagon.area())  # 259.8076211353316
    print(hexagon.get_geometric_type())  # RegularHexagon
    hexagon.plot(0.8, (75, 77))  # Plotting at (75, 77), ratio 0.8.

    square = Square(12)
    print(square.area())  # 144
    print(square.get_geometric_type())  # Square
    square.plot(0.93, (74, 75))  # Plotting at (74, 75), ratio 0.93.
```

Take a look at the preceding code: the class `Shape` has one attribute, `geometric_type`, and two methods: `area` and `get_geometric_type`. It's quite common to use base classes (like `Shape`, in our example) to define an *interface*: methods for which children must provide an implementation. There are different and better ways to do this, but I want to keep this example as simple as possible.

We also have the `Plotter` class, which adds the `plot` method, thereby providing plotting capabilities for any class that inherits from it. Of course, the `plot` implementation is just a dummy `print` in this example. The first interesting class is `Polygon`, which inherits from both `Shape` and `Plotter`.

There are many types of polygons, one of which is the regular one, which is both equiangular (all angles are equal) and equilateral (all sides are equal), so we create the `RegularPolygon` class that inherits from `Polygon`. Because for a regular polygon, all sides are equal, we can implement a simple `__init__` method on `RegularPolygon`, which takes the length of the side. Finally, we create the `RegularHexagon` and `Square` classes, which both inherit from `RegularPolygon`.

This structure is quite long, but hopefully gives you an idea of how to specialize the classification of your objects when you design the code.

Now, please take a look at the last eight lines. Note that when I call the `area` method on `hexagon` and `square`, I get the correct area for both. This is because they both provide the correct implementation logic for it. Also, I can call `get_geometric_type` on both of them, even though it is not defined on their classes, and Python has to go all the way up to `Shape` to find an implementation for it. Note that, even though the implementation is provided in the `Shape` class, the `self.geometric_type` used for the return value is correctly taken from the caller instance.

The `plot` method calls are also interesting, and show you how you can enrich your objects with capabilities they wouldn't otherwise have. This technique is very popular in web frameworks such as Django (which we'll explore in two later chapters), which provides special classes called **mixins**, whose capabilities you can just use out of the box. All you have to do is to define the desired mixin as one the base classes for your own, and that's it.

Multiple inheritance is powerful, but can also get really messy, so we need to make sure we understand what happens when we use it.

Method resolution order

By now, we know that when you ask for `someobject.attribute`, and `attribute` is not found on that object, Python starts searching in the class `someobject` was created from. If it's not there either, Python searches up the inheritance chain until either `attribute` is found or the `object` class is reached. This is quite simple to understand if the inheritance chain is only comprised of single inheritance steps, which means that classes have only one parent. However, when multiple inheritance is involved, there are cases when it's not straightforward to predict what will be the next class that will be searched for if an attribute is not found.

Python provides a way to always know what is the order in which classes are searched on attribute lookup: the method resolution order.

The **method resolution order** (**MRO**) is the order in which base classes are searched for a member during lookup. From version 2.3 Python uses an algorithm called **C3**, which guarantees monotonicity.

In Python 2.2, **new-style classes** were introduced. The way you write a new-style class in Python 2.* is to define it with an explicit `object` base class. Classic classes were not explicitly inheriting from `object` and have been removed in Python 3.

One of the differences between classic and new style-classes in Python 2.* is that new-style classes are searched with the new MRO.

With regards to the previous example, let's see what is the MRO for the `Square` class:

`oop/multiple.inheritance.py`

```
print(square.__class__.__mro__)
# prints:
# (<class '__main__.Square'>, <class '__main__.RegularPolygon'>,
#  <class '__main__.Polygon'>, <class '__main__.Shape'>,
#  <class '__main__.Plotter'>, <class 'object'>)
```

To get to the MRO of a class, we can go from the instance to its `__class__` attribute and from that to its `__mro__` attribute. Alternatively, we could have called `Square.__mro__`, or `Square.mro()` directly, but if you have to do it dynamically, it's more likely you will have an object in your hands rather than a class.

Note that the only point of doubt is the bisection after `Polygon`, where the inheritance chain breaks into two ways, one leads to `Shape` and the other to `Plotter`. We know by scanning the MRO for the `Square` class that `Shape` is searched before `Plotter`.

Why is this important? Well, imagine the following code:

`oop/mro.simple.py`

```
class A:
    label = 'a'

class B(A):
    label = 'b'

class C(A):
```

```
        label = 'c'

    class D(B, C):
        pass

    d = D()
    print(d.label)  # Hypothetically this could be either 'b' or 'c'
```

Both `B` and `C` inherit from `A`, and `D` inherits from both `B` and `C`. This means that the lookup for the `label` attribute can reach the top (`A`) through both `B` or `C`. According to which is reached first, we get a different result.

So, in the preceding example we get `'b'`, which is what we were expecting, since `B` is the leftmost one amongst base classes of `D`. But what happens if I remove the `label` attribute from `B`? This would be the confusing situation: Will the algorithm go all the way up to `A` or will it get to `C` first? Let's find out!

oop/mro.py

```
    class A:
        label = 'a'

    class B(A):
        pass  # was: label = 'b'

    class C(A):
        label = 'c'

    class D(B, C):
        pass

    d = D()
    print(d.label)  # 'c'
    print(d.__class__.mro())  # notice another way to get the MRO
    # prints:
    # [<class '__main__.D'>, <class '__main__.B'>,
    #  <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

So, we learn that the MRO is `D-B-C-A-(object)`, which means when we ask for `d.label`, we get `'c'`, which is correct.

In day to day programming, it is not quite common to have to deal with the MRO, but the first time you fight against some mixin from a framework, I promise you'll be glad I spent a paragraph explaining it.

Static and class methods

Until now, we have coded classes with attributes in the form of data and instance methods, but there are two other types of methods that we can place inside a class: **static methods** and **class methods**.

Static methods

As you may recall, when you create a class object, Python assigns a name to it. That name acts as a namespace, and sometimes it makes sense to group functionalities under it. Static methods are perfect for this use case since unlike instance methods, they are not passed any special argument. Let's look at an example of an imaginary `String` class.

oop/static.methods.py

```
class String:

    @staticmethod
    def is_palindrome(s, case_insensitive=True):
        # we allow only letters and numbers
        s = ''.join(c for c in s if c.isalnum())  # Study this!
        # For case insensitive comparison, we lower-case s
        if case_insensitive:
            s = s.lower()
        for c in range(len(s) // 2):
            if s[c] != s[-c -1]:
                return False
        return True

    @staticmethod
    def get_unique_words(sentence):
        return set(sentence.split())

print(String.is_palindrome(
    'Radar', case_insensitive=False))  # False: Case Sensitive
print(String.is_palindrome('A nut for a jar of tuna'))  # True
print(String.is_palindrome('Never Odd, Or Even!'))  # True
print(String.is_palindrome(
    'In Girum Imus Nocte Et Consumimur Igni')  # Latin! Show-off!
)  # True

print(String.get_unique_words(
    'I love palindromes. I really really love them!'))
# {'them!', 'really', 'palindromes.', 'I', 'love'}
```

The preceding code is quite interesting. First of all, we learn that static methods are created by simply applying the `staticmethod` decorator to them. You can see that they aren't passed any special argument so, apart from the decoration, they really just look like functions.

We have a class, `String`, which acts as a container for functions. Another approach would be to have a separate module with functions inside. It's really a matter of preference most of the time.

The logic inside `is_palindrome` should be straightforward for you to understand by now, but, just in case, let's go through it. First we remove all characters from `s` that are not either letters or numbers. In order to do this, we use the `join` method of a string object (an empty string object, in this case). By calling `join` on an empty string, the result is that all elements in the iterable you pass to `join` will be concatenated together. We feed `join` a generator expression that says, *take any character from s if the character is either alphanumeric or a number*. I hope you have been able to find that out by yourself, maybe using the inside-out technique I showed you in one of the preceding chapters.

We then lowercase `s` if `case_insensitive` is `True`, and then we proceed to check if it is a palindrome. In order to do this, we compare the first and last characters, then the second and the second to last, and so on. If at any point we find a difference, it means the string isn't a palindrome and therefore we can return `False`. On the other hand, if we exit the `for` loop normally, it means no differences were found, and we can therefore say the string is a palindrome.

Notice that this code works correctly regardless of the length of the string, that is, if the length is odd or even. `len(s) // 2` reaches half of `s`, and if `s` is an odd amount of characters long, the middle one won't be checked (like in *RaDaR*, *D* is not checked), but we don't care; it would be compared with itself so it's always passing that check.

`get_unique_words` is much simpler, it just returns a set to which we feed a list with the words from a sentence. The `set` class removes any duplication for us, therefore we don't need to do anything else.

The `String` class provides us a nice container namespace for methods that are meant to work on strings. I could have coded a similar example with a `Math` class, and some static methods to work on numbers, but I wanted to show you something different.

Class methods

Class methods are slightly different from instance methods in that they also take a special first argument, but in this case, it is the class object itself. Two very common use cases for coding class methods are to provide factory capability to a class and to allow breaking up static methods (which you have to then call using the class name) without having to hardcode the class name in your logic. Let's look at an example of both of them.

`oop/class.methods.factory.py`

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

    @classmethod
    def from_tuple(cls, coords):  # cls is Point
        return cls(*coords)

    @classmethod
    def from_point(cls, point):  # cls is Point
        return cls(point.x, point.y)

p = Point.from_tuple((3, 7))
print(p.x, p.y)  # 3 7
q = Point.from_point(p)
print(q.x, q.y)  # 3 7
```

In the preceding code, I showed you how to use a class method to create a factory for the class. In this case, we want to create a `Point` instance by passing both coordinates (regular creation `p = Point(3, 7)`), but we also want to be able to create an instance by passing a tuple (`Point.from_tuple`) or another instance (`Point.from_point`).

Within the two class methods, the `cls` argument refers to the `Point` class. As with instance method, which take `self` as the first argument, class method take a `cls` argument. Both `self` and `cls` are named after a convention that you are not forced to follow but are strongly encouraged to respect. This is something that no Python coder would change because it is so strong a convention that parsers, linters, and any tool that automatically does something with your code would expect, so it's much better to stick to it.

Let's look at an example of the other use case: splitting a static method.

oop/class.methods.split.py

```
class String:

    @classmethod
    def is_palindrome(cls, s, case_insensitive=True):
        s = cls._strip_string(s)
        # For case insensitive comparison, we lower-case s
        if case_insensitive:
            s = s.lower()
        return cls._is_palindrome(s)

    @staticmethod
    def _strip_string(s):
        return ''.join(c for c in s if c.isalnum())

    @staticmethod
    def _is_palindrome(s):
        for c in range(len(s) // 2):
            if s[c] != s[-c -1]:
                return False
        return True

    @staticmethod
    def get_unique_words(sentence):
        return set(sentence.split())

print(String.is_palindrome('A nut for a jar of tuna'))  # True
print(String.is_palindrome('A nut for a jar of beans'))  # False
```

Compare this code with the previous version. First of all note that even though `is_palindrome` is now a class method, we call it in the same way we were calling it when it was a static one. The reason why we changed it to a class method is that after factoring out a couple of pieces of logic (`_strip_string` and `_is_palindrome`), we need to get a reference to them and if we have no `cls` in our method, the only option would be to call them like this: `String._strip_string(...)` and `String._is_palindrome(...)`, which is not good practice, because we would hardcode the class name in the `is_palindrome` method, thereby putting ourselves in the condition of having to modify it whenever we would change the class name. Using `cls` will act as the class name, which means our code won't need any amendments.

Note also that, by naming the *factored-out* methods with a leading underscore, I am hinting that those methods are not supposed to be called from outside the class, but this will be the subject of the next paragraph.

Private methods and name mangling

If you have any background with languages like Java, C#, C++, or similar, then you know they allow the programmer to assign a privacy status to attributes (both data and methods). Each language has its own slightly different flavor for this, but the gist is that public attributes are accessible from any point in the code, while private ones are accessible only within the scope they are defined in.

In Python, there is no such thing. Everything is public; therefore, we rely on conventions and on a mechanism called **name mangling**.

The convention is as follows: if an attribute's name has no leading underscores it is considered public. This means you can access it and modify it freely. When the name has one leading underscore, the attribute is considered private, which means it's probably meant to be used internally and you should not use it or modify it from the outside. A very common use case for private attributes are helper methods that are supposed to be used by public ones (possibly in call chains in conjunction with other methods), and internal data, like scaling factors, or any other data that ideally we would put in a constant (a variable that cannot change, but, surprise, surprise, Python doesn't have those either).

This characteristic usually scares people from other backgrounds off; they feel threatened by the lack of privacy. To be honest, in my whole professional experience with Python, I've never heard anyone screaming *Oh my God, we have a terrible bug because Python lacks private attributes!* Not once, I swear.

That said, the call for privacy actually makes sense because without it, you risk introducing bugs into your code for real. Let's look at a simple example:

`oop/private.attrs.py`

```
class A:
    def __init__(self, factor):
        self._factor = factor

    def op1(self):
        print('Op1 with factor {}...'.format(self._factor))

class B(A):
    def op2(self, factor):
```

```
            self._factor = factor
            print('Op2 with factor {}...'.format(self._factor))


    obj = B(100)
    obj.op1()    # Op1 with factor 100...
    obj.op2(42)  # Op2 with factor 42...
    obj.op1()    # Op1 with factor 42...  <- This is BAD
```

In the preceding code, we have an attribute called `_factor`, and let's pretend it's very important that it isn't modified at runtime after the instance is created, because `op1` depends on it to function correctly. We've named it with a leading underscore, but the issue here is that when we call `obj.op2(42)`, we modify it, and this reflects in subsequent calls to `op1`.

Let's fix this undesired behavior by adding another leading underscore:

`oop/private.attrs.fixed.py`

```
    class A:
        def __init__(self, factor):
            self.__factor = factor

        def op1(self):
            print('Op1 with factor {}...'.format(self.__factor))

    class B(A):
        def op2(self, factor):
            self.__factor = factor
            print('Op2 with factor {}...'.format(self.__factor))


    obj = B(100)
    obj.op1()    # Op1 with factor 100...
    obj.op2(42)  # Op2 with factor 42...
    obj.op1()    # Op1 with factor 100...  <- Wohoo! Now it's GOOD!
```

Wow, look at that! Now it's working as desired. Python is kind of magic and in this case, what is happening is that the name mangling mechanism has kicked in.

Name mangling means that any attribute name that has at least two leading underscores and at most one trailing underscore, like `__my_attr`, is replaced with a name that includes an underscore and the class name before the actual name, like `_ClassName__my_attr`.

This means that when you inherit from a class, the mangling mechanism gives your private attribute two different names in the base and child classes so that name collision is avoided. Every class and instance object stores references to their attributes in a special attribute called `__dict__`, so let's inspect `obj.__dict__` to see name mangling in action:

`oop/private.attrs.py`

```
print(obj.__dict__.keys())
# dict_keys(['_factor'])
```

This is the `_factor` attribute that we find in the problematic version of this example. But look at the one that is using `__factor`:

`oop/private.attrs.fixed.py`

```
print(obj.__dict__.keys())
# dict_keys(['_A__factor', '_B__factor'])
```

See? `obj` has two attributes now, `_A__factor` (mangled within the `A` class), and `_B__factor` (mangled within the `B` class). This is the mechanism that makes possible that when you do `obj.__factor = 42`, `__factor` in `A` isn't changed, because you're actually touching `_B__factor`, which leaves `_A__factor` safe and sound.

If you're designing a library with classes that are meant to be used and extended by other developers, you will need to keep this in mind in order to avoid unintentional overriding of your attributes. Bugs like these can be pretty subtle and hard to spot.

The property decorator

Another thing that would be a crime not to mention is the `property` decorator. Imagine that you have an `age` attribute in a `Person` class and at some point you want to make sure that when you change its value, you're also checking that `age` is within a proper range, like [18, 99]. You can write accessor methods, like `get_age()` and `set_age()` (also called **getters** and **setters**) and put the logic there. `get_age()` will most likely just return `age`, while `set_age()` will also do the range check. The problem is that you may already have a lot of code accessing the `age` attribute directly, which means you're now up to some good (and potentially dangerous and tedious) refactoring. Languages like Java overcome this problem by using the accessor pattern basically by default. Many Java **Integrated Development Environments** (**IDEs**) autocomplete an attribute declaration by writing getter and setter accessor methods stubs for you on the fly.

Python is smarter, and does this with the `property` decorator. When you decorate a method with `property`, you can use the name of the method as if it was a data attribute. Because of this, it's always best to refrain from putting logic that would take a while to complete in such methods because, by accessing them as attributes, we are not expecting to wait.

Let's look at an example:

oop/property.py

```
class Person:
    def __init__(self, age):
        self.age = age  # anyone can modify this freely

class PersonWithAccessors:
    def __init__(self, age):
        self._age = age

    def get_age(self):
        return self._age

    def set_age(self):
        if 18 <= age <= 99:
            self._age = age
        else:
            raise ValueError('Age must be within [18, 99]')

class PersonPythonic:
    def __init__(self, age):
        self._age = age

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, age):
        if 18 <= age <= 99:
            self._age = age
        else:
            raise ValueError('Age must be within [18, 99]')

person = PersonPythonic(39)
```

```
print(person.age)  # 39 - Notice we access as data attribute
person.age = 42  # Notice we access as data attribute
print(person.age)  # 42
person.age = 100  # ValueError: Age must be within [18, 99]
```

The `Person` class may be the first version we write. Then we realize we need to put the range logic in place so, with another language, we would have to rewrite `Person` as the `PersonWithAccessors` class, and refactor all the code that was using `Person.age`. In Python, we rewrite `Person` as `PersonPythonic` (you normally wouldn't change the name, of course) so that the age is stored in a private `_age` variable, and we define property getters and setters using that decoration, which allow us to keep using the `person` instances as we were before. A **getter** is a method that is called when we access an attribute for reading. On the other hand, a **setter** is a method that is called when we access an attribute to write it. In other languages, like Java for example, it's customary to define them as `get_age()` and `set_age(int value)`, but I find the Python syntax much neater. It allows you to start writing simple code and refactor later on, only when you need it, there is no need to pollute your code with accessors only because they may be helpful in the future.

The `property` decorator also allows for read-only data (no setter) and for special actions when the attribute is deleted. Please refer to the official documentation to dig deeper.

Operator overloading

I find Python's approach to **operator overloading** to be brilliant. To overload an operator means to give it a meaning according to the context in which it is used. For example, the `+` operator means addition when we deal with numbers, but concatenation when we deal with sequences.

In Python, when you use operators, you're most likely calling the special methods of some objects behind the scenes. For example, the call `a[k]` roughly translates to `type(a).__getitem__(a, k)`.

As an example, let's create a class that stores a string and evaluates to `True` if `'42'` is part of that string, and `False` otherwise. Also, let's give the class a length property which corresponds to that of the stored string.

`oop/operator.overloading.py`

```
class Weird:
    def __init__(self, s):
        self._s = s

    def __len__(self):
```

```
            return len(self._s)

        def __bool__(self):
            return '42' in self._s


    weird = Weird('Hello! I am 9 years old!')
    print(len(weird))  # 24
    print(bool(weird))  # False

    weird2 = Weird('Hello! I am 42 years old!')
    print(len(weird2))  # 25
    print(bool(weird2))  # True
```

That was fun, wasn't it? For the complete list of magic methods that you can override in order to provide your custom implementation of operators for your classes, please refer to the Python data model in the official documentation.

Polymorphism – a brief overview

The word **polymorphism** comes from the Greek *polys* (many, much) and *morphē* (form, shape), and its meaning is the provision of a single interface for entities of different types.

In our car example, we call `engine.start()`, regardless of what kind of engine it is. As long as it exposes the start method, we can call it. That's polymorphism in action.

In other languages, like Java, in order to give a function the ability to accept different types and call a method on them, those types need to be coded in such a way that they share an interface. In this way, the compiler knows that the method will be available regardless of the type of the object the function is fed (as long as it extends the proper interface, of course).

In Python, things are different. Polymorphism is implicit, nothing prevents you to call a method on an object, therefore, technically, there is no need to implement interfaces or other patterns.

There is a special kind of polymorphism called **ad hoc polymorphism**, which is what we saw in the last paragraph: operator overloading. The ability of an operator to change shape, according to the type of data it is fed.

I cannot spend too much time on polymorphism, but I encourage you to check it out by yourself, it will expand your understanding of OOP. Good luck!

Writing a custom iterator

Now we have all the tools to appreciate how we can write our own custom iterator. Let's first define what is an iterable and an iterator:

- **Iterable**: An object is said to be iterable if it's capable of returning its members one at a time. Lists, tuples, strings, dicts, are all iterables. Custom objects that define either of `__iter__` or `__getitem__` methods are also iterables.
- **Iterator**: An object is said to be an iterator if it represents a stream of data. A custom iterator is required to provide an implementation for `__iter__` that returns the object itself, and an implementation for `__next__`, which returns the next item of the data stream until the stream is exhausted, at which point all successive calls to `__next__` simply raise the `StopIteration` exception. Built-in functions such as `iter` and `next` are mapped to call `__iter__` and `__next__` on an object, behind the scenes.

Let's write an iterator that returns all the odd characters from a string first, and then the even ones.

`iterators/iterator.py`

```
class OddEven:

    def __init__(self, data):
        self._data = data
        self.indexes = (list(range(0, len(data), 2)) +
            list(range(1, len(data), 2)))

    def __iter__(self):
        return self

    def __next__(self):
        if self.indexes:
            return self._data[self.indexes.pop(0)]
        raise StopIteration

oddeven = OddEven('ThIsIsCoOl!')
print(''.join(c for c in oddeven))  # TIICO!hssol

oddeven = OddEven('HoLa')  # or manually...
it = iter(oddeven)  # this calls oddeven.__iter__ internally
```

```
print(next(it))  # H
print(next(it))  # L
print(next(it))  # o
print(next(it))  # a
```

So, we needed to provide an implementation for `__iter__` which returned the object itself, and then one for `__next__`. Let's go through it. What needs to happen is that we return `_data[0]`, `_data[2]`, `_data[4]`, ..., `_data[1]`, `_data[3]`, `_data[5]`, ... until we have returned every item in the data. In order to do this, we prepare a list, indexes, like [0, 2, 4, 6, ..., 1, 3, 5, ...], and while there is at least an element in it, we pop the first one and return the element from the data that is at that position, thereby achieving our goal. When `indexes` is empty, we raise `StopIteration`, as required by the iterator protocol.

There are other ways to achieve the same result, so go ahead and try to code a different one yourself. Make sure the end result works for all edge cases, empty sequences, sequences of length 1, 2, and so on.

Summary

In this chapter, we saw decorators, discovered the reasons for having them, and a few examples using one or more at the same time. We also saw decorators that take arguments, which are usually used as decorator factories.

We scratched the surface of object-oriented programming in Python. We covered all the basics in a way that you should now be able to understand fairly easily the code that will come in future chapters. We talked about all kinds of methods and attributes that one can write in a class, we explored inheritance versus composition, method overriding, properties, operator overloading, and polymorphism.

At the end, we very briefly touched base on iterators, so now you have all the knowledge to also understand generators more deeply.

In the next chapter, we take a steep turn. It will start the second half of the book, which is much more project-oriented so, from now on, it will be less theory and more code, I hope you will enjoy following the examples and getting your hands dirty, very dirty.

They say that a smooth sea never made a skillful sailor, so keep exploring, break things, read the error messages as well as the documentation, and let's see if we can get to see that white rabbit.

7
Testing, Profiling, and Dealing with Exceptions

"*Code without tests is broken by design.*"

- Jacob Kaplan-Moss

Jacob Kaplan-Moss is one of the core developers of the Django web framework. We're going to explore it in the next chapters. I strongly agree with this quote of his. I believe code without tests shouldn't be deployed to production.

Why are tests so important? Well, for one, they give you predictability. Or, at least, they help you achieve high predictability. Unfortunately, there is always some bug that sneaks into our code. But we definitely want our code to be as predictable as possible. What we don't want is to have a surprise, our code behaving in an unpredictable way. Would you be happy to know that the software that checks on the sensors of the plane that is taking you on holidays sometimes goes crazy? No, probably not.

Therefore we need to test our code, we need to check that its behavior is correct, that it works as expected when it deals with edge cases, that it doesn't hang when the components it's talking to are down, that the performances are well within the acceptable range, and so on.

This chapter is all about this topic, making sure that your code is prepared to face the scary outside world, that is fast enough and that it can deal with unexpected or exceptional conditions.

We're going to explore testing, including a brief introduction to **test-driven development** (**TDD**), which is one of my favorite working methodologies. Then, we're going to explore the world of exceptions, and finally we're going to talk a little bit about performances and profiling. Deep breath, and here we go...

Testing your application

There are many different kinds of tests, so many in fact that companies often have a dedicated department, called **quality assurance** (**QA**), made up of individuals that spend their day testing the software the company developers produce.

To start making an initial classification, we can divide tests into two broad categories: white-box and black-box tests.

White-box tests are those which exercise the internals of the code, they inspect it down to a very fine level of granularity. On the other hand, **black-box tests** are those which consider the software under testing as if being within a box, the internals of which are ignored. Even the technology, or the language used inside the box is not important for black-box tests. What they do is to plug input to one end of the box and verify the output at the other end, and that's it.

There is also an in-between category, called **gray-box** testing, that involves testing a system in the same way we do with the black-box approach, but having some knowledge about the algorithms and data structures used to write the software and only partial access to its source code.

There are many different kinds of tests in these categories, each of which serves a different purpose. Just to give you an idea, here's a few:

- **Front-end tests** make sure that the client side of your application is exposing the information that it should, all the links, the buttons, the advertising, everything that needs to be shown to the client. It may also verify that it is possible to walk a certain path through the user interface.
- **Scenario tests** make use of stories (or scenarios) that help the tester work through a complex problem or test a part of the system.
- **Integration tests** verify the behavior of the various components of your application when they are working together sending messages through interfaces.

- **Smoke tests** are particularly useful when you deploy a new update on your application. They check whether the most essential, vital parts of your application are still working as they should and that they are not *on fire*. This term comes from when engineers tested circuits by making sure nothing was smoking.
- **Acceptance tests**, or **user acceptance testing** (**UAT**) is what a developer does with a product owner (for example, in a SCRUM environment) to determine if the work that was commissioned was carried out correctly.
- **Functional tests** verify the features or functionalities of your software.
- **Destructive tests** take down parts of your system, simulating a failure, in order to establish how well the remaining parts of the system perform. These kinds of tests are performed extensively by companies that need to provide an extremely reliable service, such as Amazon, for example.
- **Performance tests** aim to verify how well the system performs under a specific load of data or traffic so that, for example, engineers can get a better understanding of which are the bottlenecks in the system that could bring it down to its knees in a heavy load situation, or those which prevent scalability.
- **Usability tests**, and the closely related **user experience** (**UX**) tests, aim to check if the user interface is simple and easy to understand and use. They aim to provide input to the designers so that the user experience is improved.
- **Security and penetration tests** aim to verify how well the system is protected against attacks and intrusions.
- **Unit tests** help the developer to write the code in a robust and consistent way, providing the first line of feedback and defense against coding mistakes, refactoring mistakes, and so on.
- **Regression tests** provide the developer with useful information about a feature being compromised in the system after an update. Some of the causes for a system being said to have a regression are an old bug coming back to life, an existing feature being compromised, or a new issue being introduced.

Many books and articles have been written about testing, and I have to point you to those resources if you're interested in finding out more about all the different kinds of tests. In this chapter, we will concentrate on unit tests, since they are the backbone of software crafting and form the vast majority of tests that are written by a developer.

Testing is an *art*, an art that you don't learn from books, I'm afraid. You can learn all the definitions (and you should), and try and collect as much knowledge about testing as you can but I promise you, you will be able to test your software properly only when you have done it for long enough in the field.

When you are having trouble refactoring a bit of code, because every little thing you touch makes a test blow up, you learn how to write less rigid and limiting tests, which still verify the correctness of your code but, at the same time, allow you the freedom and joy to play with it, to shape it as you want.

When you are being called too often to fix unexpected bugs in your code, you learn how to write tests more thoroughly, how to come up with a more comprehensive list of edge cases, and strategies to cope with them before they turn into bugs.

When you are spending too much time reading tests and trying to refactor them in order to change a small feature in the code, you learn to write simpler, shorter, and better focused tests.

I could go on with this *when you... you learn...*, but I guess you get the picture. You need to get your hands dirty and build experience. My suggestion? Study the theory as much as you can, and then experiment using different approaches. Also, try to learn from experienced coders; it's very effective.

The anatomy of a test

Before we concentrate on unit tests, let's see what a test is, and what its purpose is.

A **test** is a piece of code whose purpose is to verify something in our system. It may be that we're calling a function passing two integers, that an object has a property called `donald_duck`, or that when you place an order on some API, after a minute you can see it dissected into its basic elements, in the database.

A test is typically comprised of three sections:

- **Preparation**: This is where you set up the scene. You prepare all the data, the objects, the services you need in the places you need them so that they are ready to be used.
- **Execution**: This is where you execute the bit of logic that you're checking against. You perform an action using the data and the interfaces you have set up in the preparation phase.
- **Verification**: This is where you verify the results and make sure they are according to your expectations. You check the returned value of a function, or that some data is in the database, some is not, some has changed, a request has been made, something has happened, a method has been called, and so on.

Testing guidelines

Like software, tests can be good or bad, with the whole range of shades in the middle. In order to write good tests, here are some guidelines:

- **Keep them as simple as possible**: It's okay to violate some good coding rules, such as hardcoding values or duplicating code. Tests need first and foremost to be as readable as possible and easy to understand. When tests are hard to read or understand, you can never be sure if they are actually making sure your code is performing correctly.
- **Tests should verify one thing and one thing only**: It's very important that you keep them short and contained. It's perfectly fine to write multiple tests to exercise a single object or function. Just make sure that each test has one and only one purpose.
- **Tests should not make any unnecessary assumption when verifying data**: This is tricky to understand at first, but say you are testing the return value of a function and it is an unordered list of numbers (like `[2, 3, 1]`). If the order in that list is random, in the test you may be tempted to sort it and compare it with `[1, 2, 3]`. If you do, you will introduce an extra assumption on the ordering of the result of your function call, and *this is bad practice*. You should always find a way to verify things without introducing any assumptions or any feature that doesn't belong in the use case you're describing with your test.
- **Tests should exercise the what, rather than the how**: Tests should focus on checking *what* a function is supposed to do, rather than *how* it is doing it. For example, focus on the fact that it's calculating the square root of a number (the *what*), instead of on the fact that it is calling `math.sqrt` to do it (the *how*). Unless you're writing performance tests or you have a particular need to verify how a certain action is performed, try to avoid this type of testing and focus on the *what*. Testing the *how* leads to restrictive tests and makes refactoring hard. Moreover, the type of test you have to write when you concentrate on the *how* is more likely to degrade the quality of your testing code base when you amend your software frequently (more on this later).

- **Tests should assume the least possible in the preparation phase**: Say you have 10 tests that are checking how a data structure is manipulated by a function. And let's say this data structure is a dict with five key/value pairs. If you put the complete dict in each test, the moment you have to change something in that dict, you also have to amend all ten tests. On the other hand, if you strip down the test data as much as you can, you will find that, most of the time, it's possible to have the majority of tests checking only a partial version of the data, and only a few running with a full version of it. This means that when you need to change your data, you will have to amend only those tests that are actually exercising it.
- **Test should run as fast as possible**: A good test codebase could end up being much longer than the code being tested itself. It varies according to the situation and the developer but whatever the length, you'll end up having hundreds, if not thousands, of tests to run, which means the faster they run, the faster you can get back to writing code. When using TDD, for example, you run tests very often, so speed is essential.
- **Tests should use up the least possible amount of resources**: The reason for this is that every developer who checks out your code should be able to run your tests, no matter how powerful their box is. It could be a skinny virtual machine or a neglected Jenkins box, your tests should run without chewing up too many resources.

A **Jenkins** box is a machine that runs Jenkins, software that is capable of, amongst many other things, running your tests automatically. Jenkins is frequently used in companies where developers use practices like continuous integration, extreme programming, and so on.

Unit testing

Now that you have an idea about what testing is and why we need it, let's finally introduce the developer's best friend: the **unit test**.

Before we proceed with the examples, allow me to spend some words of caution: I'll try to give you the fundamentals about unit testing, but I don't follow any particular school of thought or methodology to the letter. Over the years, I have tried many different testing approaches, eventually coming up with my own way of doing things, which is constantly evolving. To put it as Bruce Lee would have:

> *"Absorb what is useful, discard what is useless and add what is specifically your own".*

Writing a unit test

In order to explain how to write a unit test, let's help ourselves with a simple snippet:

`data.py`

```
    def get_clean_data(source):
        data = load_data(source)
        cleaned_data = clean_data(data)
        return cleaned_data
```

The function `get_clean_data` is responsible for getting data from `source`, cleaning it, and returning it to the caller. How do we test this function?

One way of doing this is to call it and then make sure that `load_data` was called once with `source` as its only argument. Then we have to verify that `clean_data` was called once, with the return value of `load_data`. And, finally, we would need to make sure that the return value of `clean_data` is what is returned by the `get_clean_data` function as well.

In order to do this, we need to set up the source and run this code, and this may be a problem. One of the golden rules of unit testing is that *anything that crosses the boundaries of your application needs to be simulated*. We don't want to talk to a real data source, and we don't want to actually run real functions if they are communicating with anything that is not contained in our application. A few examples would be a database, a search service, an external API, a file in the filesystem, and so on.

We need these restrictions to act as a shield, so that we can always run our tests safely without the fear of destroying something in a real data source.

Another reason is that it may be quite difficult for a single developer to reproduce the whole architecture on their box. It may require the setting up of databases, APIs, services, files and folders, and so on and so forth, and this can be difficult, time consuming, or sometimes not even possible.

Very simply put, an **application programming interface** (**API**) is a set of tools for building software applications. An API expresses a software component in terms of its operations, inputs and outputs, and underlying types. For example, if you create a software that needs to interface with a data provider service, it's very likely that you will have to go through their API in order to gain access to the data.

Therefore, in our unit tests, we need to simulate all those things in some way. Unit tests need to be run by any developer without the need for the whole system to be set up on their box.

A different approach, which I always favor when it's possible to do so, is to simulate entities without using fake objects, but using special purpose test objects instead. For example, if your code talks to a database, instead of faking all the functions and methods that talk to the database and programming the fake objects so that they return what the real ones would, I'd much rather prefer to spawn a test database, set up the tables and data I need, and then patch the connection settings so that my tests are running real code, against the test database, thereby doing no harm at all. In-memory databases are excellent options for these cases.

One of the applications that allow you to spawn a database for testing, is Django. Within the `django.test` package you can find several tools that help you write your tests so that you won't have to simulate the dialog with a database. By writing tests this way, you will also be able to check on transactions, encodings, and all other database related aspects of programming. Another advantage of this approach consists in the ability of checking against things that can change from one database to another.

Sometimes, though, it's still not possible, and we need to use fakes, therefore let's talk about them.

Mock objects and patching

First of all, in Python, these fake objects are called **mocks**. Up to version 3.3, the `mock` library was a third-party library that basically every project would install via `pip` but, from version 3.3, it has been included in the standard library under the `unittest` module, and rightfully so, given its importance and how widespread it is.

The act of replacing a real object or function (or in general, any piece of data structure) with a mock, is called **patching**. The `mock` library provides the `patch` tool, which can act as a function or class decorator, and even as a context manager (more on this in *Chapter 8*, *The Edges – GUIs and Scripts*), that you can use to mock things out. Once you have replaced everything you need not to run, with suitable mocks, you can pass to the second phase of the test and run the code you are exercising. After the execution, you will be able to check those mocks to verify that your code has worked correctly.

Assertions

The verification phase is done through the use of assertions. An **assertion** is a function (or method) that you can use to verify equality between objects, as well as other conditions. When a condition is not met, the assertion will raise an exception that will make your test fail. You can find a list of assertions in the `unittest` module documentation, and their corresponding Pythonic version in the nose third-party library, which provides a few advantages over the sheer `unittest` module, starting from an improved test discovery strategy (which is the way a test runner detects and discovers the tests in your application).

A classic unit test example

Mocks, patches, and assertions are the basic tools we'll be using to write tests. So, finally, let's see an example. I'm going to write a function that takes a list of integers and filters out all those which aren't positive.

`filter_funcs.py`

```
def filter_ints(v):
    return [num for num in v if is_positive(num)]

def is_positive(n):
    return n > 0
```

In the preceding example, we define the `filter_ints` function, which basically uses a list comprehension to retain all the numbers in `v` that are positive, discarding zeros and negative ones. I hope, by now, any further explanation of the code would be insulting.

What is interesting, though, is to start thinking about how we can test it. Well, how about we call `filter_ints` with a list of numbers and we make sure that `is_positive` is called for each of them? Of course, we would have to test `is_positive` as well, but I will show you later on how to do that. Let's write a simple test for `filter_ints` now.

Just to be sure we're on the same page, I am putting the code for this chapter in a folder called `ch7`, which lies within the root of our project. At the same level of `ch7`, I have created a folder called `tests`, in which I have placed a folder called `test_ch7`. In this folder I have one test file, called `test_filter_func.py`.

Basically, within the `tests` folder, I will recreate the tree structure of the code I'm testing, prepending everything with `test_`. This way, finding tests is really easy, as well as is keeping them tidy.

tests/test_ch7/test_filter_funcs.py

```
from unittest import TestCase  # 1
from unittest.mock import patch, call  # 2
from nose.tools import assert_equal  # 3
from ch7.filter_funcs import filter_ints  # 4

class FilterIntsTestCase(TestCase):  # 5

    @patch('ch7.filter_funcs.is_positive')  # 6
    def test_filter_ints(self, is_positive_mock):  # 7
        # preparation
        v = [3, -4, 0, 5, 8]

        # execution
        filter_ints(v)  # 8

        # verification
        assert_equal(
            [call(3), call(-4), call(0), call(5), call(8)],
            is_positive_mock.call_args_list
        )  # 9
```

My, oh my, so little code, and yet so much to say. First of all: `#1`. The `TestCase` class is the base class that we use to have a contained entity in which to run our tests. It's not just a bare container; it provides you with methods to write tests more easily.

On `#2`, we import `patch` and `call` from the `unittest.mock` module. `patch` is responsible for substituting an object with a `Mock` instance, thereby giving us the ability to check on it after the execution phase has been completed. `call` provides us with a nice way of expressing a (for example, function) call.

On `#3`, you can see that I prefer to use assertions from `nose`, rather than the ones that come with the `unittest` module. To give you an example, `assert_equal(...)` would become `self.assertEqual(...)` if I didn't use `nose`. I don't enjoy typing `self.` for any assertion, if there is a way to avoid it, and I don't particularly enjoy **camel case**, therefore I always prefer to use `nose` to make my assertions.

`assert_equal` is a function that takes two parameters (and an optional third one that acts as a message) and verifies that they are the same. If they are equal, nothing happens, but if they differ, then an `AssertionError` exception is raised, telling us something is wrong. When I write my tests, I always put the expected value as the first argument, and the real one as the second. This convention saves me time when I'm reading tests.

On #4, we import the function we want to test, and then (#5) we proceed to create the class where our tests will live. Each method of this class starting with `test_`, will be interpreted as a test. As you can see, we need to decorate `test_filter_ints` with `patch` (#6). Understanding this part is crucial, we need to patch the object where it is actually used. In this case, the path is very simple: `ch7.filter_func.is_positive`.

Patching can be very tricky, so I urge you to read the *Where to patch* section in the mock documentation: `https://docs.python.org/3/library/unittest.mock.html#where-to-patch`.

When we decorate a function using `patch`, like in our example, we get an extra argument in the test signature (#7), which I like to call as the patched function name, plus a `_mock` suffix, just to make it clear that the object has been patched (or mocked out).).

Finally, we get to the body of the test, and we have a very simple preparation phase in which we set up a list with at least one representative of all the integer number categories (negative, zero, and positive).

Then, in #8, we perform the execution phase, which runs the `filter_ints` function, without collecting its results. If all has gone as expected, the fake `is_positive` function must have been called with each of the integers in `v`.

We can verify this by comparing a list of call objects to the `call_args_list` attribute on the mock (#9). This attribute is the list of all the calls performed on the object since its creation.

Let's run this test. First of all, make sure that you install `nose` (`$ pip freeze` will tell you if you have it already):

```
$ pip install nose
```

Then, change into the root of the project (mine is called `learning.python`), and run the tests like this:

```
$ nosetests tests/test_ch7/
.
----------------------------------------------------------------
Ran 1 test in 0.006s
OK
```

The output shows one dot (each dot is a test), a separation line, and the time taken to run the whole test suite. It also says `OK` at the end, which means that our tests were all successful.

Making a test fail

Good, so just for fun let's make one fail. In the test file, change the last call from `call(8)` to `call(9)`, and run the tests again:

```
$ nosetests tests/test_ch7/
F
======================================================================
FAIL: test_filter_ints (test_filter_funcs.FilterIntsTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/usr/lib/python3.4/unittest/mock.py", line 1125, in patched
    return func(*args, **keywargs)
  File "/home/fab/srv/learning.python/tests/test_ch7/test_filter_funcs.
py", line 21, in test_filter_ints
    is_positive_mock.call_args_list
AssertionError: [call(3), call(-4), call(0), call(5), call(9)] !=
[call(3), call(-4), call(0), call(5), call(8)]
----------------------------------------------------------------------
Ran 1 test in 0.008s
FAILED (failures=1)
```

Wow, we made the beast angry! So much wonderful information, though. This tells you that the test `test_filter_ints` (with the path to it), was run and that it failed (the big `F` at the top, where the dot was before). It gives you a `Traceback`, that tells you that in the `test_filter_funcs.py` module, at line 21, when asserting on `is_positive_mock.call_args_list`, we have a discrepancy. The test expects the list of calls to end with a `call(9)` instance, but the real list ends with a `call(8)`. This is nothing less than wonderful.

If you have a test like this, can you imagine what would happen if you refactored and introduced a bug into your function by mistake? Well, your tests will break! They will tell you that *you have screwed something up, and here's the details*. So, you go and check out what you broke.

Interface testing

Let's add another test that checks on the returned value. It's another method in the class, so I won't reproduce the whole code again:

tests/test_ch7/test_filter_funcs.py

```
    def test_filter_ints_return_value(self):
        v = [3, -4, 0, -2, 5, 0, 8, -1]

        result = filter_ints(v)

        assert_list_equal([3, 5, 8], result)
```

This test is a bit different from the previous one. Firstly, we cannot mock the `is_positive` function, otherwise we wouldn't be able to check on the result. Secondly, we don't check on calls, but only on the result of the function when input is given.

I like this test much more than the previous one. This type of test is called an **interface test** because it checks on the interface (the set of inputs and outputs) of the function we're testing. It doesn't use any mocks, which is why I use this technique much more than the previous one. Let's run the new test suite and then let's see why I like interface testing more than those with mocks.

```
$ nosetests tests/test_ch7/
..
----------------------------------------------------------------
Ran 2 tests in 0.006s
OK
```

Two tests ran, all good (I changed that `9` back to an `8` in the first test, of course).

Comparing tests with and without mocks

Now, let's see why I don't really like mocks and use them only when I have no choice. Let's refactor the code in this way:

filter_funcs_refactored.py

```
    def filter_ints(v):
        v = [num for num in v if num != 0]  # 1
        return [num for num in v if is_positive(num)]
```

The code for `is_positive` is the same as before. But the logic in `filter_ints` has now changed in a way that `is_positive` will never be called with a `0`, since they are all filtered out in `#1`. This leads to an interesting result, so let's run the tests again:

```
$ nosetests tests/test_ch7/test_filter_funcs_refactored.py
F.
======================================================================
FAIL: test_filter_ints (test_filter_funcs_refactored.FilterIntsTestCase)
----------------------------------------------------------------------
... omit ...
AssertionError: [call(3), call(-4), call(0), call(5), call(8)] !=
[call(3), call(-4), call(5), call(8)]
----------------------------------------------------------------------
Ran 2 tests in 0.002s
FAILED (failures=1)
```

One test succeeded but the other one, the one with the mocked `is_positive` function, failed. The `AssertionError` message shows us that we now need to amend the list of expected calls, removing `call(0)`, because it is no longer performed.

This is not good. We have changed neither the interface of the function nor its behavior. The function is still keeping to its *original contract*. What we've done by testing it with a mocked object is limit ourselves. In fact, we now have to amend the test in order to use the new logic.

This is just a simple example but it shows one important flaw in the whole mock mechanism. *You must keep your mocks up-to-date and in sync with the code they are replacing*, otherwise you risk having issues like the preceding one, or even worse. Your tests may not fail because they are using mocked objects that perform fine, but because the real ones, now not in sync any more, are actually failing.

So *use mocks only when necessary*, only when there is no other way of testing your functions. When you cross the boundaries of your application in a test, try to use a replacement, like a test database, or a fake API, and only when it's not possible, resort to mocks. They are very powerful, but also very dangerous when not handled properly.

So, let's remove that first test and keep only the second one, so that I can show you another issue you could run into when writing tests. The whole test module now looks like this:

tests/test_ch7/test_filter_funcs_final.py

```
from unittest import TestCase
from nose.tools import assert_list_equal
from ch7.filter_funcs import filter_ints

class FilterIntsTestCase(TestCase):
    def test_filter_ints_return_value(self):
        v = [3, -4, 0, -2, 5, 0, 8, -1]
        result = filter_ints(v)
        assert_list_equal([3, 5, 8], result)
```

If we run it, it will pass.

A brief chat about triangulation. Now let me ask you: what happens if I change my `filter_ints` function to this?

filter_funcs_triangulation.py

```
def filter_ints(v):
    return [3, 5, 8]
```

If you run the test suite, the test we have will still pass! You may think I'm crazy but I'm showing you this because I want to talk about a concept called **triangulation**, which is very important when doing interface testing with TDD.

The whole idea is to remove cheating code, or badly performing code, by pinpointing it from different angles (like going to one vertex of a triangle from the other two) in a way that makes it impossible for our code to cheat, and the bug is exposed. We can simply modify the test like this:

tests/test_ch7/test_filter_funcs_final_triangulation.py

```
def test_filter_ints_return_value(self):
    v1 = [3, -4, 0, -2, 5, 0, 8, -1]
    v2 = [7, -3, 0, 0, 9, 1]

    assert_list_equal([3, 5, 8], filter_ints(v1))
    assert_list_equal([7, 9, 1], filter_ints(v2))
```

I have moved the execution section in the assertions directly, and you can see that we're now pinpointing our function from two different angles, thereby requiring that the real code be in it. It's no longer possible for our function to cheat.

Triangulation is a very powerful technique that teaches us to always try to exercise our code from many different angles, to cover all possible edge cases to expose any problems.

Boundaries and granularity

Let's now add a test for the `is_positive` function. I know it's a one-liner, but it presents us with opportunity to discuss two very important concepts: **boundaries** and **granularity**.

That `0` in the body of the function is a **boundary**, the `>` in the inequality is how we behave with regards to this boundary. Typically, when you set a boundary, you divide the space into three areas: what lies before the boundary, after the boundary, and on the boundary itself. In the example, before the boundary we find the negative numbers, the boundary is the element `0` and, after the boundary, we find the positive numbers. We need to test each of these areas to be sure we're testing the function correctly. So, let's see one possible solution (I will add the test to the class, but I won't show the repeated code):

`tests/test_ch7/test_filter_funcs_is_positive_loose.py`

```
    def test_is_positive(self):
        assert_equal(False, is_positive(-2))  # before boundary
        assert_equal(False, is_positive(0))  # on the boundary
        assert_equal(True, is_positive(2))  # after the boundary
```

You can see that we are exercising one number for each different area around the boundary. Do you think this test is good? Think about it for a minute before reading on.

The answer is no, this test is not good. Not good enough, anyway. If I change the body of the `is_positive` function to read `return n > 1`, I would expect my test to fail, but it won't. `-2` is still `False`, as well as `0`, and `2` is still `True`. Why does that happen? It is because we haven't taken granularity properly into account. We're dealing with integers, so what is the minimum granularity when we move from one integer to the next one? It's 1. Therefore, when we surround the boundary, taking all three areas into account is not enough. We need to do it with the minimum possible granularity. Let's change the test:

tests/test_ch7/test_filter_funcs_is_positive_correct.py

```
    def test_is_positive(self):
        assert_equal(False, is_positive(-1))
        assert_equal(False, is_positive(0))
        assert_equal(True, is_positive(1))
```

Ah, now it's better. Now if we change the body of `is_positive` to read `return n > 1`, the third assertion will fail, which is what we want. Can you think of a better test?

tests/test_ch7/test_filter_funcs_is_positive_better.py

```
    def test_is_positive(self):
        assert_equal(False, is_positive(0))
        for n in range(1, 10 ** 4):
            assert_equal(False, is_positive(-n))
            assert_equal(True, is_positive(n))
```

This test is even better. We test the first ten thousand integers (both positive and negative) and 0. It basically provides us with a better coverage than just the one across the boundary. So, keep this in mind. Zoom closely around each boundary with minimal granularity, but try to expand as well, finding a good compromise between optimal coverage and execution speed. We would love to check the first billion integers, but we can't wait days for our tests to run.

A more interesting example

Okay, this was as gentle an introduction as I could give you, so let's move on to something more interesting. Let's write and test a function that flattens a nested dictionary structure. For a couple of years, I have worked very closely with Twitter and Facebook APIs. Handling such humongous data structures is not easy, especially since they're often deeply nested. It turns out that it's much easier to flatten them in a way that you can work on them without losing the original structural information, and then recreate the nested structure from the flat one. To give you an example, we want something like this:

data_flatten.py

```
    nested = {
        'fullname': 'Alessandra',
        'age': 41,
        'phone-numbers': ['+447421234567', '+447423456789'],
        'residence': {
            'address': {
                'first-line': 'Alexandra Rd',
                'second-line': '',
```

```
            },
            'zip': 'N8 0PP',
            'city': 'London',
            'country': 'UK',
        },
    }

    flat = {
        'fullname': 'Alessandra',
        'age': 41,
        'phone-numbers': ['+447421234567', '+447423456789'],
        'residence.address.first-line': 'Alexandra Rd',
        'residence.address.second-line': '',
        'residence.zip': 'N8 0PP',
        'residence.city': 'London',
        'residence.country': 'UK',
    }
```

A structure like `flat` is much simpler to manipulate. Before writing the flattener, let's make some assumptions: the keys are strings, we leave every data structure as it is unless it's a dictionary, in which case we flatten it, we use the dot as separator, but we want to be able to pass a different one to our function. Here's the code:

`data_flatten.py`

```
    def flatten(data, prefix='', separator='.'):
        """Flattens a nested dict structure. """
        if not isinstance(data, dict):
            return {prefix: data} if prefix else data

        result = {}
        for (key, value) in data.items():
            result.update(
                flatten(
                    value,
                    _get_new_prefix(prefix, key, separator),
                    separator=separator))
        return result

    def _get_new_prefix(prefix, key, separator):
        return (separator.join((prefix, str(key)))
                if prefix else str(key))
```

The preceding example is not difficult, but also not trivial so let's go through it. At first, we check if `data` is a dictionary. If it's not a dictionary, then it's data that doesn't need to be flattened; therefore, we simply return either `data` or, if `prefix` is not an empty string, a dictionary with one key/value pair: `prefix/data`.

If instead `data` is a dict, we prepare an empty `result` dict to return, then we parse the list of `data`'s items, which, at I'm sure you will remember, are 2-tuples *(key, value)*. For each *(key, value)* pair, we recursively call `flatten` on them, and we update the `result` dict with what's returned by that call. Recursion is excellent when running through nested structures.

At a glance, can you understand what the `_get_new_prefix` function does? Let's use the inside-out technique once again. I see a ternary operator that returns the stringified `key` when `prefix` is an empty string. On the other hand, when `prefix` is a non-empty string, we use the `separator` to `join` the `prefix` with the stringified version of `key`. Notice that the braces inside the call to `join` aren't redundant, we need them. Can you figure out why?

Let's write a couple of tests for this function:

tests/test_ch7/test_data_flatten.py

```
    # ... imports omitted ...
    class FlattenTestCase(TestCase):

        def test_flatten(self):
            test_cases = [
                ({'A': {'B': 'C', 'D': [1, 2, 3], 'E': {'F': 'G'}},
                  'H': 3.14,
                  'J': ['K', 'L'],
                  'M': 'N'},
                 {'A.B': 'C',
                  'A.D': [1, 2, 3],
                  'A.E.F': 'G',
                  'H': 3.14,
                  'J': ['K', 'L'],
                  'M': 'N'}),
                (0, 0),
                ('Hello', 'Hello'),
                ({'A': None}, {'A': None}),
            ]
            for (nested, flat) in test_cases:
                assert_equal(flat, flatten(nested))

        def test_flatten_custom_separator(self):
```

```
nested = {'A': {'B': {'C': 'D'}}}
assert_equal(
    {'A#B#C': 'D'}, flatten(nested, separator='#'))
```

Let's start from `test_flatten`. I defined a list of 2-tuples `(nested, flat)`, each of which represents a test case (I highlighted `nested` to ease reading). I have one big dict with three levels of nesting, and then some smaller data structures that won't change when passed to the `flatten` function. These test cases are probably not enough to cover all edge cases, but they should give you a good idea of how you could structure a test like this. With a simple `for` loop, I cycle through each test case and assert that the result of `flatten(nested)` is equal to `flat`.

One thing to say about this example is that, when you run it, it will show you that two tests have been run. This is actually not correct because even if technically there were only two tests running, in one of them we have multiple test cases. It would be nicer to have them run in a way that they were recognized as separate. This is possible through the use of libraries such as `nose-parameterized`, which I encourage you to check out. It's on `https://pypi.python.org/pypi/nose-parameterized`.

I also provided a second test to make sure the custom separator feature worked. As you can see, I used only one data structure, which is much smaller. We don't need to go big again, nor to test other edge cases. Remember, tests should make sure of one thing and one thing only, and `test_flatten_custom_separator` just takes care of verifying whether or not we can feed the `flatten` function a different `separator`.

I could keep blathering on about tests for about another book if only I had the space, but unfortunately, we need to stop here. I haven't told you about **doctests** (tests written in the documentation using a Python interactive shell style), and about another half a million things that could be said about this subject. You'll have to discover that for yourself.

Take a look at the documentation for the `unittest` module, the `nose` and `nose-parameterized` libraries, and `pytest` (`http://pytest.org/`), and you will be fine. In my experience, mocking and patching seem to be quite hard to get a good grasp of for developers who are new to them, so allow yourself a little time to digest these techniques. Try and learn them gradually.

Test-driven development

Let's talk briefly about **test-driven development** or **TDD**. It is a methodology that was rediscovered by Kent Beck, who wrote *Test Driven Development by Example*, *Addison Wesley – 2002*, which I encourage you to check out if you want to learn about the fundamentals of this subject, which I'm quite obsessed with.

> *TDD is a software development methodology that is based on the continuous repetition of a very short development cycle.*

At first, the developer writes a test, and makes it run. The test is supposed to check a feature that is not yet part of the code. Maybe is a new feature to be added, or something to be removed or amended. Running the test will make it fail and, because of this, this phase is called **Red**.

When the test has failed, the developer writes the minimal amount of code to make it pass. When running the test succeeds, we have the so-called **Green** phase. In this phase, it is okay to write code that cheats, just to make the test pass (that's why you would then use triangulation). This technique is called, *fake it 'til you make it*.

The last piece of this cycle is where the developer takes care of both the code and the tests (in separate times) and refactors them until they are in the desired state. This last phase is called **Refactor**.

The **TDD mantra** therefore recites, **Red-Green-Refactor**.

At first, it feels really weird to write tests before the code, and I must confess it took me a while to get used to it. If you stick to it, though, and force yourself to learn this slightly counter-intuitive way of working, at some point something almost magical happens, and you will see the quality of your code increase in a way that wouldn't be possible otherwise.

When you write your code before the tests, you have to take care of *what* the code has to do and *how* it has to do it, both at the same time. On the other hand, when you write tests before the code, you can concentrate on the *what* part alone, while you write them. When you write the code afterwards, you will mostly have to take care of *how* the code has to implement *what* is required by the tests. This shift in focus allows your mind to concentrate on the *what* and *how* parts in separate moments, yielding a brain power boost that will surprise you.

There are several other benefits that come from the adoption of this technique:

- **You will refactor with much more confidence**: Because when you touch your code you know that if you screw things up, you will break at least one test. Moreover, you will be able to take care of the architectural design in the refactor phase, where having tests that act as guardians will allow you to enjoy massaging the code until it reaches a state that satisfies you.
- **The code will be more readable**: This is crucial in our time, when coding is a social activity and every professional developer spends much more time reading code than writing it.
- **The code will be more loose-coupled and easier to test and maintain**: This is simply because writing the tests first forces you to think more deeply about its structure.
- **Writing tests first requires you to have a better understanding of the business requirements**: This is fundamental in delivering what was actually asked for. If your understanding of the requirements is lacking information, you'll find writing a test extremely challenging and this situation acts as a sentinel for you.
- **Having everything unit tested means the code will be easier to debug**: Moreover, small tests are perfect for providing alternative documentation. English can be misleading, but five lines of Python in a simple test are very hard to be misunderstood.
- **Higher speed**: It's faster to write tests and code than it is to write the code first and then lose time debugging it. If you don't write tests, you will probably deliver the code sooner, but then you will have to track the bugs down and solve them (and, rest assured, there will be bugs). The combined time taken to write the code and then debug it is usually longer than the time taken to develop the code with TDD, where having tests running before the code is written, ensuring that the amount of bugs in it will be much lower than in the other case.

On the other hand, the main shortcomings of this technique are:

- **The whole company needs to believe in it**: Otherwise you will have to constantly argue with your boss, who will not understand why it takes you so long to deliver. The truth is, it may take you a bit longer to deliver in the short term, but in the long term you gain a lot with TDD. However, it is quite hard to see the long term because it's not under our noses like the short term is. I have fought battles with stubborn bosses in my career, to be able to code using TDD. Sometimes it has been painful, but always well worth it, and I have never regretted it because, in the end, the quality of the result has always been appreciated.

- **If you fail to understand the business requirements, this will reflect in the tests you write, and therefore it will reflect in the code too**: This kind of problem is quite hard to spot until you do UAT, but one thing that you can do to reduce the likelihood of it happening is to pair with another developer. Pairing will inevitably require discussions about the business requirements, and this will help having a better idea about them before the tests are written.
- **Badly written tests are hard to maintain**: This is a fact. Tests with too many mocks or with extra assumptions or badly structured data will soon become a burden. Don't let this discourage you; just keep experimenting and change the way you write them until you find a way that doesn't require you a huge amount of work every time you touch your code.

I'm so passionate about TDD that when I interview for a job, I always ask if the company I'm about to join adopts it. If the answer is no, it's kind of a deal-breaker for me. I encourage you to check it out and use it. Use it until you feel something clicking in your mind. You won't regret it, I promise.

Exceptions

Even though I haven't formally introduced them to you, by now I expect you to at least have a vague idea of what an **exception** is. In the previous chapters, we've seen that when an iterator is exhausted, calling `next` on it raises a `StopIteration` exception. We've met `IndexError` when we tried accessing a list at a position that was outside the valid range. We've also met `AttributeError` when we tried accessing an attribute on an object that didn't have it, and `KeyError` when we did the same with a key and a dictionary. We've also just met `AssertionError` when running tests.

Now, the time has come for us to talk about exceptions.

Sometimes, even though an operation or a piece of code is correct, there are conditions in which something may go wrong. For example, if we're converting user input from `string` to `int`, the user could accidentally type a letter in place of a digit, making it impossible for us to convert that value into a number. When dividing numbers, we may not know in advance if we're attempting a division by zero. When opening a file, it could be missing or corrupted.

When an error is detected during execution, it is called an **exception**. Exceptions are not necessarily lethal; in fact, we've seen that `StopIteration` is deeply integrated in Python generator and iterator mechanisms. Normally, though, if you don't take the necessary precautions, an exception will cause your application to break. Sometimes, this is the desired behavior but in other cases, we want to prevent and control problems such as these. For example, we may alert the user that the file they're trying to open is corrupted or that it is missing so that they can either fix it or provide another file, without the need for the application to die because of this issue. Let's see an example of a few exceptions:

`exceptions/first.example.py`

```
>>> gen = (n for n in range(2))
>>> next(gen)
0
>>> next(gen)
1
>>> next(gen)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> print(undefined_var)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'undefined_var' is not defined
>>> mylist = [1, 2, 3]
>>> mylist[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> mydict = {'a': 'A', 'b': 'B'}
>>> mydict['c']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'c'
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

As you can see, the Python shell is quite forgiving. We can see the `Traceback`, so that we have information about the error, but the program doesn't die. This is a special behavior, a regular program or a script would normally die if nothing were done to handle exceptions.

To handle an exception, Python gives you the `try` statement. What happens when you enter the `try` clause is that Python will watch out for one or more different types of exceptions (according to how you instruct it), and if they are raised, it will allow you to react. The `try` statement is comprised of the `try` clause, which opens the statement; one or more `except` clauses (all optional) that define what to do when an exception is caught; an `else` clause (optional), which is executed when the `try` clause is exited without any exception raised; and a `finally` clause (optional), whose code is executed regardless of whatever happened in the other clauses. The `finally` clause is typically used to clean up resources. Mind the order, it's important. Also, `try` must be followed by at least one `except` clause or a `finally` clause. Let's see an example:

`exceptions/try.syntax.py`

```
def try_syntax(numerator, denominator):
    try:
        print('In the try block: {}/{}'
              .format(numerator, denominator))
        result = numerator / denominator
    except ZeroDivisionError as zde:
        print(zde)
    else:
        print('The result is:', result)
        return result
    finally:
        print('Exiting')

print(try_syntax(12, 4))
print(try_syntax(11, 0))
```

The preceding example defines a simple `try_syntax` function. We perform the division of two numbers. We are prepared to catch a `ZeroDivisionError` exception if we call the function with `denominator = 0`. Initially, the code enters the `try` block. If `denominator` is not `0`, `result` is calculated and the execution, after leaving the `try` block, resumes in the `else` block. We print `result` and return it. Take a look at the output and you'll notice that just before returning `result`, which is the exit point of the function, Python executes the `finally` clause.

When `denominator` is `0`, things change. We enter the `except` block and print `zde`. The `else` block isn't executed because an exception was raised in the `try` block. Before (implicitly) returning `None`, we still execute the `finally` block. Take a look at the output and see if it makes sense to you:

```
$ python exceptions/try.syntax.py
In the try block: 12/4
The result is: 3.0
Exiting
3.0
In the try block: 11/0
division by zero
Exiting
None
```

When you execute a `try` block, you may want to catch more than one exception. For example, when trying to decode a JSON object, you may incur into `ValueError` for malformed JSON, or `TypeError` if the type of the data you're feeding to `json.loads()` is not a string. In this case, you may structure your code like this:

`exceptions/json.example.py`

```
import json
json_data = '{}'
try:
    data = json.loads(json_data)
except (ValueError, TypeError) as e:
    print(type(e), e)
```

This code will catch both `ValueError` and `TypeError`. Try changing `json_data = '{}'` to `json_data = 2` or `json_data = '{{'`, and you'll see the different output.

JSON stands for **JavaScript Object Notation** and it's an open standard format that uses human-readable text to transmit data objects consisting of key/value pairs. It's an exchange format widely used when moving data across applications, especially when data needs to be treated in a language or platform-agnostic way.

If you want to handle multiple exceptions differently, you can just add more `except` clauses, like this:

`exceptions/multiple.except.py`

```
try:
    # some code
except Exception1:
    # react to Exception1
except (Exception2, Exception3):
    # react to Exception2 and Exception3
except Exception3:
    # react to Exception3
...
```

Keep in mind that an exception is handled in the first block that defines that exception class or any of its bases. Therefore, when you stack multiple `except` clauses like we've just done, make sure that you put specific exceptions at the top and generic ones at the bottom. In OOP terms, children on top, grandparents at the bottom. Moreover, remember that only one `except` handler is executed when an exception is raised.

You can also write **custom exceptions**. In order to do that, you just have to inherit from any other exception class. Python built-in exceptions are too many to be listed here, so I have to point you towards the official documentation. One important thing to know is that every Python exception derives from `BaseException`, but your custom exceptions should never inherit directly from that one. The reason for it is that handling such an exception will trap also **system-exiting exceptions** such as `SystemExit` and `KeyboardInterrupt`, which derive from `BaseException`, and this could lead to severe issues. In case of disaster, you want to be able to *Ctrl* + *C* your way out of an application.

You can easily solve the problem by inheriting from `Exception`, which inherits from `BaseException`, but doesn't include any system-exiting exception in its children because they are siblings in the built-in exceptions hierarchy (see `https://docs.python.org/3/library/exceptions.html#exception-hierarchy`).

Programming with exceptions can be very tricky. You could inadvertently silence out errors, or trap exceptions that aren't meant to be handled. Play it safe by keeping in mind a few guidelines: always put in the `try` clause only the code that may cause the exception(s) that you want to handle. When you write `except` clauses, be as specific as you can, don't just resort to `except Exception` because it's easy. Use tests to make sure your code handles edge cases in a way that requires the least possible amount of exception handling. Writing an `except` statement without specifying any exception would catch any exception, therefore exposing your code to the same risks you incur when you derive your custom exceptions from `BaseException`.

You will find information about exceptions almost everywhere on the web. Some coders use them abundantly, others sparingly (I belong to the latter category). Find your own way of dealing with them by taking examples from other people's source code. There's plenty of interesting projects whose sources are open, and you can find them on either GitHub (`https://github.com`) or Bitbucket (`https://bitbucket.org/`).

Before we talk about **profiling**, let me show you an unconventional use of exceptions, just to give you something to help you expand your views on them. They are not just simply errors.

`exceptions/for.loop.py`

```
n = 100
found = False
for a in range(n):
    if found: break
    for b in range(n):
        if found: break
        for c in range(n):
            if 42 * a + 17 * b + c == 5096:
                found = True
                print(a, b, c)  # 79 99 95
```

The preceding code is quite a common idiom if you deal with numbers. You have to iterate over a few nested ranges and look for a particular combination of `a`, `b`, and `c` that satisfies a condition. In the example, condition is a trivial linear equation, but imagine something much cooler than that. What bugs me is having to check if the solution has been found at the beginning of each loop, in order to break out of them as fast as we can when it is. The break out logic interferes with the rest of the code and I don't like it, so I came up with a different solution for this. Take a look at it, and see if you can adapt it to other cases too.

`exceptions/for.loop.py`

```
class ExitLoopException(Exception):
    pass

try:
    n = 100
    for a in range(n):
        for b in range(n):
            for c in range(n):
                if 42 * a + 17 * b + c == 5096:
                    raise ExitLoopException(a, b, c)
except ExitLoopException as ele:
    print(ele)  # (79, 99, 95)
```

Can you see how much more elegant it is? Now the breakout logic is entirely handled with a simple exception whose name even hints at its purpose. As soon as the result is found, we raise it, and immediately the control is given to the except clause which handles it. This is food for thought. This example indirectly shows you how to raise your own exceptions. Read up on the official documentation to dive into the beautiful details of this subject.

Profiling Python

There are a few different ways to profile a Python application. Profiling means having the application run while keeping track of several different parameters, like the number of times a function is called, the amount of time spent inside it, and so on. Profiling can help us find the bottlenecks in our application, so that we can improve only what is really slowing us down.

If you take a look at the profiling section in the standard library official documentation, you will see that there are a couple of different implementations of the same profiling interface: `profile` and `cProfile`.

- `cProfile` is recommended for most users, it's a C extension with reasonable overhead that makes it suitable for profiling long-running programs
- `profile` is a pure Python module whose interface is imitated by `cProfile`, but which adds significant overhead to profiled programs

This interface does **determinist profiling**, which means that all function calls, function returns and exception events are monitored, and precise timings are made for the intervals between these events. Another approach, called **statistical profiling**, randomly samples the effective instruction pointer, and deduces where time is being spent.

The latter usually involves less overhead, but provides only approximate results. Moreover, because of the way the Python interpreter runs the code, deterministic profiling doesn't add that as much overhead as one would think, so I'll show you a simple example using `cProfile` from the command line.

We're going to calculate Pythagorean triples (I know, you've missed them...) using the following code:

`profiling/triples.py`

```
def calc_triples(mx):
    triples = []
    for a in range(1, mx + 1):
        for b in range(a, mx + 1):
            hypotenuse = calc_hypotenuse(a, b)
            if is_int(hypotenuse):
                triples.append((a, b, int(hypotenuse)))
    return triples

def calc_hypotenuse(a, b):
    return (a**2 + b**2) ** .5

def is_int(n):  # n is expected to be a float
    return n.is_integer()

triples = calc_triples(1000)
```

The script is extremely simple; we iterate over the interval [1, *mx*] with `a` and `b` (avoiding repetition of pairs by setting `b >= a`) and we check if they belong to a right triangle. We use `calc_hypotenuse` to get `hypotenuse` for `a` and `b`, and then, with `is_int`, we check if it is an integer, which means (*a*, *b*, *c*) is a Pythagorean triple. When we profile this script, we get information in tabular form. The columns are `ncalls`, `tottime`, `percall`, `cumtime`, `percall`, and `filename:lineno(function)`. They represent the amount of calls we made to a function, how much time we spent in it, and so on. I'll trim a couple of columns to save space, so if you run the profiling yourself, don't worry if you get a different result.

```
$ python -m cProfile profiling/triples.py
1502538 function calls in 0.750 seconds
Ordered by: standard name
ncalls  tottime  percall filename:lineno(function)
500500    0.469    0.000 triples.py:14(calc_hypotenuse)
500500    0.087    0.000 triples.py:18(is_int)
```

```
     1    0.000    0.000 triples.py:4(<module>)
     1    0.163    0.163 triples.py:4(calc_triples)
     1    0.000    0.000 {built-in method exec}
  1034    0.000    0.000 {method 'append' of 'list' objects}
     1    0.000    0.000 {method 'disable' of '_lsprof.Profil...
500500    0.032    0.000 {method 'is_integer' of 'float' objects}
```

Even with this limited amount of data, we can still infer some useful information about this code. Firstly, we can see that the time complexity of the algorithm we have chosen grows with the square of the input size. The amount of times we get inside the inner loop body is exactly *mx (mx + 1) / 2*. We run the script with `mx = 1000`, which means we get `500500` times inside the inner `for` loop. Three main things happen inside that loop, we call `calc_hypotenuse`, we call `is_int` and, if the condition is met, we append to the `triples` list.

Taking a look at the profiling report, we notice that the algorithm has spent `0.469` seconds inside `calc_hypotenuse`, which is way more than the `0.087` seconds spent inside `is_int`, given that they were called the same number of times, so let's see if we can boost `calc_hypotenuse` a little.

As it turns out, we can. As I mentioned earlier on in the book, the power operator `**` is quite expensive, and in `calc_hypotenuse`, we're using it three times. Fortunately, we can easily transform two of those into simple multiplications, like this:

profiling/triples.py

```
    def calc_hypotenuse(a, b):
        return (a*a + b*b) ** .5
```

This simple change should improve things. If we run the profiling again, we see that now the `0.469` is now down to `0.177`. Not bad! This means now we're spending only about 37% of the time inside `calc_hypotenuse` as we were before.

Let's see if we can improve `is_int` as well, by changing it like this:

profiling/triples.py

```
    def is_int(n):
        return n == int(n)
```

This implementation is different and the advantage is that it also works when `n` is an integer. Alas, when we run the profiling against it, we see that the time taken inside the `is_int` function has gone up to `0.141` seconds. This means that it has roughly doubled, compared to what it was before. In this case, we need to revert to the previous implementation.

This example was trivial, of course, but enough to show you how one could profile an application. Having the amount of calls that are performed against a function helps us understand better the time complexity of our algorithms. For example, you wouldn't believe how many coders fail to see that those two `for` loops run proportionally to the square of the input size.

One thing to mention: depending on what system you're using, results may be different. Therefore, it's quite important to be able to profile software on a system that is as close as possible to the one the software is deployed on, if not actually on that one.

When to profile?

Profiling is super cool, but we need to know when it is appropriate to do it, and in what measure we need to address the results we get from it.

Donald Knuth once said that *premature optimization is the root of all evil* and, although I wouldn't have put it down so drastically, I do agree with him. After all, who am I to disagree with the man that gave us *The Art of Computer Programming*, *TeX*, and some of the coolest algorithms I have ever studied when I was a university student?

So, first and foremost: *correctness*. You want you code to deliver the result correctly, therefore write tests, find edge cases, and stress your code in every way you think makes sense. Don't be protective, don't put things in the back of your brain for later because you think they're not likely to happen. Be thorough.

Secondly, take care of coding *best practices*. Remember readability, extensibility, loose coupling, modularity, and design. Apply OOP principles: encapsulation, abstraction, single responsibility, open/closed, and so on. Read up on these concepts. They will open horizons for you, and they will expand the way you think about code.

Thirdly, *refactor like a beast!* The Boy Scouts Rule says to *Always leave the campground cleaner than you found it*. Apply this rule to your code.

And, finally, when all of the above has been taken care of, then and only then, you take care of profiling.

Run your profiler and identify bottlenecks. When you have an idea of the bottlenecks you need to address, start with the worst one first. Sometimes, fixing a bottleneck causes a ripple effect that will expand and change the way the rest of the code works. Sometimes this is only a little, sometimes a bit more, according to how your code was designed and implemented. Therefore, start with the biggest issue first.

One of the reasons Python is so popular is that it is possible to implement it in many different ways. So, if you find yourself having troubles boosting up some part of your code using sheer Python, nothing prevents you from rolling up your sleeves, buying a couple of hundred liters of coffee, and rewriting the slow piece of code in C. Guaranteed to be fun!

Summary

In this chapter, we explored the world of testing, exceptions, and profiling.

I tried to give you a fairly comprehensive overview of testing, especially unit testing, which is the kind of testing that a developer mostly does. I hope I have succeeded in channeling the message that testing is not something that is perfectly defined and that you can learn from a book. You need to experiment with it a lot before you get comfortable. Of all the efforts a coder must make in terms of study and experimentation, I'd say testing is one of those that are most worth it.

We've briefly seen how we can prevent our program from dying because of errors, called exceptions, that happen at runtime. And, to steer away from the usual ground, I have given you an example of a somewhat unconventional use of exceptions to break out of nested `for` loops. That's not the only case, and I'm sure you'll discover others as you grow as a coder.

In the end, we very briefly touched base on profiling, with a simple example and a few guidelines. I wanted to talk about profiling for the sake of completeness, so at least you can play around with it.

We're now about to enter *Chapter 8*, *The Edges – GUIs and Scripts*, where we're going to get our hands dirty with scripts and GUIs and, hopefully, come up with something interesting.

I am aware that I gave you a lot of pointers in this chapter, with no links or directions. I'm afraid this is by choice. As a coder, there won't be a single day at work when you won't have to look something up in a documentation page, in a manual, on a website, and so on. I think it's vital for a coder to be able to search effectively for the information they need, so I hope you'll forgive me for this extra training. After all, it's all for your benefit.

8
The Edges – GUIs and Scripts

"A user interface is like a joke. If you have to explain it, it's not that good."

- Martin LeBlanc

In this chapter, we're going to work on a project together. We're going to prepare a very simple HTML page with a few images, and then we're going to scrape it, in order to save those images.

We're going to write a script to do this, which will allow us to talk about a few concepts that I'd like to run by you. We're also going to add a few options to save images based on their format, and to choose the way we save them. And, when we're done with the script, we're going to write a GUI application that does basically the same thing, thus killing two birds with one stone. Having only one project to explain will allow me to show a wider range of topics in this chapter.

A **graphical user interface** (**GUI**) is a type of interface that allows the user to interact with an electronic device through graphical icons, buttons and widgets, as opposed to text-based or command-line interfaces, which require commands or text to be typed on the keyboard. In a nutshell, any browser, any office suite such as LibreOffice, and, in general, anything that pops up when you click on an icon, is a GUI application.

So, if you haven't already done so, this would be the perfect time to start a console and position yourself in a folder called `ch8` in the root of your project for this book. Within that folder, we'll create two Python modules (`scrape.py` and `guiscrape.py`) and one standard folder (`simple_server`). Within `simple_server`, we'll write our HTML page (`index.html`) in `simple_server`. Images will be stored in `ch8/simple_server/img`. The structure in `ch8` should look like this:

```
$ tree -A
.
├── guiscrape.py
├── scrape.py
└── simple_server
    ├── img
    │   ├── owl-alcohol.png
    │   ├── owl-book.png
    │   ├── owl-books.png
    │   ├── owl-ebook.jpg
    │   └── owl-rose.jpeg
    ├── index.html
    └── serve.sh
```

If you're using either Linux or Mac, you can do what I do and put the code to start the HTTP server in a `serve.sh` file. On Windows, you'll probably want to use a batch file.

The HTML page we're going to scrape has the following structure:

`simple_server/index.html`

```
<!DOCTYPE html>
<html lang="en">
  <head><title>Cool Owls!</title></head>
  <body>
    <h1>Welcome to my owl gallery</h1>
    <div>
      <img src="img/owl-alcohol.png" height="128" />
      <img src="img/owl-book.png" height="128" />
      <img src="img/owl-books.png" height="128" />
      <img src="img/owl-ebook.jpg" height="128" />
      <img src="img/owl-rose.jpeg" height="128" />
```

```
    </div>
    <p>Do you like my owls?</p>
  </body>
</html>
```

It's an extremely simple page, so let's just note that we have five images, three of which are PNGs and two are JPGs (note that even though they are both JPGs, one ends with `.jpg` and the other with `.jpeg`, which are both valid extensions for this format).

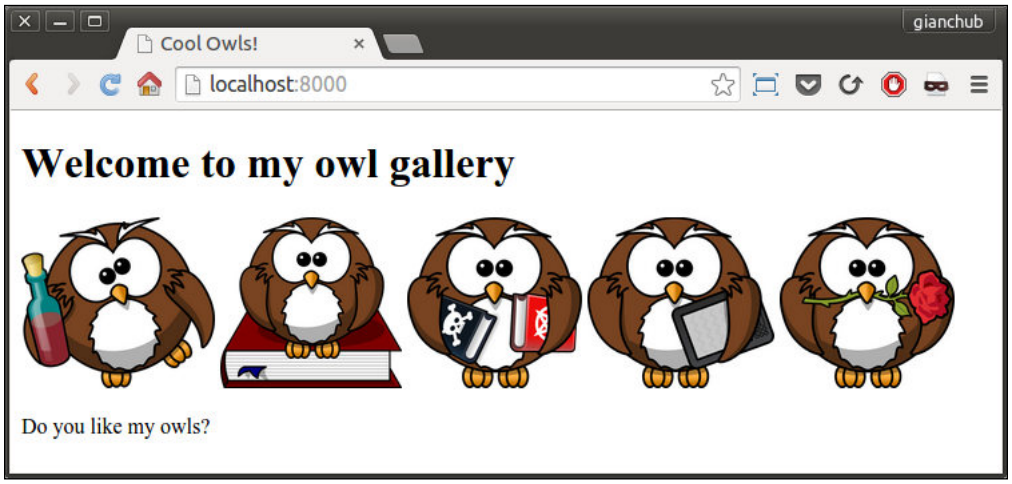
So, Python gives you a very simple HTTP server for free that you can start with the following command (in the `simple_server` folder):

```
$ python -m http.server 8000
Serving HTTP on 0.0.0.0 port 8000 ...
127.0.0.1 - - [31/Aug/2015 16:11:10] "GET / HTTP/1.1" 200 -
```

The last line is the log you get when you access `http://localhost:8000`, where our beautiful page will be served. Alternatively, you can put that command in a file called `serve.sh`, and just run that with this command (make sure it's executable):

```
$ ./serve.sh
```

It will have the same effect. If you have the code for this book, your page should look something like this:



Feel free to use any other set of images, as long as you use at least one PNG and one JPG, and that in the `src` tag you use relative paths, not absolute. I got those lovely owls from `https://openclipart.org/`.

First approach – scripting

Now, let's start writing the script. I'll go through the source in three steps: imports first, then the argument parsing logic, and finally the business logic.

The imports

`scrape.py (Imports)`

```
import argparse
import base64
import json
import os
from bs4 import BeautifulSoup
import requests
```

Going through them from the top, you can see that we'll need to parse the arguments. which we'll feed to the script itself (`argparse`). We will need the `base64` library to save the images within a JSON file (`base64` and `json`), and we'll need to open files for writing (`os`). Finally, we'll need `BeautifulSoup` for scraping the web page easily, and `requests` to fetch its content. `requests` is an extremely popular library for performing HTTP requests, built to avoid the difficulties and quirks of using the standard library `urllib` module. It's based on the fast `urllib3` third-party library.

We will explore the HTTP protocol and `requests` mechanism in *Chapter 10*, *Web Development Done Right* so, for now, let's just (simplistically) say that we perform an HTTP request to fetch the content of a web page. We can do it programmatically using a library such as `requests`, and it's more or less the equivalent of typing a URL in your browser and pressing *Enter* (the browser then fetches the content of a web page and also displays it to you).

Of all these imports, only the last two don't belong to the Python standard library, but they are so widely used throughout the world that I dare not exclude them in this book. Make sure you have them installed:

```
$ pip freeze | egrep -i "soup|requests"
beautifulsoup4==4.4.0
requests==2.7.0
```

Of course, the version numbers might be different for you. If they're not installed, use this command to do so:

```
$ pip install beautifulsoup4 requests
```

At this point, the only thing that I reckon might confuse you is the `base64/json` couple, so allow me to spend a few words on that.

As we saw in the previous chapter, JSON is one of the most popular formats for data exchange between applications. It's also widely used for other purposes too, for example, to save data in a file. In our script, we're going to offer the user the ability to save images as image files, or as a JSON single file. Within the JSON, we'll put a dictionary with keys as the images names and values as their content. The only issue is that saving images in the binary format is tricky, and this is where the `base64` library comes to the rescue. **Base64** is a very popular binary-to-text encoding scheme that represents binary data in an ASCII string format by translating it into a radix-64 representation.

The **radix-64** representation uses the letters *A-Z*, *a-z*, and the digits *0-9*, plus the two symbols + and / for a grand total of 64 symbols altogether. Therefore, not surprisingly, the Base64 alphabet is made up of these 64 symbols.

If you think you have never used it, think again. Every time you send an email with an image attached to it, the image gets encoded with Base64 before the email is sent. On the recipient side, images are automatically decoded into their original binary format so that the email client can display them.

Parsing arguments

Now that the technicalities are out of the way, let's see the second section of our script (it should be at the end of the `scrape.py` module).

```
scrape.py (Argument parsing and scraper triggering)

    if __name__ == "__main__":
        parser = argparse.ArgumentParser(
            description='Scrape a webpage.')
        parser.add_argument(
            '-t',
            '--type',
            choices=['all', 'png', 'jpg'],
            default='all',
            help='The image type we want to scrape.')
        parser.add_argument(
            '-f',
            '--format',
            choices=['img', 'json'],
```

```
        default='img',
        help='The format images are saved to.')
    parser.add_argument(
        'url',
        help='The URL we want to scrape for images.')
    args = parser.parse_args()
    scrape(args.url, args.format, args.type)
```

Look at that first line; it is a very common idiom when it comes to scripting. According to the official Python documentation, the string `'__main__'` is the name of the scope in which top-level code executes. A module's `__name__` is set equal to `'__main__'` when read from standard input, a script, or from an interactive prompt.

Therefore, if you put the execution logic under that `if`, the result is that you will be able to use the module as a library should you need to import any of the functions or objects defined in it, because when importing it from another module, `__name__` won't be `'__main__'`. On the other hand, when you run the script directly, like we're going to, `__name__` will be `'__main__'`, so the execution logic will run.

The first thing we do then is define our parser. I would recommend using the standard library module, `argparse`, which is simple enough and quite powerful. There are other options out there, but in this case, `argparse` will provide us with all we need.

We want to feed our script three different data: the type of images we want to save, the format in which we want to save them, and the URL for the page to be scraped.

The type can be PNG, JPG or both (default), while the format can be either image or JSON, image being the default. URL is the only mandatory argument.

So, we add the `-t` option, allowing also the long version `--type`. The choices are `'all'`, `'png'`, and `'jpg'`. We set the default to `'all'` and we add a help message.

We do a similar procedure for the `format` argument allowing both the short and long syntax (`-f` and `--format`), and finally we add the `url` argument, which is the only one that is specified differently so that it won't be treated as an option, but rather as a positional argument.

In order to parse all the arguments, all we need is `parser.parse_args()`. Very simple, isn't it?

The last line is where we trigger the actual logic, by calling the `scrape` function, passing all the arguments we just parsed. We will see its definition shortly.

The nice thing about `argparse` is that if you call the script by passing `-h`, it will print a nice **usage text** for you automatically. Let's try it out:

```
$ python scrape.py -h
usage: scrape.py [-h] [-t {all,png,jpg}] [-f {img,json}] url


Scrape a webpage.


positional arguments:
  url                   The URL we want to scrape for images.


optional arguments:
  -h, --help            show this help message and exit
  -t {all,png,jpg}, --type {all,png,jpg}
                        The image type we want to scrape.
  -f {img,json}, --format {img,json}
                        The format images are saved to.
```

If you think about it, the one true advantage of this is that we just need to specify the arguments and we don't have to worry about the usage text, which means we won't have to keep it in sync with the arguments' definition every time we change something. This is precious.

Here's a few different ways to call our `scrape.py` script, which demonstrate that `type` and `format` are optional, and how you can use the short and long syntax to use them:

```
$ python scrape.py http://localhost:8000
$ python scrape.py -t png http://localhost:8000
$ python scrape.py --type=jpg -f json http://localhost:8000
```

The first one is using default values for `type` and `format`. The second one will save only PNG images, and the third one will save only JPGs, but in JSON format.

The business logic

Now that we've seen the scaffolding, let's dive deep into the actual logic (if it looks intimidating don't worry; we'll go through it together). Within the script, this logic lies after the imports and before the parsing (before the `if __name__` clause):

```
scrape.py (Business logic)
```

```
    def scrape(url, format_, type_):
        try:
            page = requests.get(url)
        except requests.RequestException as rex:
            print(str(rex))
        else:
            soup = BeautifulSoup(page.content, 'html.parser')
            images = _fetch_images(soup, url)
            images = _filter_images(images, type_)
            _save(images, format_)

    def _fetch_images(soup, base_url):
        images = []
        for img in soup.findAll('img'):
            src = img.get('src')
            img_url = (
                '{base_url}/{src}'.format(
                    base_url=base_url, src=src))
            name = img_url.split('/')[-1]
            images.append(dict(name=name, url=img_url))
        return images

    def _filter_images(images, type_):
        if type_ == 'all':
            return images
        ext_map = {
            'png': ['.png'],
            'jpg': ['.jpg', '.jpeg'],
        }
        return [
            img for img in images
            if _matches_extension(img['name'], ext_map[type_])
        ]

    def _matches_extension(filename, extension_list):
        name, extension = os.path.splitext(filename.lower())
        return extension in extension_list

    def _save(images, format_):
        if images:
            if format_ == 'img':
                _save_images(images)
            else:
```

```
                _save_json(images)
            print('Done')
        else:
            print('No images to save.')

    def _save_images(images):
        for img in images:
            img_data = requests.get(img['url']).content
            with open(img['name'], 'wb') as f:
                f.write(img_data)

    def _save_json(images):
        data = {}
        for img in images:
            img_data = requests.get(img['url']).content
            b64_img_data = base64.b64encode(img_data)
            str_img_data = b64_img_data.decode('utf-8')
            data[img['name']] = str_img_data

        with open('images.json', 'w') as ijson:
            ijson.write(json.dumps(data))
```

Let's start with the `scrape` function. The first thing it does is fetch the page at the given `url` argument. Whatever error may happen while doing this, we trap it in the `RequestException rex` and we print it. The `RequestException` is the base exception class for all the exceptions in the `requests` library.

However, if things go well, and we have a page back from the `GET` request, then we can proceed (`else` branch) and feed its content to the `BeautifulSoup` parser. The `BeautifulSoup` library allows us to parse a web page in no time, without having to write all the logic that would be needed to find all the images in a page, which we really don't want to do. It's not as easy as it seems, and reinventing the wheel is never good. To fetch images, we use the `_fetch_images` function and we filter them with `_filter_images`. Finally, we call `_save` with the result.

Splitting the code into different functions with meaningful names allows us to read it more easily. Even if you haven't seen the logic of the `_fetch_images`, `_filter_images`, and `_save` functions, it's not hard to predict what they do, right?

`_fetch_images` takes a `BeautifulSoup` object and a base URL. All it does is looping through all of the images found on the page and filling in the `'name'` and `'url'` information about them in a dictionary (one per image). All dictionaries are added to the `images` list, which is returned at the end.

There is some trickery going on when we get the name of an image. What we do is split the `img_url` (`http://localhost:8000/img/my_image_name.png`) string using `'/'` as a separator, and we take the last item as the image name. There is a more robust way of doing this, but for this example it would be overkill. If you want to see the details of each step, try to break this logic down into smaller steps, and print the result of each of them to help yourself understand.

Towards the end of the book, I'll show you another technique to debug in a much more efficient way.

Anyway, by just adding `print(images)` at the end of the `_fetch_images` function, we get this:

```
[{'url': 'http://localhost:8000/img/owl-alcohol.png', 'name': 'owl-
alcohol.png'}, {'url': 'http://localhost:8000/img/owl-book.png', 'name':
'owl-book.png'}, ...]
```

I truncated the result for brevity. You can see each dictionary has a `'url'` and `'name'` key/value pair, which we can use to fetch, identify and save our images as we like. At this point, I hear you asking what would happen if the images on the page were specified with an absolute path instead of a relative one, right? Good question!

The answer is that the script will fail to download them because this logic expects relative paths. I was about to add a bit of logic to solve this issue when I thought that, at this stage, it would be a nice exercise for you to do it, so I'll leave it up to you to fix it.

Hint: inspect the start of that `src` variable. If it starts with `'http'`, then it's probably an absolute path.

I hope the body of the `_filter_images` function is interesting to you. I wanted to show you how to check on multiple extensions by using a mapping technique.

In this function, if `type_` is `'all'`, then no filtering is required, so we just return all the images. On the other hand, when `type_` is not `'all'`, we get the allowed extensions from the `ext_map` dictionary, and use it to filter the images in the list comprehension that ends the function body. You can see that by using another helper function, `_matches_extension`, I have made the list comprehension simpler and more readable.

All `_matches_extension` does is split the name of the image getting its extension and checking whether it is within the list of allowed ones. Can you find one micro improvement (speed-wise) that could be done to this function?

I'm sure that you're wondering why I have collected all the images in the list and then removed them, instead of checking whether I wanted to save them before adding them to the list. The first reason is that I needed `_fetch_images` in the GUI app as it is now. A second reason is that combining, fetching, and filtering would produce a longer and a bit more complicated function, and I'm trying to keep the complexity level down. A third reason is that this could be a nice exercise for you to do. Feels like we're pairing here...

Let's keep going through the code and inspect the `_save` function. You can see that, when `images` isn't empty, this basically acts as a dispatcher. We either call `_save_images` or `_save_json`, depending on which information is stored in the `format_` variable.

We are almost done. Let's jump to `_save_images`. We loop on the `images` list and for each dictionary we find there we perform a `GET` request on the image URL and save its content in a file, which we name as the image itself. The one important thing to note here is how we save that file.

We use a **context manager**, represented by the keyword `with`, to do that. Python's `with` statement supports the concept of a runtime context defined by a context manager. This is implemented using a pair of methods (`contextmanager.__enter__()` and `contextmanager.__exit__(exc_type, exc_val, exc_tb)`) that allow user-defined classes to define a runtime context that is entered before the statement body is executed and exited when the statement ends.

In our case, using a context manager, in conjunction with the open function, gives us the guarantee that if anything bad were to happen while writing that file, the resources involved in the process will be cleaned up and released properly regardless of the error. Have you ever tried to delete a file on Windows, only to be presented with an alert that tells you that you cannot delete the file because there is another process that is holding on to it? We're avoiding that sort of very annoying thing.

When we open a file, we get a handler for it and, no matter what happens, we want to be sure we release it when we're done with the file. A context manager is the tool we need to make sure of that.

Finally, let's now step into the `_save_json` function. It's very similar to the previous one. We basically fill in the `data` dictionary. The image name is the *key*, and the Base64 representation of its binary content is the *value*. When we're done populating our dictionary, we use the `json` library to dump it in the `images.json` file. I'll give you a small preview of that:

`images.json (truncated)`

```
{
```

```
  "owl-ebook.jpg": "/9j/4AAQSkZJRgABAQEAMQAxAAD/2wBDAAEBAQ...
  "owl-book.png": "iVBORw0KGgoAAAANSUhEUgAAASwAAAEbCAYAAAB...
  "owl-books.png": "iVBORw0KGgoAAAANSUhEUgAAASwAAAElCAYAAA...
  "owl-alcohol.png": "iVBORw0KGgoAAAANSUhEUgAAASwAAAEICAYA...
  "owl-rose.jpeg": "/9j/4AAQSkZJRgABAQEANAA0AAD/2wBDAAEBAQ...
}
```

And that's it! Now, before proceeding to the next section, make sure you play with this script and understand well how it works. Try and modify something, print out intermediate results, add a new argument or functionality, or scramble the logic. We're going to migrate it into a GUI application now, which will add a layer of complexity simply because we'll have to build the GUI interface, so it's important that you're well acquainted with the business logic: it will allow you to concentrate on the rest of the code.

Second approach – a GUI application

There are several libraries to write GUI applications in Python. The most famous ones are **tkinter**, **wxPython**, **PyGTK**, and **PyQt**. They all offer a wide range of tools and widgets that you can use to compose a GUI application.

The one I'm going to use for the rest of this chapter is tkinter. **tkinter** stands for **Tk interface** and it is the standard Python interface to the Tk GUI toolkit. Both Tk and tkinter are available on most Unix platforms, Mac OS X, as well as on Windows systems.

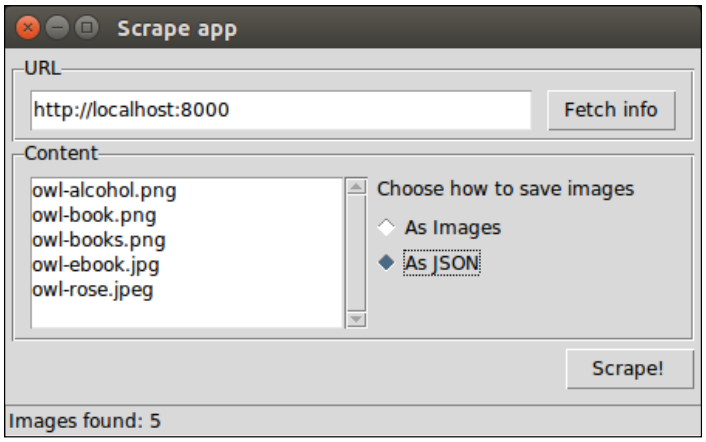
Let's make sure that `tkinter` is installed properly on your system by running this command:

```
$ python -m tkinter
```

It should open a dialog window demonstrating a simple `Tk` interface. If you can see that, then we're good to go. However, if it doesn't work, please search for `tkinter` in the Python official documentation. You will find several links to resources that will help you get up and running with it.

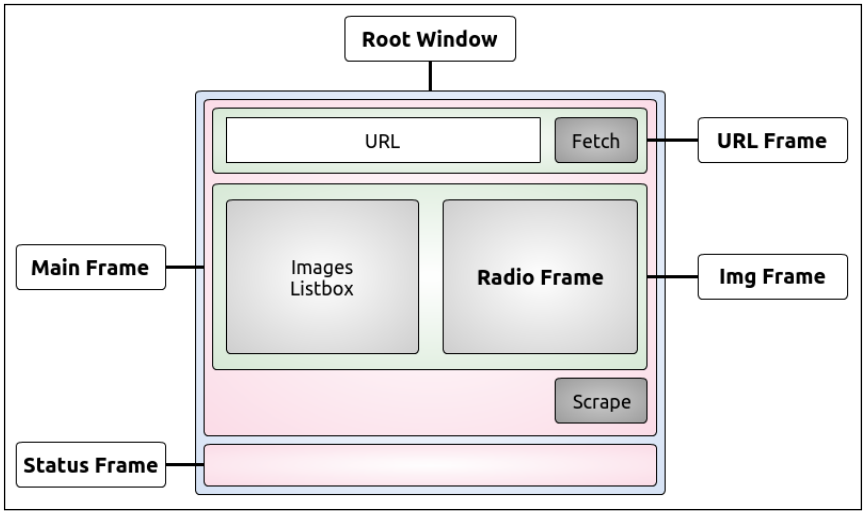
We're going to make a very simple GUI application that basically mimics the behavior of the script we saw in the first part of this chapter. We won't add the ability to save JPGs or PNGs singularly, but after you've gone through this chapter, you should be able to play with the code and put that feature back in by yourself.

So, this is what we're aiming for:



Gorgeous, isn't it? As you can see, it's a very simple interface (this is how it should look on Ubuntu). There is a frame (that is, a container) for the **URL** field and the **Fetch info** button, another frame for the **Listbox** to hold the image names and the radio button to control the way we save them, and finally there is a **Scrape!** button at the bottom. We also have a status bar, which shows us some information.

In order to get this layout, we could just place all the widgets on a root window, but that would make the layout logic quite messy and unnecessarily complicated. So, instead, we will divide the space using frames and place the widgets in those frames. This way we will achieve a much nicer result. So, this is the draft for the layout:



We have a **Root Window**, which is the main window of the application. We divide it into two rows, the first one in which we place the **Main Frame**, and the second one in which we place the **Status Frame** (which will hold the status bar). The **Main Frame** is subsequently divided into three rows itself. In the first one we place the **URL Frame**, which holds the **URL** widgets. In the second one we place the **Img Frame**, which will hold the **Listbox** and the **Radio Frame**, which will host a label and the radio button widgets. And finally a third one, which will just hold the **Scrape** button.

In order to lay out frames and widgets, we will use a layout manager called *grid*, that simply divides up the space into rows and columns, as in a matrix.

Now, all the code I'm going to write comes from the `guiscrape.py` module, so I won't repeat its name for each snippet, to save space. The module is logically divided into three sections, not unlike the script version: imports, layout logic, and business logic. We're going to analyze them line by line, in three chunks.

The imports

```
from tkinter import *
from tkinter import ttk, filedialog, messagebox
import base64
import json
import os
from bs4 import BeautifulSoup
import requests
```

We're already familiar with most of these. The interesting bit here is those first two lines. The first one is quite common practice, although it is bad practice in Python to import using the *star syntax*. You can incur in name collisions and, if the module is too big, importing everything would be expensive.

After that, we import `ttk`, `filedialog`, and `messagebox` explicitly, following the conventional approach used with this library. `ttk` is the new set of styled widgets. They behave basically like the old ones, but are capable of drawing themselves correctly according to the style your OS is set on, which is nice.

The rest of the imports is what we need in order to carry out the task you know well by now. Note that there is nothing we need to install with `pip` in this second part, we already have everything we need.

The layout logic

I'm going to paste it chunk by chunk so that I can explain it easily to you. You'll see how all those pieces we talked about in the layout draft are arranged and glued together. What I'm about to paste, as we did in the script before, is the final part of the `guiscrape.py` module. We'll leave the middle part, the business logic, for last.

```
if __name__ == "__main__":
    _root = Tk()
    _root.title('Scrape app')
```

As you know by now, we only want to execute the logic when the module is run directly, so that first line shouldn't surprise you.

In the last two lines. we set up the main window, which is an instance of the `Tk` class. We instantiate it and give it a title. Note that I use the prepending underscore technique for all the names of the `tkinter` objects, in order to avoid potential collisions with names in the business logic. I just find it cleaner like this, but you're allowed to disagree.

```
    _mainframe = ttk.Frame(_root, padding='5 5 5 5')
    _mainframe.grid(row=0, column=0, sticky=(E, W, N, S))
```

Here, we set up the **Main Frame**. It's a `ttk.Frame` instance. We set `_root` as its parent, and give it some padding. The `padding` is a measure in pixels of how much space should be inserted between the inner content and the borders in order to let our layout breathe a little, otherwise we have the *sardine effect*, where widgets are packed too tightly.

The second line is much more interesting. We place this `_mainframe` on the first row (`0`) and first column (`0`) of the parent object (`_root`). We also say that this frame needs to extend itself in each direction by using the `sticky` argument with all four cardinal directions. If you're wondering where they came from, it's the `from tkinter import *` magic that brought them to us.

```
    _url_frame = ttk.LabelFrame(
        _mainframe, text='URL', padding='5 5 5 5')
    _url_frame.grid(row=0, column=0, sticky=(E, W))
    _url_frame.columnconfigure(0, weight=1)
    _url_frame.rowconfigure(0, weight=1)
```

Next, we start by placing the **URL Frame** down. This time, the parent object is `_mainframe`, as you will recall from our draft. This is not just a simple `Frame`, but it's actually a `LabelFrame`, which means we can set the text argument and expect a rectangle to be drawn around it, with the content of the text argument written in the top-left part of it (check out the previous picture if it helps). We position this frame at (0, 0), and say that it should expand to the left and to the right. We don't need the other two directions.

Finally, we use `rowconfigure` and `columnconfigure` to make sure it behaves correctly, should it need to resize. This is just a formality in our present layout.

```
_url = StringVar()
_url.set('http://localhost:8000')
_url_entry = ttk.Entry(
    _url_frame, width=40, textvariable=_url)
_url_entry.grid(row=0, column=0, sticky=(E, W, S, N), padx=5)
_fetch_btn = ttk.Button(
    _url_frame, text='Fetch info', command=fetch_url)
_fetch_btn.grid(row=0, column=1, sticky=W, padx=5)
```

Here, we have the code to lay out the URL textbox and the `_fetch` button. A textbox in this environment is called `Entry`. We instantiate it as usual, setting `_url_frame` as its parent and giving it a width. Also, and this is the most interesting part, we set the `textvariable` argument to be `_url`. `_url` is a `StringVar`, which is an object that is now connected to `Entry` and will be used to manipulate its content. Therefore, we don't modify the text in the `_url_entry` instance directly, but by accessing `_url`. In this case, we call the `set` method on it to set the initial value to the URL of our local web page.

We position `_url_entry` at (0, 0), setting all four cardinal directions for it to stick to, and we also set a bit of extra padding on the left and right edges by using `padx`, which adds padding on the x-axis (horizontal). On the other hand, `pady` takes care of the vertical direction.

By now, you should get that every time you call the `.grid` method on an object, we're basically telling the grid layout manager to place that object somewhere, according to rules that we specify as arguments in the `grid()` call.

Similarly, we set up and place the `_fetch` button. The only interesting parameter is `command=fetch_url`. This means that when we click this button, we actually call the `fetch_url` function. This technique is called **callback**.

```
_img_frame = ttk.LabelFrame(
    _mainframe, text='Content', padding='9 0 0 0')
_img_frame.grid(row=1, column=0, sticky=(N, S, E, W))
```

This is what we called **Img Frame** in the layout draft. It is placed on the second row of its parent `_mainframe`. It will hold the `Listbox` and the **Radio Frame**.

```
_images = StringVar()
_img_listbox = Listbox(
    _img_frame, listvariable=_images, height=6, width=25)
_img_listbox.grid(row=0, column=0, sticky=(E, W), pady=5)
_scrollbar = ttk.Scrollbar(
    _img_frame, orient=VERTICAL, command=_img_listbox.yview)
_scrollbar.grid(row=0, column=1, sticky=(S, N), pady=6)
_img_listbox.configure(yscrollcommand=_scrollbar.set)
```

This is probably the most interesting bit of the whole layout logic. As we did with the `_url_entry`, we need to drive the contents of `Listbox` by tying it to a variable `_images`. We set up `Listbox` so that `_img_frame` is its parent, and `_images` is the variable it's tied to. We also pass some dimensions.

The interesting bit comes from the `_scrollbar` instance. Note that, when we instantiate it, we set its command to `_img_listbox.yview`. This is the first half of the contract between a `Listbox` and a `Scrollbar`. The other half is provided by the `_img_listbox.configure` method, which sets the `yscrollcommand=_scrollbar.set`.

By providing this reciprocal bond, when we scroll on `Listbox`, the `Scrollbar` will move accordingly and vice-versa, when we operate the `Scrollbar`, the Listbox will scroll accordingly.

```
_radio_frame = ttk.Frame(_img_frame)
_radio_frame.grid(row=0, column=2, sticky=(N, S, W, E))
```

We place the **Radio Frame**, ready to be populated. Note that the `Listbox` is occupying (0, 0) on `_img_frame`, the `Scrollbar` (0, 1) and therefore `_radio_frame` will go in (0, 2).

```
_choice_lbl = ttk.Label(
    _radio_frame, text="Choose how to save images")
_choice_lbl.grid(row=0, column=0, padx=5, pady=5)
_save_method = StringVar()
_save_method.set('img')
_img_only_radio = ttk.Radiobutton(
    _radio_frame, text='As Images', variable=_save_method,
    value='img')
_img_only_radio.grid(
    row=1, column=0, padx=5, pady=2, sticky=W)
_img_only_radio.configure(state='normal')
```

```
_json_radio = ttk.Radiobutton(
    _radio_frame, text='As JSON', variable=_save_method,
    value='json')
_json_radio.grid(row=2, column=0, padx=5, pady=2, sticky=W)
```

Firstly, we place the label, and we give it some padding. Note that the label and radio buttons are children of `_radio_frame`.

As for the `Entry` and `Listbox` objects, the `Radiobutton` is also driven by a bond to an external variable, which I called `_save_method`. Each `Radiobutton` instance sets a value argument, and by checking the value on `_save_method`, we know which button is selected.

```
_scrape_btn = ttk.Button(
    _mainframe, text='Scrape!', command=save)
_scrape_btn.grid(row=2, column=0, sticky=E, pady=5)
```

On the third row of `_mainframe` we place the **Scrape** button. Its `command` is `save`, which saves the images to be listed in `Listbox`, after we have successfully parsed a web page.

```
_status_frame = ttk.Frame(
    _root, relief='sunken', padding='2 2 2 2')
_status_frame.grid(row=1, column=0, sticky=(E, W, S))
_status_msg = StringVar()
_status_msg.set('Type a URL to start scraping...')
_status = ttk.Label(
    _status_frame, textvariable=_status_msg, anchor=W)
_status.grid(row=0, column=0, sticky=(E, W))
```

We end the layout section by placing down the status frame, which is a simple `ttk.Frame`. To give it a little status bar effect, we set its `relief` property to `'sunken'` and give it a uniform padding of 2 pixels. It needs to stick to the `_root` window left, right and bottom parts, so we set its `sticky` attribute to `(E, W, S)`.

We then place a label in it and, this time, we tie it to a `StringVar` object, because we will have to modify it every time we want to update the status bar text. You should be acquainted to this technique by now.

Finally, on the last line, we run the application by calling the `mainloop` method on the `Tk` instance.

```
_root.mainloop()
```

Please remember that all these instructions are placed under the `if __name__ == "__main__":` clause in the original script.

As you can see, the code to design our GUI application is not hard. Granted, at the beginning you have to play around a little bit. Not everything will work out perfectly at the first attempt, but I promise you it's very easy and you can find plenty of tutorials on the web. Let's now get to the interesting bit, the business logic.

The business logic

We'll analyze the business logic of the GUI application in three chunks. There is the fetching logic, the saving logic, and the alerting logic.

Fetching the web page

```
config = {}

def fetch_url():
    url = _url.get()
    config['images'] = []
    _images.set(())   # initialized as an empty tuple
    try:
        page = requests.get(url)
    except requests.RequestException as rex:
        _sb(str(rex))
    else:
        soup = BeautifulSoup(page.content, 'html.parser')
        images = fetch_images(soup, url)
        if images:
            _images.set(tuple(img['name'] for img in images))
            _sb('Images found: {}'.format(len(images)))
        else:
            _sb('No images found')
        config['images'] = images

def fetch_images(soup, base_url):
    images = []
    for img in soup.findAll('img'):
        src = img.get('src')
        img_url = (
            '{base_url}/{src}'.format(base_url=base_url, src=src))
        name = img_url.split('/')[-1]
        images.append(dict(name=name, url=img_url))
    return images
```

First of all, let me explain that `config` dictionary. We need some way of passing data between the GUI application and the business logic. Now, instead of polluting the global namespace with many different variables, my personal preference is to have a single dictionary that holds all the objects we need to pass back and forth, so that the global namespace isn't be clogged up with all those names, and we have one single, clean, easy way of knowing where all the objects that are needed by our application are.

In this simple example, we'll just populate the `config` dictionary with the images we fetch from the page, but I wanted to show you the technique so that you have at least an example. This technique comes from my experience with JavaScript. When you code a web page, you very often import several different libraries. If each of these cluttered the global namespace with all sorts of variables, there would be severe issues in making everything work, because of name clashes and variable overriding. They make the coder's life a living hell.

So, it's much better to try and leave the global namespace as clean as we can. In this case, I find that using one `config` variable is more than acceptable.

The `fetch_url` function is quite similar to what we did in the script. Firstly, we get the `url` value by calling `_url.get()`. Remember that the `_url` object is a `StringVar` instance that is tied to the `_url_entry` object, which is an `Entry`. The text field you see on the GUI is the `Entry`, but the text behind the scenes is the value of the `StringVar` object.

By calling `get()` on `_url`, we get the value of the text which is displayed in `_url_entry`.

The next step is to prepare `config['images']` to be an empty list, and to empty the `_images` variable, which is tied to `_img_listbox`. This, of course, has the effect of cleaning up all the items in `_img_listbox`.

After this preparation step, we can try to fetch the page, using the same `try/except` logic we adopted in the script at the beginning of the chapter.

The one difference is in the action we take if things go wrong. We call `_sb(str(rex))`. `_sb` is a helper function whose code we'll see shortly. Basically, it sets the text in the status bar for us. Not a good name, right? I had to explain its behavior to you: food for thought.

If we can fetch the page, then we create the `soup` instance, and fetch the images from it. The logic of `fetch_images` is exactly the same as the one explained before, so I won't repeat myself here.

If we have images, using a quick tuple comprehension (which is actually a generator expression fed to a tuple constructor) we feed the `_images` `StringVar` and this has the effect of populating our `_img_listbox` with all the image names. Finally, we update the status bar.

If there were no images, we still update the status bar, and at the end of the function, regardless of how many images were found, we update `config['images']` to hold the `images` list. In this way, we'll be able to access the images from other functions by inspecting `config['images']` without having to pass that list around.

Saving the images

The logic to save the images is pretty straightforward. Here it is:

```
def save():
    if not config.get('images'):
        _alert('No images to save')
        return

    if _save_method.get() == 'img':
        dirname = filedialog.askdirectory(mustexist=True)
        _save_images(dirname)
    else:
        filename = filedialog.asksaveasfilename(
            initialfile='images.json',
            filetypes=[('JSON', '.json')])
        _save_json(filename)

def _save_images(dirname):
    if dirname and config.get('images'):
        for img in config['images']:
            img_data = requests.get(img['url']).content
            filename = os.path.join(dirname, img['name'])
            with open(filename, 'wb') as f:
                f.write(img_data)
        _alert('Done')

def _save_json(filename):
    if filename and config.get('images'):
        data = {}
        for img in config['images']:
            img_data = requests.get(img['url']).content
            b64_img_data = base64.b64encode(img_data)
```

```
            str_img_data = b64_img_data.decode('utf-8')
            data[img['name']] = str_img_data

    with open(filename, 'w') as ijson:
        ijson.write(json.dumps(data))
    _alert('Done')
```

When the user clicks the **Scrape** button, the `save` function is called using the callback mechanism.

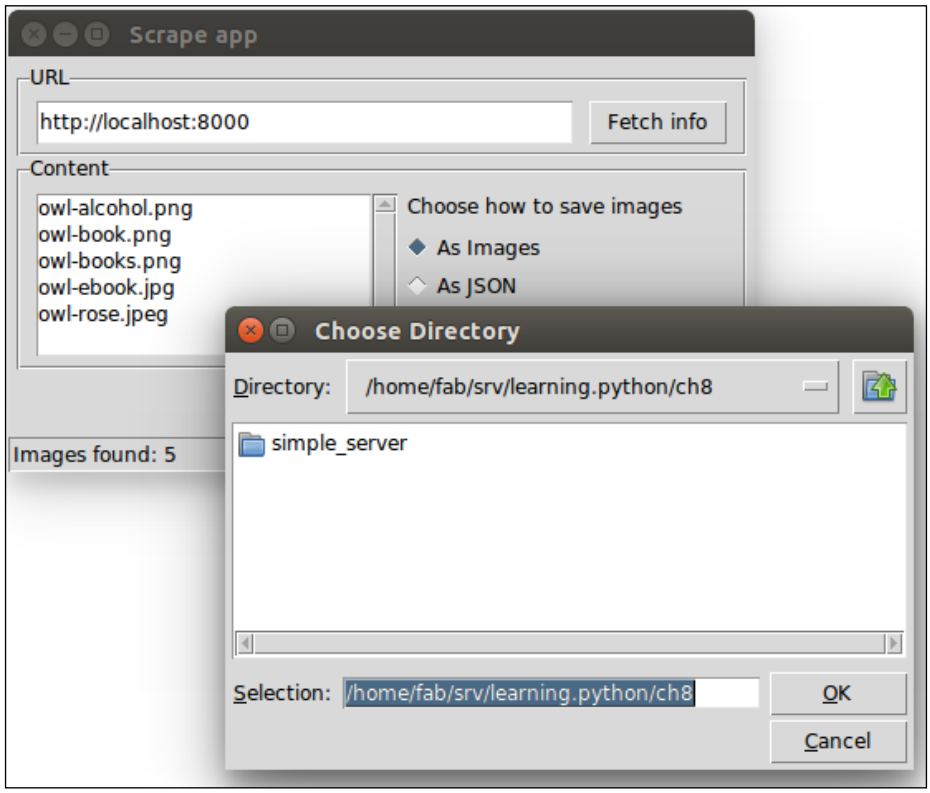
The first thing that this function does is check whether there are actually any images to be saved. If not, it alerts the user about it, using another helper function, `_alert`, whose code we'll see shortly. No further action is performed if there are no images.

On the other hand, if the `config['images']` list is not empty, `save` acts as a dispatcher, and it calls `_save_images` or `_save_json`, according to which value is held by `_same_method`. Remember, this variable is tied to the radio buttons, therefore we expect its value to be either `'img'` or `'json'`.

This dispatcher is a bit different from the one in the script. According to which method we have selected, a different action must be taken.

If we want to save the images as images, we need to ask the user to choose a directory. We do this by calling `filedialog.askdirectory` and assigning the result of the call to the variable `dirname`. This opens up a nice dialog window that asks us to choose a directory. The directory we choose must exist, as specified by the way we call the method. This is done so that we don't have to write code to deal with a potentially missing directory when saving the files.

Here's how this dialog should look on Ubuntu:



If we cancel the operation, `dirname` will be set to `None`.

Before finishing analyzing the logic in `save`, let's quickly go through `_save_images`.

It's very similar to the version we had in the script so just note that, at the beginning, in order to be sure that we actually have something to do, we check on both `dirname` and the presence of at least one image in `config['images']`.

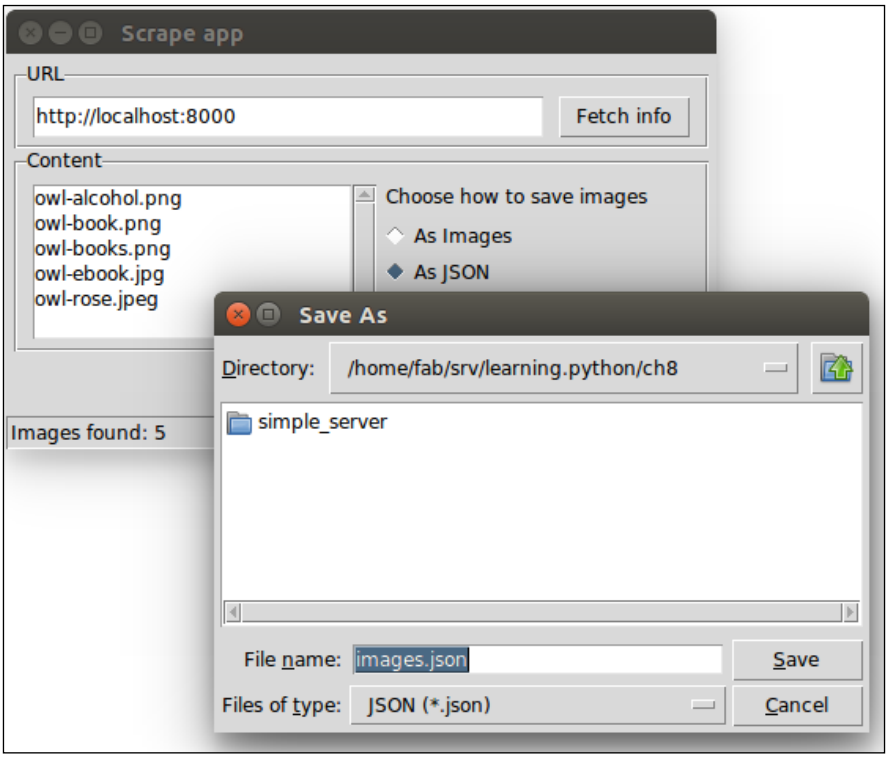
If that's the case, it means we have at least one image to save and the path for it, so we can proceed. The logic to save the images has already been explained. The one thing we do differently this time is to join the directory (which means the complete path) to the image name, by means of `os.path.join`. In the `os.path` module there's plenty of useful methods to work with paths and filenames.

At the end of `_save_images`, if we saved at least one image, we alert the user that we're done.

Let's go back now to the other branch in `save`. This branch is executed when the user selects the **As JSON** radio button before pressing the **Scrape** button. In this case, we want to save a file; therefore, we cannot just ask for a directory. We want to give the user the ability to choose a filename as well. Hence, we fire up a different dialog: `filedialog.asksaveasfilename`.

We pass an initial filename, which is proposed to the user with the ability to change it if they don't like it. Moreover, because we're saving a JSON file, we're forcing the user to use the correct extension by passing the `filetypes` argument. It is a list with any number of 2-tuples *(description, extension)* that runs the logic of the dialog.

Here's how this dialog should look on Ubuntu:



Once we have chosen a place and a filename, we can proceed with the saving logic, which is the same as it was in the previous script. We create a JSON object from a Python dictionary (`data`) that we populate with key/value pairs made by the `images` name and Base64 encoded content.

In `_save_json` as well, we have a little check at the beginning that makes sure that we don't proceed unless we have a file name and at least one image to save.

This ensures that if the user presses the **Cancel** button, nothing bad happens.

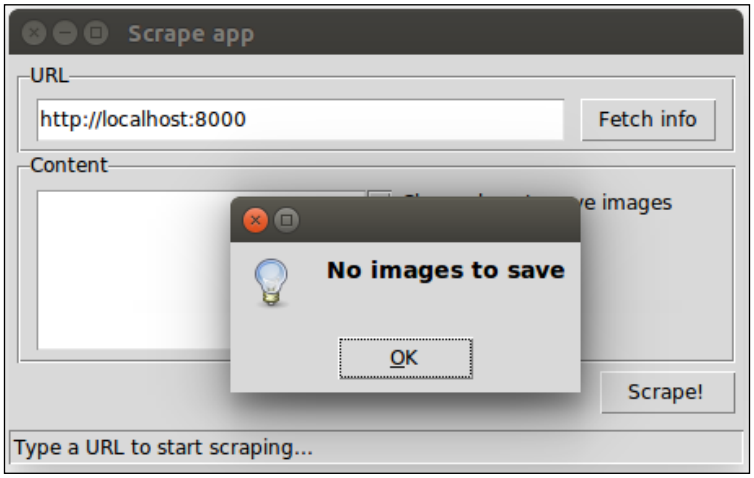
Alerting the user

Finally, let's see the alerting logic. It's extremely simple.

```
def _sb(msg):
    _status_msg.set(msg)

def _alert(msg):
    messagebox.showinfo(message=msg)
```

That's it! To change the status bar message all we need to do is to access `_status_msg` `StringVar`, as it's tied to the `_status` label.

On the other hand, if we want to show the user a more visible message, we can fire up a message box. Here's how it should look on Ubuntu:



The `messagebox` object can also be used to warn the user (`messagebox.showwarning`) or to signal an error (`messagebox.showerror`). But it can also be used to provide dialogs that ask us if we're sure that we want to proceed or if we really want to delete that file, and so on.

If you inspect `messagebox` by simply printing out what `dir(messagebox)` returns, you'll find methods like `askokcancel`, `askquestion`, `askretrycancel`, `askyesno`, and `askyesnocancel`, as well as a set of constants to verify the response of the user, such as `CANCEL`, `NO`, `OK`, `OKCANCEL`, `YES`, `YESNOCANCEL`, and so on. You can compare these to the user's choice so that you know what the next action to execute when the dialog closes.

How to improve the application?

Now that you're accustomed to the fundamentals of designing a GUI application, I'd like to give you some suggestions on how to make ours better.

We can start from the code quality. Do you think this code is good enough, or would you improve it? If so, how? I would test it, and make sure it's robust and caters for all the various scenarios that a user might create by clicking around on the application. I would also make sure the behavior is what I would expect when the website we're scraping is down for any reason.

Another thing that we could improve is the naming. I have prudently named all the components with a leading underscore, both to highlight their somewhat "private" nature, and to avoid having name clashes with the underlying objects they are linked to. But in retrospect, many of those components could use a better name, so it's really up to you to refactor until you find the form that suits you best. You could start by giving a better name to the `_sb` function!

For what concerns the user interface, you could try and resize the main application. See what happens? The whole content stays exactly where it is. Empty space is added if you expand, or the whole widgets set disappears gradually if you shrink. This behavior isn't exactly nice, therefore one quick solution could be to make the root window fixed (that is, unable to resize).

Another thing that you could do to improve the application is to add the same functionality we had in the script, to save only PNGs or JPGs. In order to do this, you could place a combo box somewhere, with three values: All, PNGs, JPGs, or something similar. The user should be able to select one of those options before saving the files.

Even better, you could change the declaration of `Listbox` so that it's possible to select multiple images at the same time, and only the selected ones will be saved. If you manage to do this (it's not as hard as it seems, believe me), then you should consider presenting the `Listbox` a bit better, maybe providing alternating background colors for the rows.

Another nice thing you could add is a button that opens up a dialog to select a file. The file must be one of the JSON files the application can produce. Once selected, you could run some logic to reconstruct the images from their Base64-encoded version. The logic to do this is very simple, so here's an example:

```
with open('images.json', 'r') as f:
    data = json.loads(f.read())

for (name, b64val) in data.items():
    with open(name, 'wb') as f:
        f.write(base64.b64decode(b64val))
```

As you can see, we need to open `images.json` in read mode, and grab the `data` dictionary. Once we have it, we can loop through its items, and save each image with the Base64 decoded content. I'll leave it up to you to tie this logic to a button in the application.

Another cool feature that you could add is the ability to open up a preview pane that shows any image you select from the `Listbox`, so that the user can take a peek at the images before deciding to save them.

Finally, one last suggestion for this application is to add a menu. Maybe even a simple menu with **File** and **?** to provide the usual **Help** or **About**. Just for fun. Adding menus is not that complicated; you can add text, keyboard shortcuts, images, and so on.

Where do we go from here?

If you are interested in digging deeper into the world of GUIs, then I'd like to offer you the following suggestions.

The tkinter.tix module

Exploring `tkinter` and its themed widget set, `tkinter.ttk`, will take you some time. There's much to learn and play with. Another interesting module to explore, when you'll be familiar with this technology, is `tkinter.tix`.

The `tkinter.tix` (**Tk Interface Extension**) module provides an additional very rich set of widgets. The need for them stems from the fact that the widgets in the standard `Tk` library are far from complete.

The `tkinter.tix` library allows us to solve this problem by providing widgets like HList, ComboBox, Control (or SpinBox), and various scrollable widgets. Altogether, there are more than 40 widgets. They allow you to introduce different interaction techniques and paradigms into your applications, thus improving their quality and usability.

The turtle module

The `turtle` module is an extended reimplementation of the eponymous module from the Python standard distribution up to version Python 2.5. It's a very popular way to introduce children to programming.

It's based on the idea of an imaginary turtle starting at (0, 0) in the Cartesian plane. You can programmatically command the turtle to move forward and backwards, rotate, and so on. and by combining together all the possible moves, all sorts of intricate shapes and images can be drawn.

It's definitely worth checking out, if only to see something different.

wxPython, PyQt, and PyGTK

After you have explored the vastness of the `tkinter` realm, I'd suggest you to explore other GUI libraries: **wxPython**, **PyQt**, and **PyGTK**. You may find out one of these works better for you, or it makes easier for you to code the application you need.

I believe that coders can realize their ideas only when they are conscious about what tools they have available. If your toolset is too narrow, your ideas may seem impossible or extremely hard to realize, and they risk remaining exactly what they are, just ideas.

Of course, the technological spectrum today is humongous, so knowing everything is not possible; therefore, when you are about to learn a new technology or a new subject, my suggestion is to grow your knowledge by exploring breadth first.

Investigate several things not too deeply, and then go deep with the one or the few that looked most promising. This way you'll be able to be productive with at least one tool, and when the tool no longer fits your needs, you'll know where to dig deeper, thanks to your previous exploration.

The principle of least astonishment

When designing an interface, there are many different things to bear in mind. One of them, which for me is the most important, is the law or **principle of least astonishment**. It basically states that if in your design a necessary feature has a high astonishing factor, it may be necessary to redesign your application. To give you one example, when you're used to working with Windows, where the buttons to minimize, maximize and close a window are on the top-right corner, it's quite hard to work on Linux, where they are at the top-left corner. You'll find yourself constantly going to the top-right corner only to discover once more that the buttons are on the other side.

If a certain button has become so important in applications that it's now placed in a precise location by designers, please don't innovate. Just follow the convention. Users will only become frustrated when they have to waste time looking for a button that is not where it's supposed to be.

The disregard for this rule is the reason why I cannot work with products like Jira. It takes me minutes to do simple things that should require seconds.

Threading considerations

This topic is beyond the scope of an introductory book like this, but I do want to mention it. In a nutshell, a **thread** of execution is the smallest sequence of programmed instructions that can be managed independently by a **scheduler**. The reason we have the perception that modern computers can do many things at the same time is not only due to the fact that they have multiple processors. They also subdivide the work in different threads, which are then worked on in sequence. If their lifecycle is sufficiently short, threads can be worked on in one single go, but typically, what happens is that the OS works on a thread for a little time, then switches to another one, then to another one, then back to the first one, and so on. The order in which they are worked on depends on different factors. The end result is that, because computers are extremely fast in doing this switching, we perceive many things happening at the same time.

If you are coding a GUI application that needs to perform a long running operation when a button is clicked, you will see that your application will probably freeze until the operation has been carried out. In order to avoid this, and maintain the application's responsiveness, you may need to run that time-expensive operation in a different thread so that the OS will be able to dedicate a little bit of time to the GUI every now and then, to keep it responsive.

Threads are an advanced topic, especially in Python. Gain a good grasp of the fundamentals first, and then have fun exploring them!

Summary

In this chapter, we worked on a project together. We have written a script that scrapes a very simple web page and accepts optional commands that alter its behavior in doing so. We also coded a GUI application to do the same thing by clicking buttons instead of typing on a console. I hope you enjoyed reading it and following along as much as I enjoyed writing it.

We saw many different concepts like context managers, working with files, performing HTTP requests, and we've talked about guidelines for usability and design.

I have only been able to scratch the surface, but hopefully, you have a good starting point from which to expand your exploration.

Throughout the chapter, I have pointed you in several different ways on how to improve the application, and I have challenged you with a few exercises and questions. I hope you have taken the time to play with those ideas. One can learn a lot just by playing around with fun applications like the one we've coded together.

In the next chapter, we're going to talk about data science, or at least about the tools that a Python programmer has when it comes to facing this subject.

9
Data Science

"If we have data, let's look at data. If all we have are opinions, let's go with mine."

Jim Barksdale, former Netscape CEO

Data science is a very broad term, and can assume several different meanings according to context, understanding, tools, and so on. There are countless books about this subject, which is not suitable for the faint-hearted.

In order to do proper data science, you need to know mathematics and statistics at the very least. Then, you may want to dig into other subjects such as pattern recognition and machine learning and, of course, there is a plethora of languages and tools you can choose from.

Unless I transform into *The Amazing Fabrizio* in the next few minutes, I won't be able to talk about everything; I won't even get close to it. Therefore, in order to render this chapter meaningful, we're going to work on a cool project together.

About 3 years ago, I was working for a top-tier social media company in London. I stayed there for 2 years, and I was privileged to work with several people whose brilliance I can only start to describe. We were the first in the world to have access to the Twitter Ads API, and we were partners with Facebook as well. That means a lot of data.

Our analysts were dealing with a huge number of campaigns and they were struggling with the amount of work they had to do, so the development team I was a part of tried to help by introducing them to Python and to the tools Python gives you to deal with data. It was a very interesting journey that led me to mentor several people in the company and eventually to Manila where, for 2 weeks, I gave intensive training in Python and data science to our analysts there.

The project we're going to do together in this chapter is a lightweight version of the final example I presented to my Manila students. I have rewritten it to a size that will fit this chapter, and made a few adjustments here and there for teaching purposes, but all the main concepts are there, so it should be fun and instructional for you to code along.

On our journey, we're going to meet a few of the tools you can find in the Python ecosystem when it comes to dealing with data, so let's start by talking about Roman gods.

IPython and Jupyter notebook

In 2001, Fernando Perez was a graduate student in physics at CU Boulder, and was trying to improve the Python shell so that he could have some niceties like those he was used to when he was working with tools such as Mathematica and Maple. The result of that effort took the name **IPython**.

In a nutshell, that small script began as an enhanced version of the Python shell and, through the effort of other coders and eventually proper funding from several different companies, it became the wonderful and successful project it is today. Some 10 years after its birth, a notebook environment was created, powered by technologies like WebSockets, the Tornado web server, jQuery, CodeMirror, and MathJax. The ZeroMQ library was also used to handle the messages between the notebook interface and the Python core that lies behind it.

The IPython notebook has become so popular and widely used that eventually, all sorts of goodies have been added to it. It can handle widgets, parallel computing, all sorts of media formats, and much, much more. Moreover, at some point, it became possible to code in languages other than Python from within the notebook.

This has led to a huge project that only recently has been split into two: IPython has been stripped down to focus more on the kernel part and the shell, while the notebook has become a brand new project called **Jupyter**. Jupyter allows interactive scientific computations to be done in more than 40 languages.

This chapter's project will all be coded and run in a Jupyter notebook, so let me explain in a few words what a notebook is.

A notebook environment is a web page that exposes a simple menu and the cells in which you can run Python code. Even though the cells are separate entities that you can run individually, they all share the same Python kernel. This means that all the names that you define in a cell (the variables, functions, and so on) will be available in any other cell.

Simply put, a Python kernel is a process in which Python is running. The notebook web page is therefore an interface exposed to the user for driving this kernel. The web page communicates to it using a very fast messaging system.

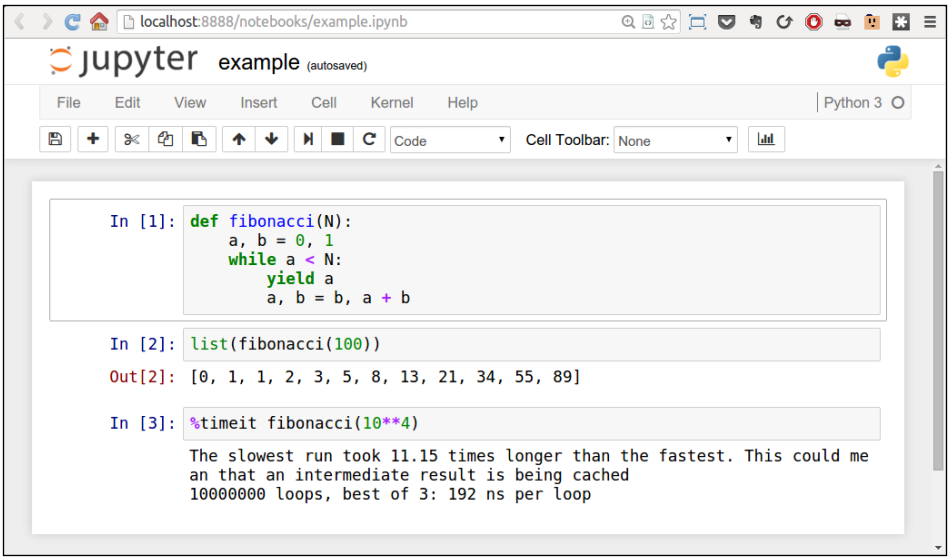
Apart from all the graphical advantages, the beauty to have such an environment consists in the ability of running a Python script in chunks, and this can be a tremendous advantage. Take a script that is connecting to a database to fetch data and then manipulate that data. If you do it in the conventional way, with a Python script, you have to fetch the data every time you want to experiment with it. Within a notebook environment, you can fetch the data in a cell and then manipulate and experiment with it in other cells, so fetching it every time is not necessary.

The notebook environment is also extremely helpful for data science because it allows for step-by-step introspection. You do one chunk of work and then verify it. You then do another chunk and verify again, and so on.

It's also invaluable for prototyping because the results are there, right in front of your eyes, immediately available.

If you want to know more about these tools, please check out `http://ipython.org/` and `http://jupyter.org/`.

I have created a very simple example notebook with a `fibonacci` function that gives you the list of all Fibonacci numbers smaller than a given `N`. In my browser, it looks like this:



Every cell has an **In []** label. If there's nothing between the braces, it means that cell has never been executed. If there is a number, it means that the cell has been executed, and the number represents the order in which the cell was executed. Finally, a * means that the cell is currently being executed.

You can see in the picture that in the first cell I have defined the `fibonacci` function, and I have executed it. This has the effect of placing the `fibonacci` name in the global frame associated with the notebook, therefore the `fibonacci` function is now available to the other cells as well. In fact, in the second cell, I can run `fibonacci(100)` and see the results in **Out [2]**. In the third cell, I have shown you one of the several magic functions you can find in a notebook in the second cell. %**timeit** runs the code several times and provides you with a nice benchmark for it. All the measurements for the list comprehensions and generators I did in *Chapter 5*, *Saving Time and Memory* were carried out with this nice feature.

You can execute a cell as many times as you want, and change the order in which you run them. Cells are very malleable, you can also put in markdown text or render them as headers.

Markdown is a lightweight markup language with plain text formatting syntax designed so that it can be converted to HTML and many other formats.

Also, whatever you place in the last row of a cell will be automatically printed for you. This is very handy because you're not forced to write `print(...)` explicitly.

Feel free to explore the notebook environment; once you're friends with it, it's a long-lasting relationship, I promise.

In order to run the notebook, you have to install a handful of libraries, each of which collaborates with the others to make the whole thing work. Alternatively, you can just install Jupyter and it will take care of everything for you. For this chapter, there are a few other dependencies that we need to install, so please run the following command:

```
$ pip install jupyter pandas matplotlib fake-factory delorean xlwt
```

Don't worry, I'll introduce you to each of these gradually. Now, when you're done installing these libraries (it may take a few minutes), you can start the notebook:

```
$ jupyter notebook
```

This will open a page in your browser at this address: `http://localhost:8888/`.

Go to that page and create a new notebook using the menu. When you have it and you're comfortable with it, we're ready to go.

If you experience any issues setting up the notebook environment, please don't get discouraged. If you get an error, it's usually just a matter of searching a little bit on the web and you'll end up on a page where someone else has had the same issue, and they have explained how to fix it. Try your best to have the notebook environment up and running before continuing with the chapter.

Our project will take place in a notebook, therefore I will tag each code snippet with the cell number it belongs to, so that you can easily reproduce the code and follow along.

If you familiarize yourself with the keyboard shortcuts (look in the notebook's help section), you will be able to move between cells and handle their content without having to reach for the mouse. This will make you more proficient and way faster when you work in a notebook.

Dealing with data

Typically, when you deal with data, this is the path you go through: you fetch it, you clean and manipulate it, then you inspect it and present results as values, spreadsheets, graphs, and so on. I want you to be in charge of all three steps of the process without having any external dependency on a data provider, so we're going to do the following:

1. We're going to create the data, simulating the fact that it comes in a format which is not perfect or ready to be worked on.
2. We're going to clean it and feed it to the main tool we'll use in the project: **DataFrame** of `pandas`.
3. We're going to manipulate the data in the DataFrame.
4. We're going to save the DataFrame to a file in different formats.
5. Finally, we're going to inspect the data and get some results out of it.

Setting up the notebook

First things first, we need to set up the notebook. This means imports and a bit of configuration.

```
#1

    import json
    import calendar
    import random
    from datetime import date, timedelta

    import faker
    import numpy as np
    from pandas import DataFrame
    from delorean import parse
    import pandas as pd

    # make the graphs nicer
    pd.set_option('display.mpl_style', 'default')
```

Cell `#1` takes care of the imports. There are quite a few new things here: the `calendar`, `random` and `datetime` modules are part of the standard library. Their names are self-explanatory, so let's look at `faker`. The `fake-factory` library gives you this module, which you can use to prepare fake data. It's very useful in tests, when you prepare your fixtures, to get all sorts of things such as names, e-mail addresses, phone numbers, credit card details, and much more. It is all fake, of course.

`numpy` is the NumPy library, the fundamental package for scientific computing with Python. I'll spend a few words on it later on in the chapter.

`pandas` is the very core upon which the whole project is based. It stands for **Python Data Analysis Library**. Among many others, it provides the **DataFrame**, a matrix-like data structure with advanced processing capabilities. It's customary to import the `DataFrame` separately and then do `import pandas as pd`.

`delorean` is a nice third-party library that speeds up dealing with dates dramatically. Technically, we could do it with the standard library, but I see no reason not to expand a bit the range of the example and show you something different.

Finally, we have an instruction on the last line that will make our graphs at the end a little bit nicer, which doesn't hurt.

Preparing the data

We want to achieve the following data structure: we're going to have a list of user objects. Each user object will be linked to a number of campaign objects.

In Python, everything is an object, so I'm using this term in a generic way. The user object may be a string, a dict, or something else.

A **campaign** in the social media world is a promotional campaign that a media agency runs on social media networks on behalf of a client.

Remember that we're going to prepare this data so that it's not in perfect shape (but it won't be so bad either...).

```
#2

    fake = faker.Faker()
```

Firstly, we instantiate the `Faker` that we'll use to create the data.

```
#3

    usernames = set()
    usernames_no = 1000
    # populate the set with 1000 unique usernames
    while len(usernames) < usernames_no:
        usernames.add(fake.user_name())
```

Then we need usernames. I want 1,000 unique usernames, so I loop over the length of the `usernames` set until it has 1,000 elements. A set doesn't allow duplicated elements, therefore uniqueness is guaranteed.

```
#4

    def get_random_name_and_gender():
        skew = .6  # 60% of users will be female
        male = random.random() > skew
        if male:
            return fake.name_male(), 'M'
        else:
            return fake.name_female(), 'F'

    def get_users(usernames):
        users = []
        for username in usernames:
```

```
            name, gender = get_random_name_and_gender()
            user = {
                'username': username,
                'name': name,
                'gender': gender,
                'email': fake.email(),
                'age': fake.random_int(min=18, max=90),
                'address': fake.address(),
            }
            users.append(json.dumps(user))
        return users

    users = get_users(usernames)
    users[:3]
```

Here, we create a list of users. Each `username` has now been augmented to a full-blown `user` dict, with other details such as name, gender, e-mail, and so on. Each `user` dict is then dumped to JSON and added to the list. This data structure is not optimal, of course, but we're simulating a scenario where users come to us like that.

Note the skewed use of `random.random()` to make 60% of users female. The rest of the logic should be very easy for you to understand.

Note also the last line. Each cell automatically prints what's on the last line; therefore, the output of this is a list with the first three users:

```
Out #4
```

```
['{"gender": "F", "age": 48, "email": "jovani.dickinson@gmail.
com", "address": "2006 Sawayn Trail Apt. 207\\nHyattview, MO 27278",
"username": "darcy00", "name": "Virgia Hilpert"}',
 '{"gender": "F", "age": 58, "email": "veum.javen@hotmail.com",
"address": "5176 Andres Plains Apt. 040\\nLakinside, GA 92446",
"username": "renner.virgie", "name": "Miss Clarabelle Kertzmann MD"}',
 '{"gender": "M", "age": 33, "email": "turner.felton@rippin.com",
"address": "1218 Jacobson Fort\\nNorth Doctor, OK 04469", "username":
"hettinger.alphonsus", "name": "Ludwig Prosacco"}']
```

I hope you're following along with your own notebook. If you do, please note that all data is generated using random functions and values; therefore, you will see different results. They will change every time you execute the notebook.

#5

```
    # campaign name format:
    # InternalType_StartDate_EndDate_TargetAge_TargetGender_Currency
    def get_type():
        # just some gibberish internal codes
        types = ['AKX', 'BYU', 'GRZ', 'KTR']
        return random.choice(types)

    def get_start_end_dates():
        duration = random.randint(1, 2 * 365)
        offset = random.randint(-365, 365)
        start = date.today() - timedelta(days=offset)
        end = start + timedelta(days=duration)

        def _format_date(date_):
            return date_.strftime("%Y%m%d")

        return _format_date(start), _format_date(end)

    def get_age():
        age = random.randint(20, 45)
        age -= age % 5
        diff = random.randint(5, 25)
        diff -= diff % 5
        return '{}-{}'.format(age, age + diff)

    def get_gender():
        return random.choice(('M', 'F', 'B'))

    def get_currency():
        return random.choice(('GBP', 'EUR', 'USD'))

    def get_campaign_name():
        separator = '_'
        type_ = get_type()
        start_end = separator.join(get_start_end_dates())
        age = get_age()
        gender = get_gender()
        currency = get_currency()
        return separator.join(
            (type_, start_end, age, gender, currency))
```

In `#5`, we define the logic to generate a campaign name. Analysts use spreadsheets all the time and they come up with all sorts of coding techniques to compress as much information as possible into the campaign names. The format I chose is a simple example of that technique: there is a code that tells the campaign type, then start and end dates, then the target age and gender, and finally the currency. All values are separated by an underscore.

In the `get_type` function, I use `random.choice()` to get one value randomly out of a collection. Probably more interesting is `get_start_end_dates`. First, I get the duration for the campaign, which goes from 1 day to 2 years (randomly), then I get a random offset in time which I subtract from today's date in order to get the start date. Given that the offset is a random number between -365 and 365, would anything be different if I added it to today's date instead of subtracting it?

When I have both the start and end dates, I return a stringified version of them, joined by an underscore.

Then, we have a bit of modular trickery going on with the age calculation. I hope you remember the modulo operator (`%`) from *Chapter 2*, *Built-in Data Types*.

What happens here is that I want a date range that has multiples of 5 as extremes. So, there are many ways to do it, but what I do is to get a random number between 20 and 45 for the left extreme, and remove the remainder of the division by 5. So, if, for example, I get 28, I will remove *28 % 5 = 3* to it, getting 25. I could have just used `random.randrange()`, but it's hard to resist modular division.

The rest of the functions are just some other applications of `random.choice()` and the last one, `get_campaign_name`, is nothing more than a collector for all these puzzle pieces that returns the final campaign name.

```
#6

    def get_campaign_data():
        name = get_campaign_name()
        budget = random.randint(10**3, 10**6)
        spent = random.randint(10**2, budget)
        clicks = int(random.triangular(10**2, 10**5, 0.2 * 10**5))
        impressions = int(random.gauss(0.5 * 10**6, 2))
        return {
            'cmp_name': name,
            'cmp_bgt': budget,
            'cmp_spent': spent,
            'cmp_clicks': clicks,
            'cmp_impr': impressions
        }
```

In `#6`, we write a function that creates a complete campaign object. I used a few different functions from the `random` module. `random.randint()` gives you an integer between two extremes. The problem with it is that it follows a uniform probability distribution, which means that any number in the interval has the same probability of coming up.

Therefore, when dealing with a lot of data, if you distribute your fixtures using a uniform distribution, the results you will get will all look similar. For this reason, I chose to use `triangular` and `gauss`, for `clicks` and `impressions`. They use different probability distributions so that we'll have something more interesting to see in the end.

Just to make sure we're on the same page with the terminology: `clicks` represents the number of clicks on a campaign advertisement, `budget` is the total amount of money allocated for the campaign, `spent` is how much of that money has already been spent, and `impressions` is the number of times the campaign has been fetched, as a resource, from its source, regardless of the amount of clicks that were performed on the campaign. Normally, the amount of impressions is greater than the amount of clicks.

Now that we have the data, it's time to put it all together:

`#7`

```
def get_data(users):
    data = []
    for user in users:
        campaigns = [get_campaign_data()
                     for _ in range(random.randint(2, 8))]
        data.append({'user': user, 'campaigns': campaigns})
    return data
```

As you can see, each item in `data` is a dict with a user and a list of campaigns that are associated with that user.

Cleaning the data

Let's start cleaning the data:

`#8`

```
rough_data = get_data(users)
rough_data[:2]  # let's take a peek
```

We simulate fetching the data from a source and then inspect it. The notebook is the perfect tool to inspect your steps. You can vary the granularity to your needs. The first item in `rough_data` looks like this:

```
[{'campaigns': [{'cmp_bgt': 130532,
    'cmp_clicks': 25576,
    'cmp_impr': 500001,
    'cmp_name': 'AKX_20150826_20170305_35-50_B_EUR',
    'cmp_spent': 57574},
   ... omit ...
   {'cmp_bgt': 884396,
    'cmp_clicks': 10955,
    'cmp_impr': 499999,
    'cmp_name': 'KTR_20151227_20151231_45-55_B_GBP',
    'cmp_spent': 318887}],
  'user': '{"age": 44, "username": "jacob43",
            "name": "Holland Strosin",
            "email": "humberto.leuschke@brakus.com",
            "address": "1038 Runolfsdottir Parks\\nElmapo...",
            "gender": "M"}'}]
```

So, we now start working with it.

```
#9

    data = []
    for datum in rough_data:
        for campaign in datum['campaigns']:
            campaign.update({'user': datum['user']})
            data.append(campaign)
    data[:2]  # let's take another peek
```

The first thing we need to do in order to be able to feed a DataFrame with this data is to denormalize it. This means transforming the data into a list whose items are campaign dicts, augmented with their relative user dict. Users will be duplicated in each campaign they belong to. The first item in `data` looks like this:

```
[{'cmp_bgt': 130532,
  'cmp_clicks': 25576,
  'cmp_impr': 500001,
  'cmp_name': 'AKX_20150826_20170305_35-50_B_EUR',
```

```
'cmp_spent': 57574,
'user': '{"age": 44, "username": "jacob43",
          "name": "Holland Strosin",
          "email": "humberto.leuschke@brakus.com",
          "address": "1038 Runolfsdottir Parks\\nElmaport...",
          "gender": "M"}'}]
```

You can see that the user object has been brought into the campaign dict which was repeated for each campaign.

Creating the DataFrame

Now it's time to create the `DataFrame`:

```
#10

    df = DataFrame(data)
    df.head()
```

Finally, we will create the `DataFrame` and inspect the first five rows using the `head` method. You should see something like this:

	cmp_bgt	cmp_clicks	cmp_impr	cmp_name	cmp_spent	user
0	130532	25576	500001	AKX_20150826_20170305_35-50_B_EUR	57574	{"age": 44, "username": "jacob43", "name": "Ho...
1	262852	61247	499999	AKX_20141015_20141219_40-50_F_GBP	226319	{"age": 44, "username": "jacob43", "name": "Ho...
2	12098	15582	500004	KTR_20150222_20160114_20-25_F_GBP	4354	{"age": 44, "username": "jacob43", "name": "Ho...
3	888381	35843	499998	KTR_20140918_20160416_35-50_B_EUR	472363	{"age": 44, "username": "jacob43", "name": "Ho...
4	361909	84759	499998	GRZ_20150726_20170615_25-35_M_GBP	257560	{"age": 44, "username": "jacob43", "name": "Ho...

Jupyter renders the output of the `df.head()` call as HTML automatically. In order to have a text-based output, simply wrap `df.head()` in a `print` call.

The `DataFrame` structure is very powerful. It allows us to do a great deal of manipulation on its contents. You can filter by rows, columns, aggregate on data, and many other operations. You can operate with rows or columns without suffering the time penalty you would have to pay if you were working on data with pure Python. This happens because, under the covers, `pandas` is harnessing the power of the NumPy library, which itself draws its incredible speed from the low-level implementation of its core. NumPy stands for **Numeric Python**, and it is one of the most widely used libraries in the data science environment.

Using a DataFrame allows us to couple the power of NumPy with spreadsheet-like capabilities so that we'll be able to work on our data in a fashion that is similar to what an analyst could do. Only, we do it with code.

But let's go back to our project. Let's see two ways to quickly get a bird's eye view of the data:

```
#11

    df.count()
```

`count` yields a count of all the non-empty cells in each column. This is good to help you understand how sparse your data can be. In our case, we have no missing values, so the output is:

```
cmp_bgt       4974
cmp_clicks    4974
cmp_impr      4974
cmp_name      4974
cmp_spent     4974
user          4974
dtype: int64
```

Nice! We have 4,974 rows, and the data type is integers (`dtype: int64` means long integers because they take 64 bits each). Given that we have 1,000 users and the amount of campaigns per user is a random number between 2 and 8, we're exactly in line with what I was expecting.

```
#12

    df.describe()
```

`describe` is a nice and quick way to introspect a bit further:

```
            cmp_bgt    cmp_clicks       cmp_impr      cmp_spent
count    4974.000000   4974.000000    4974.000000    4974.000000
mean   503272.706876  40225.764978  499999.495979  251150.604343
std    289393.747465  21910.631950       2.035355  220347.594377
min      1250.000000    609.000000  499992.000000     142.000000
25%    253647.500000  22720.750000  499998.000000   67526.750000
50%    508341.000000  36561.500000  500000.000000  187833.000000
75%    757078.250000  55962.750000  500001.000000  385803.750000
max    999631.000000  98767.000000  500006.000000  982716.000000
```

As you can see, it gives us several measures such as `count`, `mean`, `std` (standard deviation), `min`, `max`, and shows how data is distributed in the various quadrants. Thanks to this method, we could already have a rough idea of how our data is structured.

Let's see which are the three campaigns with the highest and lowest budgets:

```
#13

   df.sort_index(by=['cmp_bgt'], ascending=False).head(3)
```

This gives the following output (truncated):

```
      cmp_bgt  cmp_clicks  cmp_impr                  cmp_name
4655   999631       15343    499997  AKX_20160814_20180226_40
3708   999606       45367    499997  KTR_20150523_20150527_35
1995   999445       12580    499998  AKX_20141102_20151009_30
```

And (`#14`) a call to `.tail(3)`, shows us the ones with the lowest budget.

Unpacking the campaign name

Now it's time to increase the complexity up a bit. First of all, we want to get rid of that horrible campaign name (`cmp_name`). We need to explode it into parts and put each part in one dedicated column. In order to do this, we'll use the `apply` method of the **Series** object.

The `pandas.core.series.Series` class is basically a powerful wrapper around an array (think of it as a list with augmented capabilities). We can extrapolate a `Series` object from a `DataFrame` by accessing it in the same way we do with a key in a dict, and we can call `apply` on that `Series` object, which will run a function feeding each item in the `Series` to it. We compose the result into a new `DataFrame`, and then join that `DataFrame` with our beloved `df`.

```
#15

   def unpack_campaign_name(name):
       # very optimistic method, assumes data in campaign name
       # is always in good state
       type_, start, end, age, gender, currency = name.split('_')
       start = parse(start).date
       end = parse(end).date
       return type_, start, end, age, gender, currency

   campaign_data = df['cmp_name'].apply(unpack_campaign_name)
   campaign_cols = [
```

```
        'Type', 'Start', 'End', 'Age', 'Gender', 'Currency']
    campaign_df = DataFrame(
        campaign_data.tolist(), columns=campaign_cols, index=df.index)
    campaign_df.head(3)
```

Within `unpack_campaign_name`, we split the campaign `name` in parts. We use `delorean.parse()` to get a proper date object out of those strings (`delorean` makes it really easy to do it, doesn't it?), and then we return the objects. A quick peek at the last line reveals:

```
  Type       Start         End    Age Gender Currency
0  KTR  2016-06-16  2017-01-24  20-30      M      EUR
1  BYU  2014-10-25  2015-07-31  35-50      B      USD
2  BYU  2015-10-26  2016-03-17  35-50      M      EUR
```

Nice! One important thing: even if the dates appear as strings, they are just the representation of the real `date` objects that are hosted in the `DataFrame`.

Another very important thing: when joining two DataFrame instances, it's imperative that they have the same index, otherwise `pandas` won't be able to know which rows go with which. Therefore, when we create `campaign_df`, we set its index to the one from `df`. This enables us to join them. When creating this DataFrame, we also pass the columns names.

```
#16

    df = df.join(campaign_df)
```

And after the join, we take a peek, hoping to see matching data (output truncated):

```
#17

    df[['cmp_name'] + campaign_cols].head(3)
```

Gives:

```
                           cmp_name Type       Start         End
0  KTR_20160616_20170124_20-30_M_EUR  KTR  2016-06-16  2017-01-24
1  BYU_20141025_20150731_35-50_B_USD  BYU  2014-10-25  2015-07-31
2  BYU_20151026_20160317_35-50_M_EUR  BYU  2015-10-26  2016-03-17
```

As you can see, the join was successful; the campaign name and the separate columns expose the same data. Did you see what we did there? We're accessing the `DataFrame` using the square brackets syntax, and we pass a list of column names. This will produce a brand new `DataFrame`, with those columns (in the same order), on which we then call `head()`.

Unpacking the user data

We now do the exact same thing for each piece of `user` JSON data. We call `apply` on the `user` Series, running the `unpack_user_json` function, which takes a JSON `user` object and transforms it into a list of its fields, which we can then inject into a brand new DataFrame `user_df`. After that, we'll join `user_df` back with `df`, like we did with `campaign_df`.

```
#18

    def unpack_user_json(user):
        # very optimistic as well, expects user objects
        # to have all attributes
        user = json.loads(user.strip())
        return [
            user['username'],
            user['email'],
            user['name'],
            user['gender'],
            user['age'],
            user['address'],
        ]

    user_data = df['user'].apply(unpack_user_json)
    user_cols = [
        'username', 'email', 'name', 'gender', 'age', 'address']
    user_df = DataFrame(
        user_data.tolist(), columns=user_cols, index=df.index)
```

Very similar to the previous operation, isn't it? We should also note here that, when creating `user_df`, we need to instruct `DataFrame` about the column names and, very important, the index. Let's join (`#19`) and take a quick peek (`#20`):

```
    df = df.join(user_df)
    df[['user'] + user_cols].head(2)
```

The output shows us that everything went well. We're good, but we're not done yet.

If you call `df.columns` in a cell, you'll see that we still have ugly names for our columns. Let's change that:

```
#21

    better_columns = [
        'Budget', 'Clicks', 'Impressions',
        'cmp_name', 'Spent', 'user',
```

```
        'Type', 'Start', 'End',
        'Target Age', 'Target Gender', 'Currency',
        'Username', 'Email', 'Name',
        'Gender', 'Age', 'Address',
    ]
    df.columns = better_columns
```

Good! Now, with the exception of `'cmp_name'` and `'user'`, we only have nice names.

Completing the `datasetNext` step will be to add some extra columns. For each campaign, we have the amount of clicks and impressions, and we have the spent. This allows us to introduce three measurement ratios: **CTR**, **CPC**, and **CPI**. They stand for **Click Through Rate**, **Cost Per Click**, and **Cost Per Impression**, respectively.

The last two are easy to understand, but CTR is not. Suffice it to say that it is the ratio between clicks and impressions. It gives you a measure of how many clicks were performed on a campaign advertisement per impression: the higher this number, the more successful the advertisement is in attracting users to click on it.

```
#22

    def calculate_extra_columns(df):
        # Click Through Rate
        df['CTR'] = df['Clicks'] / df['Impressions']
        # Cost Per Click
        df['CPC'] = df['Spent'] / df['Clicks']
        # Cost Per Impression
        df['CPI'] = df['Spent'] / df['Impressions']
    calculate_extra_columns(df)
```

I wrote this as a function, but I could have just written the code in the cell. It's not important. What I want you to notice here is that we're adding those three columns with one line of code each, but the `DataFrame` applies the operation automatically (the division, in this case) to each pair of cells from the appropriate columns. So, even if they are masked as three divisions, these are actually *4974 * 3* divisions, because they are performed for each row. Pandas does a lot of work for us, and also does a very good job in hiding the complexity of it.

The function, `calculate_extra_columns`, takes a `DataFrame`, and works directly on it. This mode of operation is called **in-place**. Do you remember how `list.sort()` was sorting the list? It is the same deal.

We can take a look at the results by filtering on the relevant columns and calling `head`.

```
#23

    df[['Spent', 'Clicks', 'Impressions',
        'CTR', 'CPC', 'CPI']].head(3)
```

This shows us that the calculations were performed correctly on each row:

```
    Spent  Clicks  Impressions       CTR       CPC       CPI
0   57574   25576       500001  0.051152  2.251095  0.115148
1  226319   61247       499999  0.122494  3.695185  0.452639
2    4354   15582       500004  0.031164  0.279425  0.008708
```

Now, I want to verify the accuracy of the results manually for the first row:

```
#24

    clicks = df['Clicks'][0]
    impressions = df['Impressions'][0]
    spent = df['Spent'][0]
    CTR = df['CTR'][0]
    CPC = df['CPC'][0]
    CPI = df['CPI'][0]
    print('CTR:', CTR, clicks / impressions)
    print('CPC:', CPC, spent / clicks)
    print('CPI:', CPI, spent / impressions)
```

It yields the following output:

```
CTR: 0.0511518976962 0.0511518976962
CPC: 2.25109477635 2.25109477635
CPI: 0.115147769704 0.115147769704
```

This is exactly what we saw in the previous output. Of course, I wouldn't normally need to do this, but I wanted to show you how can you perform calculations this way. You can access a Series (a column) by passing its name to the `DataFrame`, in square brackets, and then you access each row by its position, exactly as you would with a regular list or tuple.

We're almost done with our `DataFrame`. All we are missing now is a column that tells us the duration of the campaign and a column that tells us which day of the week corresponds to the start date of each campaign. This allows me to expand on how to play with date objects.

```
#25

    def get_day_of_the_week(day):
        number_to_day = dict(enumerate(calendar.day_name, 1))
        return number_to_day[day.isoweekday()]

    def get_duration(row):
        return (row['End'] - row['Start']).days

    df['Day of Week'] = df['Start'].apply(get_day_of_the_week)
    df['Duration'] = df.apply(get_duration, axis=1)
```

We used two different techniques here but first, the code.

`get_day_of_the_week` takes a date object. If you cannot understand what it does, please take a few moments to try and understand for yourself before reading the explanation. Use the inside-out technique like we've done a few times before.

So, as I'm sure you know by now, if you put `calendar.day_name` in a `list` call, you get `['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']`. This means that, if we enumerate `calendar.day_name` starting from 1, we get pairs such as `(1, 'Monday')`, `(2, 'Tuesday')`, and so on. If we feed these pairs to a dict, we get a mapping between the days of the week as numbers (1, 2, 3, ...) and their names. When the mapping is created, in order to get the name of a day, we just need to know its number. To get it, we call `date.isoweekday()`, which tells us which day of the week that date is (as a number). You feed that into the mapping and, boom! You have the name of the day.

`get_duration` is interesting as well. First, notice it takes an entire row, not just a single value. What happens in its body is that we perform a subtraction between a campaign end and start dates. When you subtract date objects the result is a `timedelta` object, which represents a given amount of time. We take the value of its `.days` property. It is as simple as that.

Now, we can introduce the fun part, the application of those two functions.

The first application is performed on a `Series` object, like we did before for `'user'` and `'cmp_name'`, there is nothing new here.

The second one is applied to the whole DataFrame and, in order to instruct Pandas to perform that operation on the rows, we pass `axis=1`.

We can verify the results very easily, as shown here:

```
#26
    df[['Start', 'End', 'Duration', 'Day of Week']].head(3)
```

Yields:

```
        Start         End  Duration Day of Week
0  2015-08-26  2017-03-05       557   Wednesday
1  2014-10-15  2014-12-19        65   Wednesday
2  2015-02-22  2016-01-14       326      Sunday
```

So, we now know that between the 26th of August 2015 and the 5th of March 2017 there are 557 days, and that the 26th of August 2015 was a Wednesday.

If you're wondering what the purpose of this is, I'll provide an example. Imagine that you have a campaign that is tied to a sports event that usually takes place on a Sunday. You may want to inspect your data according to the days so that you can correlate them to the various measurements you have. We're not going to do it in this project, but it was useful to see, if only for the different way of calling `apply()` on a DataFrame.

Cleaning everything up

Now that we have everything we want, it's time to do the final cleaning: remember we still have the `'cmp_name'` and `'user'` columns. Those are useless now, so they have to go. Also, I want to reorder the columns in the DataFrame so that it is more relevant to the data it now contains. In order to do this, we just need to filter `df` on the column list we want. We'll get back a brand new DataFrame that we can reassign to `df` itself.

```
#27
    final_columns = [
        'Type', 'Start', 'End', 'Duration', 'Day of Week', 'Budget',
        'Currency', 'Clicks', 'Impressions', 'Spent', 'CTR', 'CPC',
        'CPI', 'Target Age', 'Target Gender', 'Username', 'Email',
        'Name', 'Gender', 'Age'
    ]
    df = df[final_columns]
```

I have grouped the campaign information at the beginning, then the measurements, and finally the user data at the end. Now our DataFrame is clean and ready for us to inspect.

Before we start going crazy with graphs, what about taking a snapshot of our DataFrame so that we can easily reconstruct it from a file, rather than having to redo all the steps we did to get here. Some analysts may want to have it in spreadsheet form, to do a different kind of analysis than the one we want to do, so let's see how to save a DataFrame to a file. It's easier done than said.

Saving the DataFrame to a file

We can save a DataFrame in many different ways. You can type `df.to_` and then press *Tab* to make auto-completion pop up, to see all the possible options.

We're going to save our DataFrame in three different formats, just for fun: **comma-separated values** (**CSV**), JSON, and Excel spreadsheet.

```
#28

   df.to_csv('df.csv')

#29

   df.to_json('df.json')

#30

   df.to_excel('df.xls')
```

The CSV file looks like this (output truncated):

```
Type,Start,End,Duration,Day of Week,Budget,Currency,Clicks,Impres
0,GRZ,2015-03-15,2015-11-10,240,Sunday,622551,GBP,35018,500002,787
1,AKX,2016-06-19,2016-09-19,92,Sunday,148219,EUR,45185,499997,6588
2,BYU,2014-09-25,2016-07-03,647,Thursday,537760,GBP,55771,500001,3
```

And the JSON one like this (again, output truncated):

```
{
  "Type": {
    "0": "GRZ",
    "1": "AKX",
    "2": "BYU",
```

So, it's extremely easy to save a DataFrame in many different formats, and the good news is that the opposite is also true: it's very easy to load a spreadsheet into a DataFrame. The programmers behind Pandas went a long way to ease our tasks, something to be grateful for.

Visualizing the results

Finally, the juicy bits. In this section, we're going to visualize some results. From a data science perspective, I'm not very interested in going deep into analysis, especially because the data is completely random, but nonetheless, this code will get you started with graphs and other features.

Something I learned in my life—and this may come as a surprise to you—is that *looks also counts* so it's very important that when you present your results, you do your best to *make them pretty*.

I won't try to prove to you how truthful that last statement is, but I really do believe in it. If you recall the last line of cell `#1`:

```
# make the graphs nicer
pd.set_option('display.mpl_style', 'default')
```

Its purpose is to make the graphs we will look at in this section a little bit prettier.

Okay, so, first of all we have to instruct the notebook that we want to use `matplotlib inline`. This means that when we ask Pandas to plot something, we will have the result rendered in the cell output frame. In order to do this, we just need one simple instruction:

```
#31

%matplotlib inline
```

You can also instruct the notebook to do this when you start it from the console by passing a parameter, but I wanted to show you this way too, since it can be annoying to have to restart the notebook just because you want to plot something. In this way, you can do it on the fly and then keep working.

Next, we're going to set some parameters on `pylab`. This is for plotting purposes and it will remove a warning that a font hasn't been found. I suggest that you do not execute this line and keep going. If you get a warning that a font is missing, come back to this cell and run it.

```
#32

import pylab
pylab.rcParams.update({'font.family' : 'serif'})
```

This basically tells Pylab to use the first available serif font. It is simple but effective, and you can experiment with other fonts too.

Now that the DataFrame is complete, let's run `df.describe()` (`#33`) again. The results should look something like this:

	Duration	Budget	Clicks	Impressions	Spent	CTR	CPC	CPI	Age
count	4969.000000	4969.000000	4969.000000	4969.000000	4969.000000	4969.000000	4969.000000	4969.000000	4969.000000
mean	364.414772	500490.476353	40110.432481	499999.465889	252225.839203	0.080221	9.835294	0.504452	53.854699
std	212.972833	288470.674072	21619.233699	2.047371	218765.741972	0.043239	16.326438	0.437532	21.149709
min	1.000000	1190.000000	535.000000	499992.000000	114.000000	0.001070	0.002838	0.000228	18.000000
25%	176.000000	252672.000000	22579.000000	499998.000000	69408.000000	0.045158	1.855093	0.138817	35.000000
50%	364.000000	502102.000000	37029.000000	499999.000000	192934.000000	0.074058	5.327965	0.385869	53.000000
75%	554.000000	749372.000000	55504.000000	500001.000000	384520.000000	0.111008	11.640189	0.769046	73.000000
max	730.000000	998817.000000	98739.000000	500007.000000	980234.000000	0.197479	322.773361	1.960468	90.000000

This kind of quick result is perfect to satisfy those managers who have 20 seconds to dedicate to you and just want rough numbers.

Once again, please keep in mind that our campaigns have different currencies, so these numbers are actually meaningless. The point here is to demonstrate the DataFrame capabilities, not to get to a correct or detailed analysis of real data.

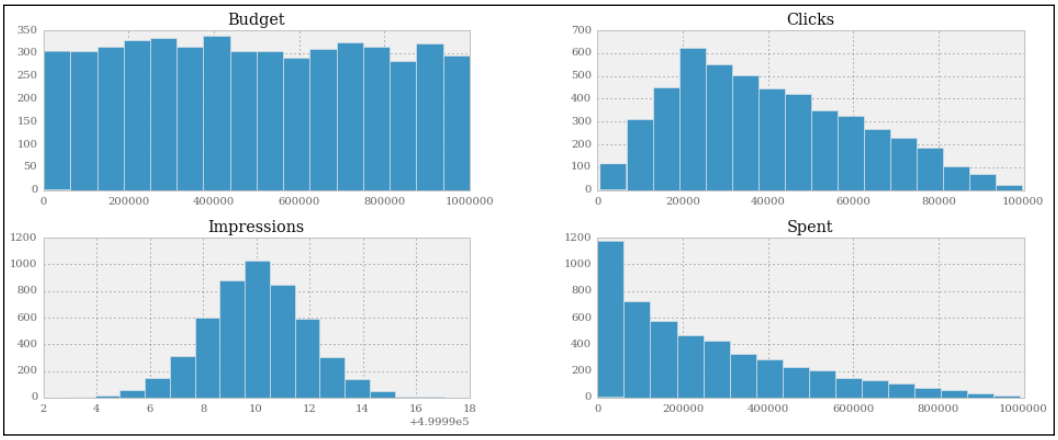
Alternatively, a graph is usually much better than a table with numbers because it's much easier to read it and it gives you immediate feedback. So, let's graph out the four pieces of information we have on each campaign: budget, spent, clicks, and impressions.

`#34`

```
df[['Budget', 'Spent', 'Clicks', 'Impressions']].hist(
    bins=16, figsize=(16, 6));
```

We extrapolate those four columns (this will give us another DataFrame made with only those columns) and call the histogram `hist()` method on it. We give some measurements on the bins and figure sizes, but basically everything is done automatically.

One important thing: since this instruction is the only one in this cell (which also means, it's the last one), the notebook will print its result before drawing the graph. To suppress this behavior and have only the graph drawn with no printing, just add a semicolon at the end (you thought I was reminiscing about Java, didn't you?). Here are the graphs:



They are beautiful, aren't they? Did you notice the serif font? How about the meaning of those figures? If you go back to `#6` and take a look at the way we generate the data, you will see that all these graphs make perfect sense.

Budget is simply a random integer in an interval, therefore we were expecting a *uniform distribution*, and there we have it; it's practically a constant line.

Spent is a *uniform distribution* as well, but the high end of its interval is the budget, which is moving, this means we should expect something like a quadratic hyperbole that decreases to the right. And there it is as well.

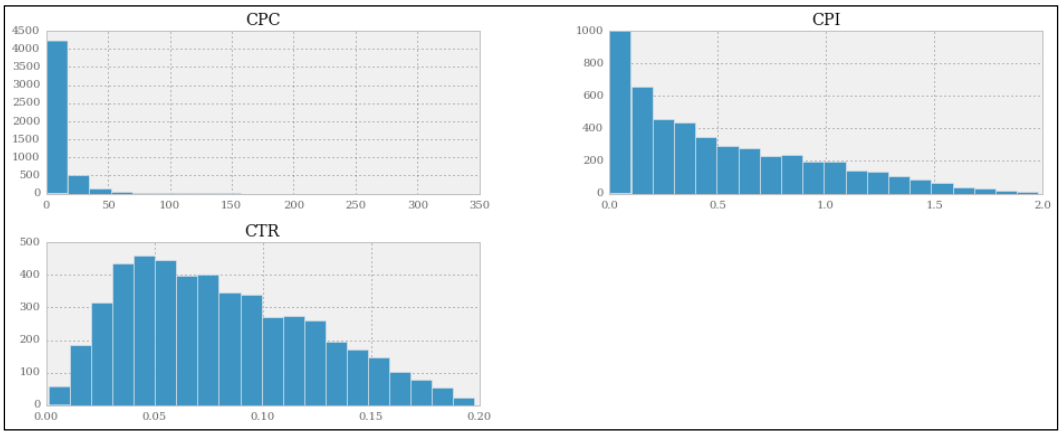
Clicks was generated with a *triangular distribution* with mean roughly 20% of the interval size, and you can see that the peak is right there, at about 20% to the left.

Finally, **Impressions** was a *Gaussian distribution*, which is the one that assumes the famous bell shape. The mean was exactly in the middle and we had standard deviation of 2. You can see that the graph matches those parameters.

Good! Let's plot out the measures we calculated:

```
#35
```

```
df[['CTR', 'CPC', 'CPI']].hist(
    bins=20, figsize=(16, 6));
```



We can see that the cost per click is highly skewed to the left, meaning that most of the CPC values are very low. The cost per impression has a similar shape, but less extreme.

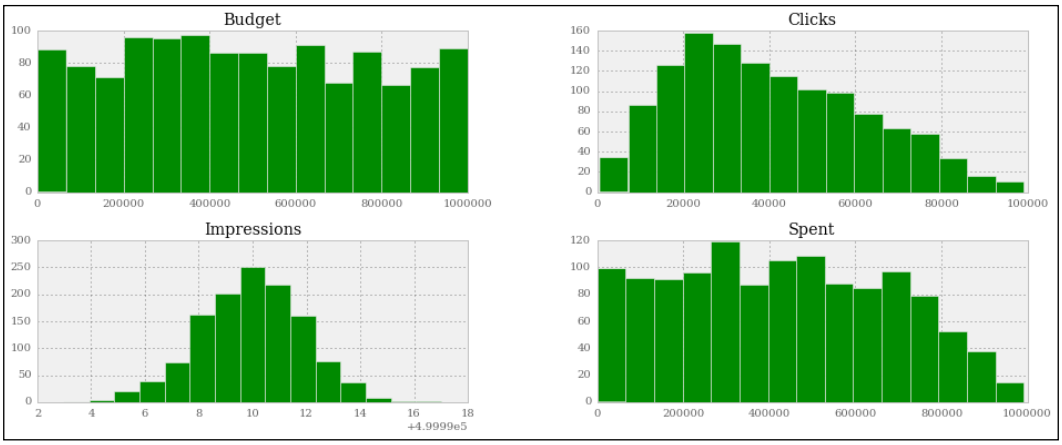
Now, all this is nice, but if you wanted to analyze only a particular segment of the data, how would you do it? We can apply a mask to a DataFrame, so that we get another one with only the rows that satisfy the mask condition. It's like applying a global row-wise `if` clause.

```
#36
```

```
mask = (df.Spent > 0.75 * df.Budget)
df[mask][['Budget', 'Spent', 'Clicks', 'Impressions']].hist(
    bins=15, figsize=(16, 6), color='g');
```

In this case, I prepared a mask to filter out all the rows for which the spent is less than or equal to 75% of the budget. In other words, we'll include only those campaigns for which we have spent at least three quarters of the budget. Notice that in the mask I am showing you an alternative way of asking for a DataFrame column, by using direct property access (`object.property_name`), instead of dict-like access (`object['property_name']`). If `property_name` is a valid Python name, you can use both ways interchangeably (JavaScript works like this as well).

The mask is applied in the same way that we access a dict with a key. When you apply a mask to a DataFrame, you get back another one and we select only the relevant columns on this, and call `hist()` again. This time, just for fun, we want the results to be painted green:



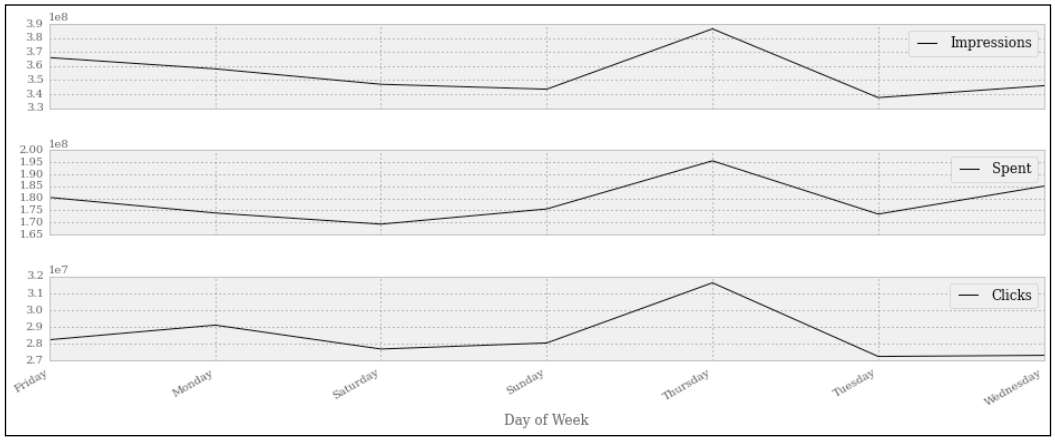
Note that the shapes of the graphs haven't changed much, apart from the spent, which is quite different. The reason for this is that we've asked only for the rows where spent is at least 75% of the budget. This means that we're including only the rows where spent is close to the budget. The budget numbers come from a uniform distribution. Therefore, it is quite obvious that the spent is now assuming that kind of shape. If you make the boundary even tighter, and ask for 85% or more, you'll see spent become more and more like budget.

Let's now ask for something different. How about the measure of spent, click, and impressions grouped by day of the week?

```
#37

    df_weekday = df.groupby(['Day of Week']).sum()
    df_weekday[['Impressions', 'Spent', 'Clicks']].plot(
        figsize=(16, 6), subplots=True);
```

The first line creates a new `DataFrame`, `df_weekday`, by asking for a grouping by `'Day of Week'` on `df`. The function used to aggregate the data is addition.

The second line gets a slice of `df_weekday` using a list of column names, something we're accustomed to by now. On the result we call `plot()`, which is a bit different to `hist()`. The option `subplots=True` makes `plot` draw three independent graphs:



Interestingly enough, we can see that most of the action happens on Thursdays. If this were meaningful data, this would potentially be important information to give to our clients, and this is the reason I'm showing you this example.

Note that the days are sorted alphabetically, which scrambles them up a bit. Can you think of a quick solution that would fix the issue? I'll leave it to you as an exercise to come up with something.

Let's finish this presentation section with a couple more things. First, a simple aggregation. We want to aggregate on `'Target Gender'` and `'Target Age'`, and show `'Impressions'` and `'Spent'`. For both, we want to see the mean and the standard deviation.

```
#38

    agg_config = {
        'Impressions': {
            'Mean Impr': 'mean',
            'Std Impr': 'std',
        },
        'Spent': ['mean', 'std'],
    }

    df.groupby(['Target Gender', 'Target Age']).agg(agg_config)
```

It's very easy to do it. We will prepare a dictionary that we'll use as a configuration. I'm showing you two options to do it. We use a nicer format for `'Impressions'`, where we pass a nested dict with description/function as key/value pairs. On the other hand, for `'Spent'`, we just use a simpler list with just the function names.

Then, we perform a grouping on the `'Target Gender'` and `'Target Age'` columns, and we pass our configuration dict to the `agg()` method. The result is truncated and rearranged a little bit to make it fit, and shown here:

```
                     Impressions              Spent
                   Mean Impr  Std Impr    mean            std
Target Target
Gender Age
B      20-25          500000  2.189102  239882  209442.168488
       20-30          500000  2.245317  271285  236854.155720
       20-35          500000  1.886396  243725  174268.898935
       20-40          499999  2.100786  247740  211540.133771
       20-45          500000  1.772811  148712  118603.932051
...                     ...       ...     ...           ...
M      20-25          500000  2.022023  212520  215857.323228
       20-30          500000  2.111882  292577  231663.713956
       20-35          499999  1.965177  255651  222790.960907
       20-40          499999  1.932473  282515  250023.393334
       20-45          499999  1.905746  271077  219901.462405
```

This is the textual representation, of course, but you can also have the HTML one. You can see that `Spent` has the `mean` and `std` columns whose labels are simply the function names, while `Impressions` features the nice titles we added to the configuration dict.

Let's do one more thing before we wrap this chapter up. I want to show you something called a **pivot table**. It's kind of a buzzword in the data environment, so an example such as this one, albeit very simple, is a must.

```
#39

   pivot = df.pivot_table(
       values=['Impressions', 'Clicks', 'Spent'],
       index=['Target Age'],
       columns=['Target Gender'],
       aggfunc=np.sum
   )
   pivot
```

We create a pivot table that shows us the correlation between the target age and impressions, clicks, and spent. These last three will be subdivided according to the target gender. The aggregation function used to calculate the results is the `numpy.sum` function (`numpy.mean` would be the default, had I not specified anything).

After creating the pivot table, we simply print it with the last line in the cell, and here's a crop of the result:

	Impressions			Clicks			Spent		
Target Gender	**B**	**F**	**M**	**B**	**F**	**M**	**B**	**F**	**M**
Target Age									
20-25	42499954	40499948	43499987	3457493	3108138	3401407	19961037	19327699	23420817
20-30	46999958	38499935	39499970	4026055	3061723	3131742	25118876	21148545	23782193
20-35	40999986	44499986	34499957	3177185	3315678	2784411	21849985	17988043	17228214
20-40	38499926	40499926	43499942	3154637	3208569	3650371	19819058	18913285	23439361
20-45	6999990	11499988	7499985	540434	952853	486264	3948282	5280748	4639397
25-30	34499979	37499967	33999950	2927039	2906621	2787992	19193461	17564978	17489240
25-35	43499973	39499982	31499977	3842891	3124853	2479525	20950036	16196097	17281155
25-40	27499973	36999941	26499972	2155983	3027652	2068474	12666195	17888093	12811784
25-45	33999981	41499988	43499964	2485752	3623777	3629015	14576611	20266282	21521781
25-50	9999985	7499983	10500012	765251	598021	815785	5715627	3564206	5091148
30-35	36499974	40999994	32999963	3030691	3672607	2965210	20269909	20762924	15639066
30-40	39999964	38999955	40499971	3186630	2904775	3518337	23996589	21778382	21769771

It's pretty clear and provides very useful information when the data is meaningful.

That's it! I'll leave you to discover more about the wonderful world of IPython, Jupyter, and data science. I strongly encourage you to get comfortable with the notebook environment. It's much better than a console, it's extremely practical and fun to use, and you can even do slides and documents with it.

Where do we go from here?

Data science is indeed a fascinating subject. As I said in the introduction, those who want to delve into its meanders need to be well trained in mathematics and statistics. Working with data that has been interpolated incorrectly renders any result about it useless. The same goes for data that has been extrapolated incorrectly or sampled with the wrong frequency. To give you an example, imagine a population of individuals that are aligned in a queue. If, for some reason, the gender of that population alternated between male and female, the queue would be something like this: F-M-F-M-F-M-F-M-F...

If you sampled it taking only the even elements, you would draw the conclusion that the population was made up only of males, while sampling the odd ones would tell you exactly the opposite.

Of course, this was just a silly example, I know, but believe me it's very easy to make mistakes in this field, especially when dealing with big data where sampling is mandatory and therefore, the quality of the introspection you make depends, first and foremost, on the quality of the sampling itself.

When it comes to data science and Python, these are the main tools you want to look at:

- **NumPy** (`http://www.numpy.org/`): This is the fundamental package for scientific computing with Python. It contains a powerful N-dimensional array object, sophisticated (broadcasting) functions, tools for integrating C/C++ and Fortran code, useful linear algebra, Fourier transform, random number capabilities, and much more.
- **Scikit-Learn** (`http://scikit-learn.org/stable/`): This is probably the most popular machine learning library in Python. It has simple and efficient tools for data mining and data analysis, accessible to everybody, and reusable in various contexts. It's built on NumPy, SciPy, and Matplotlib.
- **Pandas** (`http://pandas.pydata.org/`): This is an open source, BSD-licensed library providing high-performance, easy-to-use data structures, and data analysis tools. We've used it throughout this whole chapter.
- **IPython** (`http://ipython.org/`) / Jupyter (`http://jupyter.org/`): These provide a rich architecture for interactive computing.
- **Matplotlib** (`http://matplotlib.org/`): This is a Python 2D plotting library that produces publication-quality figures in a variety of hard copy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shell and notebook, web application servers, and six graphical user interface toolkits.

- **Numba** (`http://numba.pydata.org/`): This gives you the power to speed up your applications with high performance functions written directly in Python. With a few annotations, array-oriented and math-heavy Python code can be just-in-time compiled to native machine instructions, similar in performance to C, C++, and Fortran, without having to switch languages or Python interpreters.
- **Bokeh** (`http://bokeh.pydata.org/en/latest/`): It's a Python-interactive visualization library that targets modern web browsers for presentation. Its goal is to provide elegant, concise construction of novel graphics in the style of D3.js, but also deliver this capability with high-performance interactivity over very large or streaming datasets.

Other than these single libraries, you can also find ecosystems such as **SciPy** (`http://scipy.org/`) and **Anaconda** (`https://www.continuum.io/`), which bundle several different packages in order to give you something that just works in an "out-of-the-box" fashion.

Installing all these tools and their several dependencies is hard on some systems, so I suggest that you try out ecosystems as well and see if you are comfortable with them. It may be worth it.

Summary

In this chapter, we talked about data science. Rather than attempting to explain anything about this extremely wide subject, we delved into a project. We familiarized ourselves with the Jupyter notebook, and with different libraries such as Pandas, Matplotlib, NumPy.

Of course, having to compress all this information into one single chapter means I could only touch briefly on the subjects I presented. I hope the project we've gone through together has been comprehensive enough to give you a good idea about what could potentially be the workflow you might follow when working in this field.

The next chapter is dedicated to web development. So, make sure you have a browser ready and let's go!

10
Web Development Done Right

"Don't believe everything you read on the Web."

- Confucius

In this chapter, we're going to work on a website together. By working on a small project, my aim is to open a window for you to take a peek on what web development is, along with the main concepts and tools you should know if you want to be successful with it.

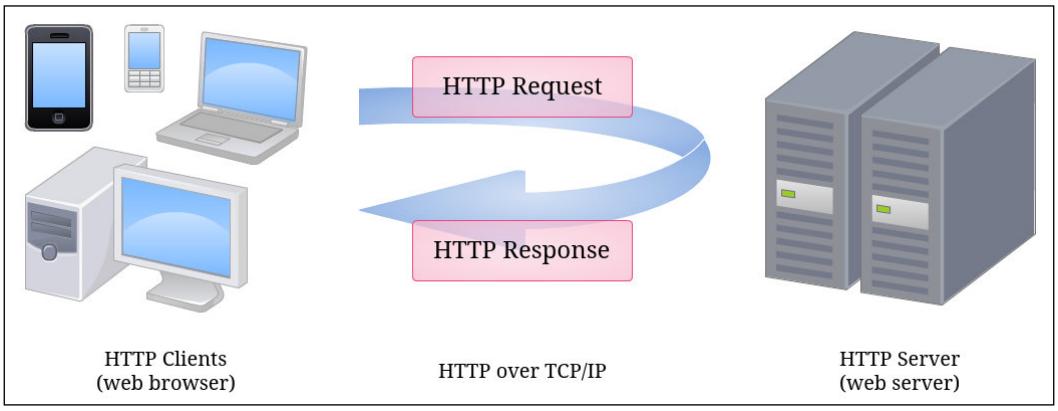
What is the Web?

The **World Wide Web**, or simply **Web**, is a way of accessing information through the use of a medium called the **Internet**. The Internet is a huge network of networks, a networking infrastructure. Its purpose is to connect billions of devices together, all around the globe, so that they can communicate with one another. Information travels through the Internet in a rich variety of languages called **protocols**, which allow different devices to speak the same tongue in order to share content.

The Web is an information-sharing model, built on top of the Internet, which employs the **Hypertext Transfer Protocol** (**HTTP**) as a basis for data communication. The Web, therefore, is just one of the several different ways information can be exchanged over the Internet: e-mail, instant messaging, news groups, and so on, they all rely on different protocols.

How does the Web work?

In a nutshell, HTTP is an asymmetric **request-response client-server** protocol. An HTTP client sends a request message to an HTTP server. The server, in turn, returns a response message. In other words, HTTP is a **pull protocol** in which the client pulls information from the server (as opposed to a **push protocol** in which the server pushes information down to the client). Take a look at the following image:



HTTP is based on **TCP/IP** (**Transmission Control Protocol/Internet Protocol**), which provides the tools for a reliable communication exchange.

An important feature of the HTTP protocol is that it's *stateless*. This means that the current request has no knowledge about what happened in previous requests. This is a limitation, but you can browse a website with the illusion of being logged in. Under the covers though, what happens is that, on login, a token of user information is saved (most often on the client side, in special files called **cookies**) so that each request the user makes carries the means for the server to recognize the user and provide a custom interface by showing their name, keeping their basket populated, and so on.

Even though it's very interesting, we're not going to delve into the rich details of HTTP and how it works. However, we're going to write a small website, which means we'll have to write the code to handle HTTP requests and return HTTP responses. I won't keep prepending HTTP to the terms *request* and *response* from now on, as I trust there won't be any confusion.

The Django web framework

For our project, we're going to use one of the most popular web frameworks you can find in the Python ecosystem: Django.

A **web framework** is a set of tools (libraries, functions, classes, and so on) that we can use to code a website. We need to decide what kind of requests we want to allow to be issued against our web server and how we respond to them. A web framework is the perfect tool to do that because it takes care of many things for us so that we can concentrate only on the important bits without having to reinvent the wheel.

There are different types of frameworks. Not all of them are designed for writing code for the web. In general, a **framework** is a tool that provides functionalities to facilitate the development of software applications, products and solutions.

Django design philosophy

Django is designed according to the following principles:

- **DRY**: As in, **Don't Repeat Yourself**. Don't repeat code, and code in a way that makes the framework deduce as much as possible from as little as possible.
- **Loose coupling**: The various layers of the framework shouldn't know about each other (unless absolutely necessary for whatever reason). Loose coupling works best when paralleled with high cohesion. To quote Robert Martin: putting together things which change for the same reason, and spreading apart those which change for different reasons.
- **Less code**: Applications should use the least possible amount of code, and be written in a way that favors reuse as much as possible.
- **Consistency**: When using the Django framework, regardless of which layer you're coding against, your experience will be very consistent with the design patterns and paradigms that were chosen to lay out the project.

The framework itself is designed around the **model-template-view** (**MTV**) pattern, which is a variant of **model-view-controller** (**MVC**), which is widely employed by other frameworks. The purpose of such patterns is to separate concerns and promote code reuse and quality.

The model layer

Of the three layers, this is the one that defines the structure of the data that is handled by the application, and deals with data sources. A **model** is a class that represents a data structure. Through some Django magic, models are mapped to database tables so that you can store your data in a relational database.

A **relational database** stores data in tables in which each column is a property of the data and each row represents a single item or entry in the collection represented by that table. Through the **primary key** of each table, which is that part of the data that allows to uniquely identify each item, it is possible to establish relationships between items belonging to different tables, that is, to put them into *relation*.

The beauty of this system is that you don't have to write database-specific code in order to handle your data. You will just have to configure your models correctly and simply use them. The work on the database is done for you by the Django **object-relational mapping** (**ORM**), which takes care of translating operations done on Python objects into a language that a relational database can understand: **SQL** (**Structured Query Language**).

One benefit of this approach is that you will be able to change databases without rewriting your code since all the database specific code is produced by Django on the fly, according to which database it's connected to. Relational databases speak SQL, but each of them has its own unique flavor of it; therefore, not having to hardcode any SQL in our application is a tremendous advantage.

Django allows you to modify your models at any time. When you do, you can run a command that creates a migration, which is the set of instructions needed to port the database in a state that represents the current definition of your models.

To summarize, this layer deals with defining the data structures you need to handle in your website and gives you the means to save and load them from and to the database by simply accessing the models, which are Python objects.

The view layer

The function of a view is handling a request, performing whatever action needs to be carried out, and eventually returning a response. For example, if you open your browser and request a page corresponding to a category of products in an e-commerce shop, the view will likely talk to the database, asking for all the categories that are children of the selected category (for example, to display them in a navigation sidebar) and for all the products that belong to the selected category, in order to display them on the page.

Therefore, the view is the mechanism through which we can fulfill a request. Its result, the response object, can assume several different forms: a JSON payload, text, an HTML page, and so on. When you code a website, your responses usually consist of HTML or JSON.

The **Hypertext Markup Language**, or **HTML**, is the standard markup language used to create web pages. Web browsers run engines that are capable of interpreting HTML code and render it into what we see when we open a page of a website.

The template layer

This is the layer that provides the bridge between backend and frontend development. When a view has to return HTML, it usually does it by preparing a **context object** (a dict) with some data, and then it feeds this context to a template, which is rendered (that is to say, transformed into HTML) and returned to the caller in the form of a response (more precisely, the body of the response). This mechanism allows for maximum code reuse. If you go back to the category example, it's easy to see that, if you browse a website that sells products, it doesn't really matter which category you click on or what type of search you perform, the layout of the products page doesn't change. What does change is the data with which that page is populated.

Therefore, the layout of the page is defined by a template, which is written in a mixture of HTML and Django template language. The view that serves that page collects all the products to be displayed in the context dict, and feeds it to the template which will be rendered into an HTML page by the Django template engine.

The Django URL dispatcher

The way Django associates a **Uniform Resource Locator** (**URL**) with a view is through matching the requested URL with the patterns that are registered in a special file. A URL represents a page in a website so, for example, `http://mysite.com/categories?id=123` would probably point to the page for the category with ID `123` on my website, while `https://mysite.com/login` would probably be the user login page.

The difference between HTTP and HTTPS is that the latter adds encryption to the protocol so that the data that you exchange with the website is secured. When you put your credit card details on a website, or log in anywhere, or do anything around sensitive data, you want to make sure that you're using HTTPS.

Regular expressions

The way Django matches URLs to patterns is through a regular expression. A **regular expression** is a sequence of characters that defines a search pattern with which we can carry out operations such as pattern and string matching, find/replace, and so on.

Regular expressions have a special syntax to indicate things like digits, letters, spaces, and so on, as well as how many times we expect a character to appear, and much more. A complete explanation of this topic is beyond of the scope of the book. However, it is a very important topic, so the project we're going to work on together will evolve around it, in the hope that you will be stimulated to find the time to explore it a bit more on your own.

To give you a quick example, imagine that you wanted to specify a pattern to match a date such as `"26-12-1947"`. This string consists of two digits, one dash, two digits, one dash, and finally four digits. Therefore, we could write it like this: `r'[0-9]{2}-[0-9]{2}-[0-9]{4}'`. We created a class by using square brackets, and we defined a range of digits inside, from 0 to 9, hence all the possible digits. Then, between curly braces, we say that we expect two of them. Then a dash, then we repeat this pattern once as it is, and once more, by changing how many digits we expect, and without the final dash. Having a class like `[0-9]` is such a common pattern that a special notation has been created as a shortcut: `'\d'`. Therefore, we can rewrite the pattern like this: `r'\d{2}-\d{2}-\d{4}'` and it will work exactly the same. That `r` in front of the string stands for *raw*, and its purpose is to alter the way every backslash `'\'` is interpreted by the regular expression engine.

A regex website

So, here we are. We'll code a website that stores regular expressions so that we'll be able to play with them a little bit.

Before we proceed creating the project, I'd like to spend a word about CSS. **CSS** (**Cascading Style Sheets**) are files in which we specify how the various elements on an HTML page look. You can set all sorts of properties such as shape, size, color, margins, borders, fonts, and so on. In this project, I have tried my best to achieve a decent result on the pages, but I'm neither a frontend developer nor a designer, so please don't pay too much attention to how things look. Try and focus on how they work.

Setting up Django

On the Django website (`https://www.djangoproject.com/`), you can follow the tutorial, which gives you a pretty good idea of Django's capabilities. If you want, you can follow that tutorial first and then come back to this example. So, first things first; let's install Django in your virtual environment:

```
$ pip install django
```

When this command is done, you can test it within a console (try doing it with bpython, it gives you a shell similar to IPython but with nice introspection capabilities):

```
>>> import django
>>> django.VERSION
(1, 8, 4, 'final', 0)
```

Now that Django is installed, we're good to go. We'll have to do some scaffolding, so I'll quickly guide you through that.

Starting the project

Choose a folder in the book's environment and change into that. I'll use `ch10`. From there, we start a Django project with the following command:

```
$ django-admin startproject regex
```

This will prepare the skeleton for a Django project called `regex`. Change into the `regex` folder and run the following:

```
$ python manage.py runserver
```

You should be able to go to `http://127.0.0.1:8000/` with your browser and see the *It worked!* default Django page. This means that the project is correctly set up. When you've seen the page, kill the server with *Ctrl* + *C* (or whatever it says in the console). I'll paste the final structure for the project now so that you can use it as a reference:

```
$ tree -A regex  # from the ch10 folder
regex
├── db.sqlite3
├── entries
│   ├── admin.py
│   ├── forms.py
│   ├── __init__.py
│   ├── migrations
│   │   ├── 0001_initial.py
│   │   └── __init__.py
│   ├── models.py
```

```
│   ├── static
│   │   └── entries
│   │       └── css
│   │           └── main.css
│   ├── templates
│   │   └── entries
│   │       ├── base.html
│   │       ├── footer.html
│   │       ├── home.html
│   │       ├── insert.html
│   │       └── list.html
│   └── views.py
├── manage.py
└── regex
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

Don't worry if you're missing files, we'll get there. A Django project is typically a collection of several different applications. Each application is meant to provide a functionality in a self-contained, reusable fashion. We'll create just one, called **entries**:

```
$ python manage.py startapp entries
```

Within the `entries` folder that has been created, you can get rid of the `tests.py` module.

Now, let's fix the `regex/settings.py` file in the `regex` folder. We need to add our application to the `INSTALLED_APPS` tuple so that we can use it (add it at the bottom of the tuple):

```
INSTALLED_APPS = (
    ... django apps ...
    'entries',
)
```

Then, you may want to fix the language and time zone according to your personal preference. I live in London, so I set them like this:

```
LANGUAGE_CODE = 'en-gb'
TIME_ZONE = 'Europe/London'
```

There is nothing else to do in this file, so you can save and close it.

Now it's time to apply the **migrations** to the database. Django needs database support to handle users, sessions, and things like that, so we need to create a database and populate it with the necessary data. Luckily, this is very easily done with the following command:

```
$ python manage.py migrate
```

For this project, we use a SQLite database, which is basically just a file. On a real project, you would probably use a different database engine like MySQL or PostgreSQL.

Creating users

Now that we have a database, we can create a superuser using the console.

```
$ python manage.py createsuperuser
```

After entering username and other details, we have a user with admin privileges. This is enough to access the Django admin section, so try and start the server:

```
$ python manage.py runserver
```

This will start the Django development server, which is a very useful built-in web server that you can use while working with Django. Now that the server is running, we can access the admin page at `http://localhost:8000/admin/`. I will show you a screenshot of this section later. If you log in with the credentials of the user you just created and head to the **Authentication and Authorization** section, you'll find **Users**. Open that and you will be able to see the list of users. You can edit the details of any user you want as an admin. In our case, make sure you create a different one so that there are at least two users in the system (we'll need them later). I'll call the first user *Fabrizio* (username: `fab`) and the second one *Adriano* (username: `adri`) in honor of my father.

By the way, you should see that the Django admin panel comes for free automatically. You define your models, hook them up, and that's it. This is an incredible tool that shows how advanced Django's introspection capabilities are. Moreover, it is completely customizable and extendable. It's truly an excellent piece of work.

Adding the Entry model

Now that the boilerplate is out of the way, and we have a couple of users, we're ready to code. We start by adding the `Entry` model to our application so that we can store objects in the database. Here's the code you'll need to add (remember to use the project tree for reference):

`entries/models.py`

```
from django.db import models
from django.contrib.auth.models import User
from django.utils import timezone

class Entry(models.Model):
    user = models.ForeignKey(User)
    pattern = models.CharField(max_length=255)
    test_string = models.CharField(max_length=255)
    date_added = models.DateTimeField(default=timezone.now)

    class Meta:
        verbose_name_plural = 'entries'
```

This is the model we'll use to store regular expressions in our system. We'll store a pattern, a test string, a reference to the user who created the entry, and the moment of creation. You can see that creating a model is actually quite easy, but nonetheless, let's go through it line by line.

First we need to import the models module from `django.db`. This will give us the base class for our `Entry` model. Django models are special classes and much is done for us behind the scenes when we inherit from `models.Model`.

We want a reference to the user who created the entry, so we need to import the `User` model from Django's authorization application and we also need to import the timezone model to get access to the `timezone.now()` function, which provides us with a timezone-aware version of `datetime.now()`. The beauty of this is that it's hooked up with the `TIME_ZONE` settings I showed you before.

As for the primary key for this class, if we don't set one explicitly, Django will add one for us. A **primary key** is a key that allows us to uniquely identify an `Entry` object in the database (in this case, Django will add an auto-incrementing integer ID).

So, we define our class, and we set up four class attributes. We have a `ForeignKey` attribute that is our reference to the *User* model. We also have two `CharField` attributes that hold the pattern and test strings for our regular expressions. We also have a `DateTimeField`, whose default value is set to `timezone.now`. Note that we don't call `timezone.now` right there, it's `now`, not `now()`. So, we're not passing a `DateTime` instance (set at the moment in time when that line is parsed) rather, we're passing a *callable*, a function that is called when we save an entry in the database. This is similar to the callback mechanism we used in *Chapter 8*, *The Edges – GUIs and Scripts*, when we were assigning commands to button clicks.

The last two lines are very interesting. We define a class `Meta` within the `Entry` class itself. The `Meta` class is used by Django to provide all sorts of extra information for a model. Django has a great deal of logic under the hood to adapt its behavior according to the information we put in the `Meta` class. In this case, in the admin panel, the pluralized version of `Entry` would be *Entrys*, which is wrong, therefore we need to manually set it. We specify the plural all lowercase, as Django takes care of capitalizing it for us when needed.

Now that we have a new model, we need to update the database to reflect the new state of the code. In order to do this, we need to instruct Django that it needs to create the code to update the database. This code is called **migration**. Let's create it and execute it:

```
$ python manage.py makemigrations entries
$ python manage.py migrate
```

After these two instructions, the database will be ready to store `Entry` objects.

There are two different kinds of migrations: data and schema migration. **Data migrations** port data from one state to another without altering its structure. For example, a data migration could set all products for a category as out of stock by switching a flag to `False` or `0`. A **schema migration** is a set of instructions that alter the structure of the database schema. For example, that could be adding an `age` column to a `Person` table, or increasing the maximum length of a field to account for very long addresses. When developing with Django, it's quite common to have to perform both kinds of migrations over the course of development. Data evolves continuously, especially if you code in an agile environment.

Customizing the admin panel

The next step is to hook the `Entry` model up with the admin panel. You can do it with one line of code, but in this case, I want to add some options to customize a bit the way the admin panel shows the entries, both in the list view of all entry items in the database and in the form view that allows us to create and modify them.

All we need to do is to add the following code:

`entries/admin.py`

```
from django.contrib import admin
from .models import Entry

@admin.register(Entry)
class EntryAdmin(admin.ModelAdmin):
    fieldsets = [
        ('Regular Expression',
         {'fields': ['pattern', 'test_string']}),
        ('Other Information',
         {'fields': ['user', 'date_added']}),
    ]
    list_display = ('pattern', 'test_string', 'user')
    list_filter = ['user']
    search_fields = ['test_string']
```

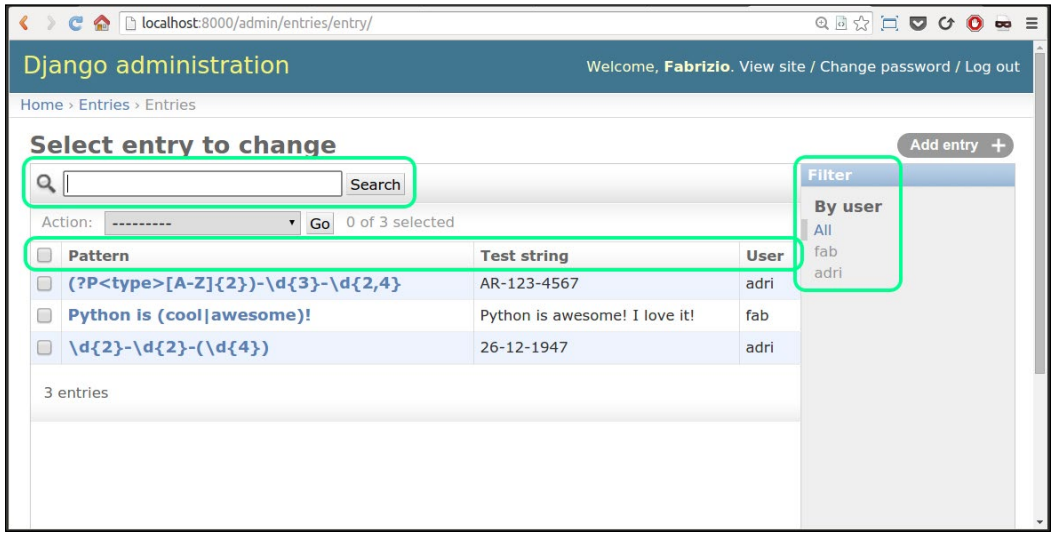
This is simply beautiful. My guess is that you probably already understand most of it, even if you're new to Django.

So, we start by importing the admin module and the `Entry` model. Because we want to foster code reuse, we import the `Entry` model using a relative import (there's a dot before `models`). This will allow us to move or rename the app without too much trouble. Then, we define the `EntryAdmin` class, which inherits from `admin.ModelAdmin`. The decoration on the class tells Django to display the `Entry` model in the admin panel, and what we put in the `EntryAdmin` class tells Django how to customize the way it handles this model.

Firstly, we specify the `fieldsets` for the create/edit page. This will divide the page into two sections so that we get a better visualization of the content (*pattern* and *test string*) and the other details (*user* and *timestamp*) separately.

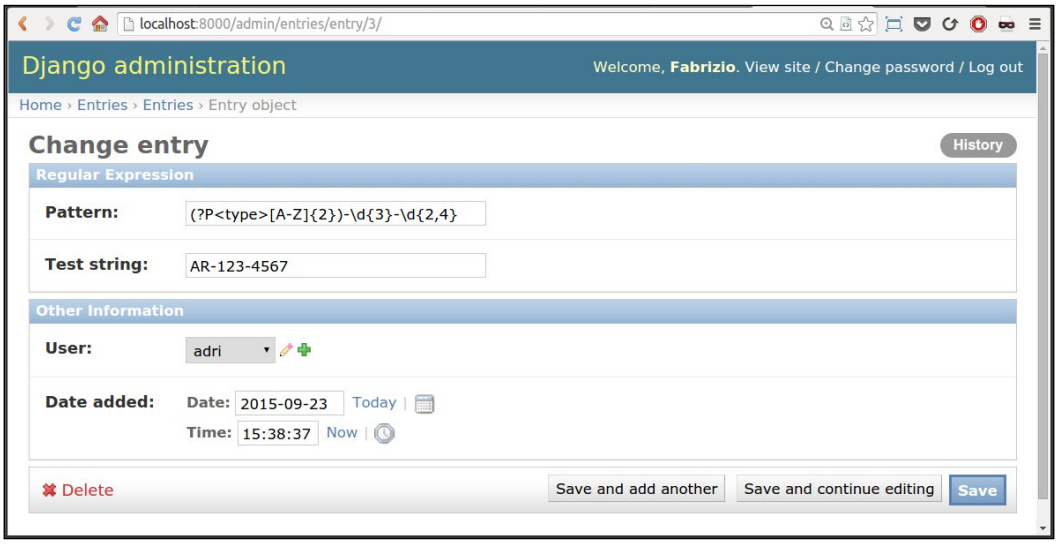
Then, we customize the way the list page displays the results. We want to see all the fields, but not the date. We also want to be able to filter on the user so that we can have a list of all the entries by just one user, and we want to be able to search on `test_string`.

I will go ahead and add three entries, one for myself and two on behalf of my father. The result is shown in the next two images. After inserting them, the list page looks like this:



I have highlighted the three parts of this view that we customized in the `EntryAdmin` class. We can filter by user, we can search and we have all the fields displayed. If you click on a pattern, the edit view opens up.

After our customization, it looks like this:



Notice how we have two sections: **Regular Expression** and **Other Information**, thanks to our custom `EntryAdmin` class. Have a go with it, add some entries to a couple of different users, get familiar with the interface. Isn't it nice to have all this for free?

Creating the form

Every time you fill in your details on a web page, you're inserting data in form fields. A **form** is a part of the **HTML Document Object Model** (**DOM**) **tree**. In HTML, you create a form by using the `form` tag. When you click on the submit button, your browser normally packs the form data together and puts it in the body of a `POST` request. As opposed to `GET` requests, which are used to ask the web server for a resource, a `POST` request normally sends data to the web server with the aim of creating or updating a resource. For this reason, handling `POST` requests usually requires more care than `GET` requests.

When the server receives data from a `POST` request, that data needs to be validated. Moreover, the server needs to employ security mechanisms to protect against various types of attacks. One attack that is very dangerous is the **cross-site request forgery** (**CSRF**) attack, which happens when data is sent from a domain that is not the one the user is authenticated on. Django allows you to handle this issue in a very elegant way.

So, instead of being lazy and using the Django admin to create the entries, I'm going to show you how to do it using a Django form. By using the tools the framework gives you, you get a very good degree of validation work already done (in fact, we won't need to add any custom validation ourselves).

There are two kinds of form classes in Django: `Form` and `ModelForm`. You use the former to create a form whose shape and behavior depends on how you code the class, what fields you add, and so on. On the other hand, the latter is a type of form that, albeit still customizable, infers fields and behavior from a model. Since we need a form for the `Entry` model, we'll use that one.

`entries/forms.py`

```
from django.forms import ModelForm
from .models import Entry

class EntryForm(ModelForm):
    class Meta:
        model = Entry
        fields = ['pattern', 'test_string']
```

Amazingly enough, this is all we have to do to have a form that we can put on a page. The only notable thing here is that we restrict the fields to only `pattern` and `test_string`. Only logged-in users will be allowed access to the insert page, and therefore we don't need to ask who the user is: we know that. As for the date, when we save an `Entry`, the `date_added` field will be set according to its default, therefore we don't need to specify that as well. We'll see in the view how to feed the user information to the form before saving. So, all the background work is done, all we need is the views and the templates. Let's start with the views.

Writing the views

We need to write three views. We need one for the home page, one to display the list of all entries for a user, and one to create a new entry. We also need views to log in and log out. But thanks to Django, we don't need to write them. I'll paste all the code, and then we'll go through it together, step by step.

`entries/views.py`

```
import re
from django.contrib.auth.decorators import login_required
from django.contrib.messages.views import SuccessMessageMixin
from django.core.urlresolvers import reverse_lazy
from django.utils.decorators import method_decorator
from django.views.generic import FormView, TemplateView
from .forms import EntryForm
from .models import Entry

class HomeView(TemplateView):
    template_name = 'entries/home.html'

    @method_decorator(
        login_required(login_url=reverse_lazy('login')))
    def get(self, request, *args, **kwargs):
        context = self.get_context_data(**kwargs)
        return self.render_to_response(context)

class EntryListView(TemplateView):
    template_name = 'entries/list.html'

    @method_decorator(
        login_required(login_url=reverse_lazy('login')))
    def get(self, request, *args, **kwargs):
        context = self.get_context_data(**kwargs)
        entries = Entry.objects.filter(
            user=request.user).order_by('-date_added')
```

```
            matches = (self._parse_entry(entry) for entry in entries)
            context['entries'] = list(zip(entries, matches))
            return self.render_to_response(context)

        def _parse_entry(self, entry):
            match = re.search(entry.pattern, entry.test_string)
            if match is not None:
                return (
                    match.group(),
                    match.groups() or None,
                    match.groupdict() or None
                )
            return None

    class EntryFormView(SuccessMessageMixin, FormView):
        template_name = 'entries/insert.html'
        form_class = EntryForm
        success_url = reverse_lazy('insert')
        success_message = "Entry was created successfully"

        @method_decorator(
            login_required(login_url=reverse_lazy('login')))
        def get(self, request, *args, **kwargs):
            return super(EntryFormView, self).get(
                request, *args, **kwargs)

        @method_decorator(
            login_required(login_url=reverse_lazy('login')))
        def post(self, request, *args, **kwargs):
            return super(EntryFormView, self).post(
                request, *args, **kwargs)

        def form_valid(self, form):
            self._save_with_user(form)
            return super(EntryFormView, self).form_valid(form)

        def _save_with_user(self, form):
            self.object = form.save(commit=False)
            self.object.user = self.request.user
            self.object.save()
```

Let's start with the imports. We need the `re` module to handle regular expressions, then we need a few classes and functions from Django, and finally, we need the `Entry` model and the `EntryForm` form.

The home view

The first view is `HomeView`. It inherits from `TemplateView`, which means that the response will be created by rendering a template with the context we'll create in the view. All we have to do is specify the `template_name` class attribute to point to the correct template. Django promotes code reuse to a point that if we didn't need to make this view accessible only to logged-in users, the first two lines would have been all we needed.

However, we want this view to be accessible only to logged-in users; therefore, we need to decorate it with `login_required`. Now, historically views in Django used to be functions; therefore, this decorator was designed to accept a *function* not a *method* like we have in this class. We're using Django class-based views in this project so, in order to make things work, we need to transform `login_required` so that it accepts a method (the difference being in the first argument: `self`). We do this by passing `login_required` to `method_decorator`.

We also need to feed the `login_required` decorator with `login_url` information, and here comes another wonderful feature of Django. As you'll see after we're done with the views, in Django, you tie a view to a URL through a pattern, consisting of a regular expression and other information. You can give a name to each entry in the `urls.py` file so that when you want to refer to a URL, you don't have to hardcode its value into your code. All you have to do is get Django to reverse-engineer that URL from the name we gave to the entry in `urls.py` defining the URL and the view that is tied to it. This mechanism will become clearer later. For now, just think of `reverse('...')` as a way of getting a URL from an identifier. In this way, you only write the actual URL once, in the `urls.py` file, which is brilliant. In the `views.py` code, we need to use `reverse_lazy`, which works exactly like `reverse` with one major difference: it only finds the URL when we actually need it (in a lazy fashion). This is needed when the `urls.py` file hasn't been loaded yet when the `reverse` function is used.

The `get` method, which we just decorated, simply calls the `get` method of the parent class. Of course, the `get` method is the method that Django calls when a `GET` request is performed against the URL tied to this view.

The entry list view

This view is much more interesting than the previous one. First of all, we decorate the `get` method as we did before. Inside of it, we need to prepare a list of `Entry` objects and feed it to the template, which shows it to the user. In order to do so, we start by getting the `context` dict like we're supposed to do, by calling the `get_context_data` method of the `TemplateView` class. Then, we use the ORM to get a list of the entries. We do this by accessing the objects manager, and calling a filter on it. We filter the entries according to which user is logged in, and we ask for them to be sorted in a descending order (that `'-'` in front of the name specifies the descending order). The `objects` manager is the default **manager** every Django model is augmented with on creation, it allows us to interact with the database through its methods.

We parse each entry to get a list of matches (actually, I coded it so that `matches` is a generator expression). Finally, we add to the context an `'entries'` key whose value is the coupling of `entries` and `matches`, so that each `Entry` instance is paired with the resulting match of its pattern and test string.

On the last line, we simply ask Django to render the template using the context we created.

Take a look at the `_parse_entry` method. All it does is perform a search on the `entry.test_string` with the `entry.pattern`. If the resulting `match` object is not `None`, it means that we found something. If so, we return a tuple with three elements: the overall group, the subgroups, and the group dictionary. If you're not familiar with these terms, don't worry, you'll see a screenshot soon with an example. We return `None` if there is no match.

The form view

Finally, let's examine `EntryFormView`. This is particularly interesting for a few reasons. Firstly, it shows us a nice example of Python's multiple inheritance. We want to display a message on the page, after having inserted an `Entry`, so we inherit from `SuccessMessageMixin`. But we want to handle a form as well, so we also inherit from `FormView`.

Note that, when you deal with mixins and inheritance, you may have to consider the order in which you specify the base classes in the class declaration.

In order to set up this view correctly, we need to specify a few attributes at the beginning: the template to be rendered, the form class to be used to handle the data from the `POST` request, the URL we need to redirect the user to in the case of success, and the success message.

Another interesting feature is that this view needs to handle both `GET` and `POST` requests. When we land on the form page for the first time, the form is empty, and that is the `GET` request. On the other hand, when we fill in the form and want to submit the `Entry`, we make a `POST` request. You can see that the body of `get` is conceptually identical to `HomeView`. Django does everything for us.

The `post` method is just like `get`. The only reason we need to code these two methods is so that we can decorate them to require login.

Within the Django form handling process (in the `FormView` class), there are a few methods that we can override in order to customize the overall behavior. We need to do it with the `form_valid` method. This method will be called when the form validation is successful. Its purpose is to save the form so that an `Entry` object is created out of it, and then stored in the database.

The only problem is that our form is missing the user. We need to intercept that moment in the chain of calls and put the user information in ourselves. This is done by calling the `_save_with_user` method, which is very simple.

Firstly, we ask Django to save the form with the `commit` argument set to `False`. This creates an `Entry` instance without attempting to save it to the database. Saving it immediately would fail because the `user` information is not there.

The next line updates the `Entry` instance (`self.object`), adding the `user` information and, on the last line, we can safely save it. The reason I called it `object` and set it on the instance like that was to follow what the original `FormView` class does.

We're fiddling with the Django mechanism here, so if we want the whole thing to work, we need to pay attention to when and how we modify its behavior, and make sure we don't alter it incorrectly. For this reason, it's very important to remember to call the `form_valid` method of the base class (we use `super` for that) at the end of our own customized version, to make sure that every other action that method usually performs is carried out correctly.

Note how the request is tied to each view instance (`self.request`) so that we don't need to pass it through when we refactor our logic into methods. Note also that the user information has been added to the request automatically by Django. Finally, note that the reason why all the process is split into very small methods like these is so that we can only override those that we need to customize. All this removes the need to write a lot of code.

Now that we have the views covered, let's see how we couple them to the URLs.

Tying up URLs and views

In the `urls.py` module, we tie each view to a URL. There are many ways of doing this. I chose the simplest one, which works perfectly for the extent of this exercise, but you may want to explore this argument more deeply if you intend to work with Django. This is the core around which the whole website logic will revolve; therefore, you should try to get it down correctly. Note that the `urls.py` module belongs to the project folder.

`regex/urls.py`

```
from django.conf.urls import include, url
from django.contrib import admin
from django.contrib.auth import views as auth_views
from django.core.urlresolvers import reverse_lazy
from entries.views import HomeView, EntryListView, EntryFormView

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^entries/$', EntryListView.as_view(), name='entries'),
    url(r'^entries/insert$',
        EntryFormView.as_view(),
        name='insert'),

    url(r'^login/$',
        auth_views.login,
        kwargs={'template_name': 'admin/login.html'},
        name='login'),
    url(r'^logout/$',
        auth_views.logout,
        kwargs={'next_page': reverse_lazy('home')},
        name='logout'),

    url(r'^$', HomeView.as_view(), name='home'),
]
```

As you can see, the magic comes from the `url` function. Firstly, we pass it a regular expression; then the view; and finally, a name, which is what we will use in the `reverse` and `reverse_lazy` functions to recover the URL.

Note that, when using class-based views, we have to transform them into functions, which is what `url` is expecting. To do that, we call the `as_view()` method on them.

Note also that the first `url` entry, for the admin, is special. Instead of specifying a URL and a view, it specifies a URL prefix and another `urls.py` module (from the `admin.site` package). In this way, Django will complete all the URLs for the admin section by prepending `'admin/'` to all the URLs specified in `admin.site.urls`. We could have done the same for our entries app (and we should have), but I feel it would have been a bit too much for this simple project.

In the regular expression language, the `'^'` and `'$'` symbols represent the *start* and *end* of a string. Note that if you use the inclusion technique, as for the admin, the `'$'` is missing. Of course, this is because `'admin/'` is just a prefix, which needs to be completed by all the definitions in the included `urls` module.

Something else worth noticing is that we can also include the stringified version of a path to a view, which we do for the `login` and `logout` views. We also add information about which templates to use with the `kwargs` argument. These views come straight from the `django.contrib.auth` package, by the way, so that we don't need to write a single line of code to handle authentication. This is brilliant and saves us a lot of time.

Each `url` declaration must be done within the `urlpatterns` list and on this matter, it's important to consider that, when Django is trying to find a view for a URL that has been requested, the patterns are exercised in order, from top to bottom. The first one that matches is the one that will provide the view for it so, in general, you have to put specific patterns before generic ones, otherwise they will never get a chance to be caught. For example, `'^shop/categories/$'` needs to come before `'^shop'` (note the absence of the `'$'` in the latter), otherwise it would never be called. Our example for the entries works fine because I thoroughly specified URLs using the `'$'` at the end.

So, models, forms, admin, views and URLs are all done. All that is left to do is take care of the templates. I'll have to be very brief on this part because HTML can be very verbose.

Writing the templates

All templates inherit from a base one, which provides the HTML structure for all others, in a very OOP type of fashion. It also specifies a few blocks, which are areas that can be overridden by children so that they can provide custom content for those areas. Let's start with the base template:

`entries/templates/entries/base.html`

```
{% load static from staticfiles %}
<!DOCTYPE html>
<html lang="en">
```

```
    <head>
      {% block meta %}
        <meta charset="utf-8">
        <meta name="viewport"
         content="width=device-width, initial-scale=1.0">
      {% endblock meta %}

      {% block styles %}
        <link href="{% static "entries/css/main.css" %}"
         rel="stylesheet">
      {% endblock styles %}

      <title> {% block title %}Title{% endblock title %} </title>
    </head>

    <body>
      <div id="page-content">
        {% block page-content %}
        {% endblock page-content %}
      </div>
      <div id="footer">
        {% block footer %}
        {% endblock footer %}
      </div>
    </body>
  </html>
```

There is a good reason to repeat the `entries` folder from the `templates` one. When you deploy a Django website, you collect all the template files under one folder. If you don't specify the paths like I did, you may get a `base.html` template in the entries app, and a `base.html` template in another app. The last one to be collected will override any other file with the same name. For this reason, by putting them in a `templates/entries` folder and using this technique for each Django app you write, you avoid the risk of name collisions (the same goes for any other static file).

There is not much to say about this template, really, apart from the fact that it loads the `static` tag so that we can get easy access to the `static` path without hardcoding it in the template by using `{% static ... %}`. The code in the special `{% ... %}` sections is code that defines logic. The code in the special `{{ ... }}` represents variables that will be rendered on the page.

We define three blocks: `title`, `page-content`, and `footer`, whose purpose is to hold the title, the content of the page, and the footer. Blocks can be optionally overridden by child templates in order to provide different content within them.

Here's the footer:

`entries/templates/entries/footer.html`

```
<div class="footer">
  Go back <a href="{% url "home" %}">home</a>.
</div>
```

It gives us a nice link to the home page.

The home page template is the following:

`entries/templates/entries/home.html`

```
{% extends "entries/base.html" %}
{% block title%}Welcome to the Entry website.{% endblock title %}

{% block page-content %}
  <h1>Welcome {{ user.first_name }}!</h1>

  <div class="home-option">To see the list of your entries
    please click <a href="{% url "entries" %}">here.</a>
  </div>
  <div class="home-option">To insert a new entry please click
    <a href="{% url "insert" %}">here.</a>
  </div>
  <div class="home-option">To login as another user please click
    <a href="{% url "logout" %}">here.</a>
  </div>
    <div class="home-option">To go to the admin panel
    please click <a href="{% url "admin:index" %}">here.</a>
  </div>
{% endblock page-content %}
```

It extends the `base.html` template, and overrides `title` and `page-content`. You can see that basically all it does is provide four links to the user. These are the list of entries, the insert page, the logout page, and the admin page. All of this is done without hardcoding a single URL, through the use of the `{% url ... %}` tag, which is the template equivalent of the `reverse` function.

The template for inserting an `Entry` is as follows:

entries/templates/entries/insert.html

```
{% extends "entries/base.html" %}
{% block title%}Insert a new Entry{% endblock title %}

{% block page-content %}
  {% if messages %}
    {% for message in messages %}
      <p class="{{ message.tags }}">{{ message }}</p>
    {% endfor %}
  {% endif %}

  <h1>Insert a new Entry</h1>
  <form action="{% url "insert" %}" method="post">
    {% csrf_token %}{{ form.as_p }}
    <input type="submit" value="Insert">
  </form><br>
{% endblock page-content %}

{% block footer %}
  <div><a href="{% url "entries" %}">See your entries.</a></div>
  {% include "entries/footer.html" %}
{% endblock footer %}
```

There is some conditional logic at the beginning to display messages, if any, and then we define the form. Django gives us the ability to render a form by simply calling `{{ form.as_p }}` (alternatively, `form.as_ul` or `form.as_table`). This creates all the necessary fields and labels for us. The difference between the three commands is in the way the form is laid out: as a paragraph, as an unordered list or as a table. We only need to wrap it in form tags and add a submit button. This behavior was designed for our convenience; we need the freedom to shape that `<form>` tag as we want, so Django isn't intrusive on that. Also, note that `{% csrf_token %}`. It will be rendered into a token by Django and will become part of the data sent to the server on submission. This way Django will be able to verify that the request was from an allowed source, thus avoiding the aforementioned *cross-site request forgery* issue. Did you see how we handled the token when we wrote the view for the `Entry` insertion? Exactly. We didn't write a single line of code for it. Django takes care of it automatically thanks to a **middleware** class (`CsrfViewMiddleware`). Please refer to the official Django documentation to explore this subject further.

For this page, we also use the footer block to display a link to the home page. Finally, we have the list template, which is the most interesting one.

entries/templates/entries/list.html

```
{% extends "entries/base.html" %}
{% block title%} Entries list {% endblock title %}

{% block page-content %}
 {% if entries %}
  <h1>Your entries ({{ entries|length }} found)</h1>
  <div><a href="{% url "insert" %}">Insert new entry.</a></div>

  <table class="entries-table">
   <thead>
     <tr><th>Entry</th><th>Matches</th></tr>
   </thead>
   <tbody>
    {% for entry, match in entries %}
     <tr class="entries-list {% cycle 'light-gray' 'white' %}">
      <td>
        Pattern: <code class="code">
         "{{ entry.pattern }}"</code><br>
        Test String: <code class="code">
         "{{ entry.test_string }}"</code><br>
        Added: {{ entry.date_added }}
      </td>
      <td>
        {% if match %}
         Group: {{ match.0 }}<br>
         Subgroups:
          {{ match.1|default_if_none:"none" }}<br>
         Group Dict: {{ match.2|default_if_none:"none" }}
        {% else %}
         No matches found.
        {% endif %}
      </td>
     </tr>
    {% endfor %}
   </tbody>
  </table>
 {% else %}
  <h1>You have no entries</h1>
  <div><a href="{% url "insert" %}">Insert new entry.</a></div>
 {% endif %}
{% endblock page-content %}
```

```
{% block footer %}
 {% include "entries/footer.html" %}
{% endblock footer %}
```

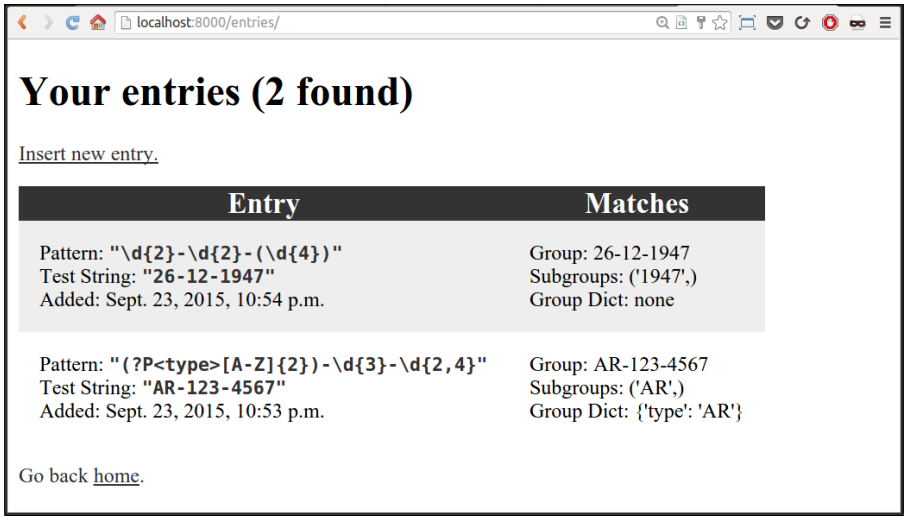
It may take you a while to get used to the template language, but really, all there is to it is the creation of a table using a `for` loop. We start by checking if there are any entries and, if so, we create a table. There are two columns, one for the `Entry`, and the other for the match.

In the `Entry` column, we display the `Entry` object (apart from the user) and in the `Matches` column, we display that 3-tuple we created in the `EntryListView`. Note that to access the attributes of an object, we use the same dot syntax we use in Python, for example `{{ entry.pattern }}` or `{{ entry.test_string }}`, and so on.

When dealing with lists and tuples, we cannot access items using the square brackets syntax, so we use the dot one as well (`{{ match.0 }}` is equivalent to `match[0]`, and so on.). We also use a filter, through the pipe (`|`) operator to display a custom value if a match is `None`.

The Django template language (which is not properly Python) is kept simple for a precise reason. If you find yourself limited by the language, it means you're probably trying to do something in the template that should actually be done in the view, where that logic is more relevant.

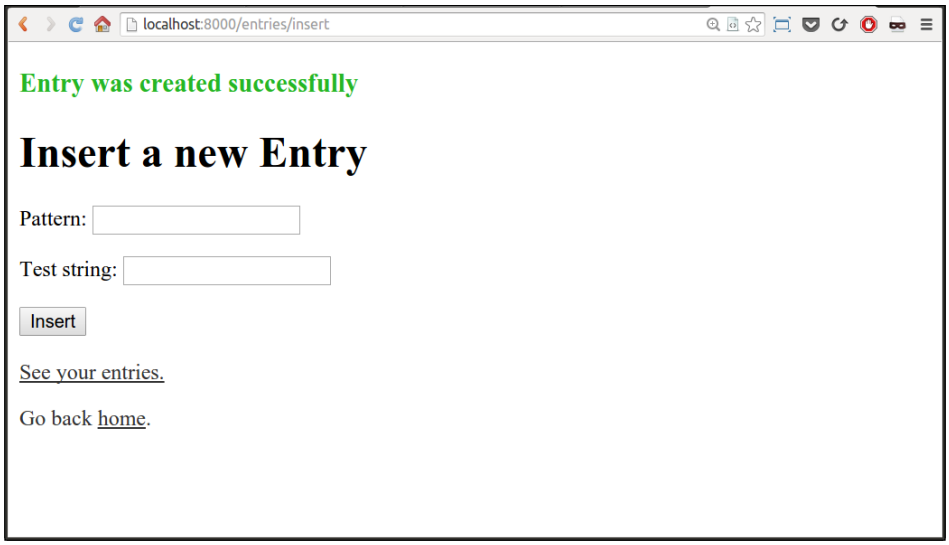
Allow me to show you a couple of screenshots of the *list* and *insert* templates. This is what the list of entries looks like for my father:



Note how the use of the cycle tag alternates the background color of the rows from white to light gray. Those classes are defined in the `main.css` file.

The `Entry` insertion page is smart enough to provide a few different scenarios. When you land on it at first, it presents you with just an empty form. If you fill it in correctly, it will display a nice message for you (see the following picture). However, if you fail to fill in both fields, it will display an error message before them, alerting you that those fields are required.

Note also the custom footer, which includes both a link to the entries list and a link to the home page:



And that's it! You can play around with the CSS styles if you wish. Download the code for the book and have fun exploring and extending this project. Add something else to the model, create and apply a migration, play with the templates, there's lots to do!

Django is a very powerful framework, and offers so much more than what I've been able to show you in this chapter, so you definitely want to check it out. The beauty of it is that it's Python, so reading its source code is a very useful exercise.

The future of web development

Computer science is a very young subject, compared to other branches of science that have existed alongside humankind for centuries or more. One of its main characteristics is that it moves extremely fast. It leaps forward with such speed that, in just a few years, you can see changes that are comparable to real world changes that took a century to happen. Therefore, as a coder, you must pay attention to what happens in this world, all the time.

Something that is happening now is that because powerful computers are now quite cheap and almost everyone has access to them, the trend is to try and avoid putting too much workload on the backend, and let the frontend handle part of it. Therefore, in the last few years, JavaScript frameworks and libraries like jQuery and Backbone have become very popular and web development has shifted from a paradigm where the backend takes care of handling data, preparing it, and serving it to the frontend to display it, to a paradigm where the backend is sometimes just used as an API, a sheer data provider. The frontend fetches the data from the backend with an API call, and then it takes care of the rest. This shift facilitates the existence of paradigms like **Single-Page Application** (**SPA**), where, ideally, the whole page is loaded once and then evolves, based on the content that usually comes from the backend. E-commerce websites that load the results of a search in a page that doesn't refresh the surrounding structure, are made with similar techniques. Browsers can perform asynchronous calls (**AJAX**) that can return data which can be read, manipulated and injected back into the page with JavaScript code.

So, if you're planning to work on web development, I strongly suggest you to get acquainted with JavaScript (if you're not already), and also with APIs. In the last few pages of this chapter, I'll give you an example of how to make a simple API using two different Python microframeworks: Flask and Falcon.

Writing a Flask view

Flask (`http://flask.pocoo.org/`) is a Python microframework. It provides fewer features than Django, but it's supposedly faster and quicker to get up and running. To be honest, getting Django up and running nowadays is also very quickly done, but Flask is so popular that it's good to see an example of it, nonetheless.

In your `ch10` folder, create a `flask` folder with the following structure:

```
$ tree -A flask  # from the ch10 folder
flask
├── main.py
└── templates
    └── main.html
```

Basically, we're going to code two simple files: a Flask application and an HTML template. Flask uses Jinja2 as template engine. It's extremely popular and very fast, and just recently even Django has started to offer native support for it, which is something that Python coders have longed for, for a long time.

flask/templates/main.html

```
<!doctype html>
<title>Hello from Flask</title>
<h1>
  {% if name %}
    Hello {{ name }}!
  {% else %}
    Hello shy person!
  {% endif %}
</h1>
```

The template is almost offensively simple; all it does is to change the greeting according to the presence of the `name` variable. A bit more interesting is the Flask application that renders it:

flask/main.py

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
@app.route('/<name>')
def hello(name=None):
    return render_template('main.html', name=name)

if __name__ == '__main__':
    app.run()
```

We create an `app` object, which is a Flask application. We only feed the fully-qualified name of the module, which is stored in `__name__`.

Then, we write a simple `hello` view, which takes an optional `name` argument. In the body of the view, we simply render the `main.html` template, passing to it the `name` argument, regardless of its value.

What's interesting is the routing. Differently from Django's way of tying up views and URLs (the `urls.py` module), in Flask you decorate your view with one or more `@app.route` decorators. In this case, we accept both the root URL without anything else, or with name information.

Change into the `flask` folder and type (make sure you have Flask installed with `$ pip install flask`):

```
$ python main.py
```

You can open a browser and go to `http://127.0.0.1:5000/`. This URL has no name information; therefore, you will see **Hello shy person!** It is written all nice and big. Try to add something to that URL like `http://127.0.0.1:5000/Adriano`. Hit *Enter* and the page will change to **Hello Adriano!**.

Of course, Flask offers you much more than this but we don't have the room to see a more complex example. It's definitely worth exploring, though. Several projects use it successfully and it's fun and it is nice to create websites or APIs with it. Flask's author, Armin Ronacher, is a successful and very prolific coder. He also created or collaborated on several other interesting projects like Werkzeug, Jinja2, Click, and Sphinx.

Building a JSON quote server in Falcon

Falcon (`http://falconframework.org/`) is another microframework written in Python, which was designed to be light, fast and flexible. I think this relatively young project will evolve to become something really popular due to its speed, which is impressive, so I'm happy to show you a tiny example using it.

We're going to build a view that returns a randomly chosen quote from the *Buddha*.

In your `ch10` folder, create a new one called `falcon`. We'll have two files: `quotes.py` and `main.py`. To run this example, install Falcon and Gunicorn (`$ pip install falcon gunicorn`). Falcon is the framework, and **Gunicorn** (**Green Unicorn**) is a Python WSGI HTTP Server for Unix (which, in layman terms, means the technology that is used to run the server). When you're all set up, start by creating the `quotes.py` file.

`falcon/quotes.py`

```
quotes = [
    "Thousands of candles can be lighted from a single candle, "
    "and the life of the candle will not be shortened. "
    "Happiness never decreases by being shared.",
    ...
    "Peace comes from within. Do not seek it without.",
]
```

You will find the complete list of quotes in the source code for this book. If you don't have it, you can also fill in your favorite quotes. Note that not every line has a comma at the end. In Python, it's possible to concatenate strings like that, as long as they are in brackets (or braces). It's called **implicit concatenation**.

The code for the main app is not long, but it is interesting:

falcon/main.py

```
import json
import random
import falcon
from quotes import quotes

class QuoteResource:
    def on_get(self, req, resp):
        quote = {
            'quote': random.choice(quotes),
            'author': 'The Buddha'
        }
        resp.body = json.dumps(quote)

api = falcon.API()
api.add_route('/quote', QuoteResource())
```

Let's start with the class. In Django we had a `get` method, in Flask we defined a function, and here we write an `on_get` method, a naming style that reminds me of C# event handlers. It takes a request and a response argument, both automatically fed by the framework. In its body, we define a dict with a randomly chosen quote, and the author information. Then we dump that dict to a JSON string and set the response body to its value. We don't need to return anything, Falcon will take care of it for us.

At the end of the file, we create the Falcon application, and we call `add_route` on it to tie the handler we have just written to the URL we want.

When you're all set up, change to the `falcon` folder and type:

```
$ gunicorn main:api
```

Then, make a request (or simply open the page with your browser) to `http://127.0.0.1:8000/quote`. When I did it, I got this JSON in response:

```
{
    quote: "The mind is everything. What you think you become.",
    author: "The Buddha"
}
```

Whatever the framework you end up using for your web development, try and keep yourself informed about other choices too. Sometimes you may be in situations where a different framework is the right way to go, and having a working knowledge of different tools will give you an advantage.

Summary

In this chapter, we caught a glimpse of web development. We talked about important concepts like the DRY philosophy and the concept of a framework as a tool that provides us with many things we need in order to write code to serve requests. We also talked about the MTV pattern, and how nicely these three layers play together to realize a request-response path.

Later on, we briefly introduced regular expressions, which is a subject of paramount importance, and it's the layer which provides the tools for URL routing.

There are many different frameworks out there, and Django is definitely one of the best and most widely used, so it's definitely worth exploring, especially its source code, which is very well written.

There are other very interesting and important frameworks too, like Flask. They provide fewer features but, in general, they are faster, both in execution time and to set up. One that is extremely fast is the relatively young Falcon project, whose benchmarks are outstanding.

It's important to get a solid understanding of how the request-response mechanism works, and how the Web in general works, so that eventually it won't matter too much which framework you have to use. You will be able to pick it up quickly because it will only be a matter of getting familiar with a way of doing something you already know a lot about.

Explore at least three frameworks and try to come up with different use cases to decide which one of them could be the ideal choice. When you are able to make that choice, you will know you have a good enough understanding of them.

The next chapter is about debugging and troubleshooting. We'll learn how to deal with errors and issues so that if you get in trouble when coding (don't worry, normally it only happens about all the time), you will be able to quickly find the solution to your problem and move on.

11
Debugging and Troubleshooting

"If debugging is the process of removing software bugs, then programming must be the process of putting them in."

- Edsger W. Dijkstra

In the life of a professional coder, debugging and troubleshooting take up a significant amount of time. Even if you work on the most beautiful codebase ever written by man, there will still be bugs in it, that is guaranteed.

We spend an awful lot of time reading other people's code and, in my opinion, a good software developer is someone who keeps their attention high, even when they're reading code that is not reported to be wrong or buggy.

Being able to debug code efficiently and quickly is a skill that any coder needs to keep improving. Some think that because they have read the manual, they're fine, but the reality is, the number of variables in the game is so big that there is no manual. There are guidelines that one can follow, but there is no magic book that will teach you everything you need to know in order to become good at this.

I feel that on this particular subject, I have learned the most from my colleagues. It amazes me to observe someone very skilled attacking a problem. I enjoy seeing the steps they take, the things they verify to exclude possible causes, and the way they consider the suspects that eventually lead them to the solution to the problem.

Every colleague we work with can teach us something, or surprise us with a fantastic guess that turns out to be the right one. When that happens, don't just remain in wonderment (or worse, in envy), but seize the moment and ask them how they got to that guess and why. The answer will allow you to see if there is something you can study in deep later on so that, maybe next time, you'll be the one who will catch the bug.

Some bugs are very easy to spot. They come out of coarse mistakes and, once you see the effects of those mistakes, it's easy to find a solution that fixes the problem.

But there are other bugs which are much more subtle, much more slippery, and require true expertise, and a great deal of creativity and out-of-the-box thinking, to be dealt with.

The worst of all, at least for me, are the nondeterministic ones. These sometimes happen, and sometimes don't. Some happen only in environment A but not in environment B, even though A and B are supposed to be exactly the same. Those bugs are the true evil ones, and they can drive you crazy.

And of course, bugs don't just happen in the sandbox, right? With your boss telling you "*don't worry! take your time to fix this, have lunch first!*". Nope. They happen on a Friday at half past five, when your brain is cooked and you just want to go home. It's in those moments, when everyone is getting upset in a split second, when your boss is breathing on your neck, that you have to be able to keep calm. And I do mean it. That's the most important skill to have if you want to be able to fight bugs effectively. If you allow your mind to get stressed, say goodbye to creative thinking, to logic deduction, and to everything you need at that moment. So take a deep breath, sit properly, and focus.

In this chapter, I will try to demonstrate some useful techniques that you can employ according to the severity of the bug, and a few suggestions that will hopefully boost your weapons against bugs and issues.

Debugging techniques

In this part, I'll present you with the most common techniques, the ones I use most often, however, please don't consider this list to be exhaustive.

Debugging with print

This is probably the easiest technique of all. It's not very effective, it cannot be used everywhere and it requires access to both the source code and a terminal that will run it (and therefore show the results of the print function calls).

However, in many situations, this is still a quick and useful way to debug. For example, if you are developing a Django website and what happens in a page is not what would you expect, you can fill the view with prints and keep an eye on the console while you reload the page. I've probably done it a million times.

When you scatter calls to `print` in your code, you normally end up in a situation where you duplicate a lot of debugging code, either because you're printing a timestamp (like we did when we were measuring how fast list comprehensions and generators were), or because you have to somehow build a string of some sort that you want to display.

Another issue is that it's extremely easy to forget calls to `print` in your code.

So, for these reasons, rather than using a bare call to `print`, I sometimes prefer to code a custom function. Let's see how.

Debugging with a custom function

Having a custom function in a snippet that you can quickly grab and paste into the code, and then use to debug, can be very useful. If you're fast, you can always code one on the fly. The important thing is to code it in a way that it won't leave stuff around when you eventually remove the calls and its definition, therefore *it's important to code it in a way that is completely self-contained*. Another good reason for this requirement is that it will avoid potential name clashes with the rest of the code.

Let's see an example of such a function.

custom.py

```
def debug(*msg, print_separator=True):
    print(*msg)
    if print_separator:
        print('-' * 40)

debug('Data is ...')
debug('Different', 'Strings', 'Are not a problem')
debug('After while loop', print_separator=False)
```

In this case, I am using a keyword-only argument to be able to print a separator, which is a line of 40 dashes.

The function is very simple, I just redirect whatever is in `msg` to a call to `print` and, if `print_separator` is `True`, I print a line separator. Running the code will show:

```
$ python custom.py
Data is ...
----------------------------------------
Different Strings Are not a problem
----------------------------------------
After while loop
```

As you can see, there is no separator after the last line.

This is just one easy way to somehow augment a simple call to the `print` function. Let's see how we can calculate a time difference between calls, using one of Python's tricky features to our advantage.

custom_timestamp.py

```
from time import sleep

def debug(*msg, timestamp=[None]):
    print(*msg)
    from time import time  # local import
    if timestamp[0] is None:
        timestamp[0] = time()  #1
    else:
        now = time()
        print(' Time elapsed: {:.3f}s'.format(
            now - timestamp[0]))
        timestamp[0] = now  #2

debug('Entering nasty piece of code...')
sleep(.3)
debug('First step done.')
sleep(.5)
debug('Second step done.')
```

This is a bit trickier, but still quite simple. First notice we import the `time` function from the `time` module from the `debug` function. This allows us to avoid having to add that import outside of the function, and maybe forget it there.

Take a look at how I defined `timestamp`. It's a list, of course, but what's important here is that it is a *mutable* object. This means that it will be set up when Python parses the function and it will retain its value throughout different calls. Therefore, if we put a timestamp in it after each call, we can keep track of time without having to use an external global variable. I borrowed this trick from my studies on **closures**, a technique that I encourage you to read about because it's very interesting.

Right, so, after having printed whatever message we had to print and importing time, we then inspect the content of the only item in `timestamp`. If it is `None`, we have no previous reference, therefore we set the value to the current time (`#1`).

On the other hand, if we have a previous reference, we can calculate a difference (which we nicely format to three decimal digits) and then we finally put the current time again in `timestamp` (`#2`). It's a nice trick, isn't it?

Running this code shows this result:

```
$ python custom_timestamp.py
Entering nasty piece of code...
First step done.
 Time elapsed: 0.300s
Second step done.
 Time elapsed: 0.501s
```

Whatever is your situation, having a self contained function like this can be very useful.

Inspecting the traceback

We briefly talked about the traceback in *Chapter 7*, *Testing, Profiling, and Dealing with Exceptions* when we saw several different kinds of exceptions. The traceback gives you information about what happened in your application that went wrong. You get a great help from reading it. Let's see a very small example:

`traceback_simple.py`

```
d = {'some': 'key'}
key = 'some-other'
print(d[key])
```

We have a dict and we have tried to access a key which isn't in it. You should remember that this will raise a `KeyError` exception. Let's run the code:

```
$ python traceback_simple.py
Traceback (most recent call last):
  File "traceback_simple.py", line 3, in <module>
    print(d[key])
KeyError: 'some-other'
```

You can see that we get all the information we need: the module name, the line that caused the error (both the number and the instruction), and the error itself. With this information, you can go back to the source code and try and understand what's going wrong.

Let's now create a more interesting example that builds on this, and exercises a feature that is only available in Python 3. Imagine that we're validating a dict, working on mandatory fields, therefore we expect them to be there. If not, we need to raise a custom `ValidationError`, that we will trap further upstream in the process that runs the validator (which is not shown here, it could be anything, really). It should be something like this:

`traceback_validator.py`

```
class ValidatorError(Exception):
    """Raised when accessing a dict results in KeyError. """

d = {'some': 'key'}
mandatory_key = 'some-other'
try:
    print(d[mandatory_key])
except KeyError:
    raise ValidatorError(
        '`{}` not found in d.'.format(mandatory_key))
```

We define a custom exception that is raised when the mandatory key isn't there. Note that its body consists of its documentation string so we don't need to add any other statements.

Very simply, we define a dummy dict and try to access it using `mandatory_key`. We trap the `KeyError` and raise `ValidatorError` when that happens. The purpose of doing this is that we may also want to raise `ValidatorError` in other circumstances, not necessarily as a consequence of a mandatory key being missing. This technique allows us to run the validation in a simple `try/except` that only cares about `ValidatorError`.

The thing is, in Python 2, this code would just display the last exception (`ValidatorError`), which means we would lose the information about the `KeyError` that precedes it. In Python 3, this behavior has changed and exceptions are now chained so that you have a much better information report when something happens. The code produces this result:

```
$ python traceback_validator.py
Traceback (most recent call last):
  File "traceback_validator.py", line 7, in <module>
    print(d[mandatory_key])
KeyError: 'some-other'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "traceback_validator.py", line 10, in <module>
    '`{}` not found in d.'.format(mandatory_key))
__main__.ValidatorError: `some-other` not found in d.
```

This is brilliant, because we can see the traceback of the exception that led us to raise `ValidationError`, as well as the traceback for the `ValidationError` itself.

I had a nice discussion with one of my reviewers about the traceback you get from the `pip` installer. He was having trouble setting everything up in order to review the code for *Chapter 9*, *Data Science*. His fresh Ubuntu installation was missing a few libraries that were needed by the `pip` packages in order to run correctly.

The reason he was blocked was that he was trying to fix the errors displayed in the traceback starting from the top one. I suggested that he started from the bottom one instead, and fix that. The reason was that, if the installer had gotten to that last line, I guess that before that, whatever error may have occurred, it was still possible to recover from it. Only after the last line, `pip` decided it wasn't possible to continue any further, and therefore I started fixing that one. Once the libraries required to fix that error had been installed, everything else went smoothly.

Reading a traceback can be tricky, and my friend was lacking the necessary experience to address this problem correctly, therefore, if you end up in the same situation, don't be discouraged, and try to shake things up a bit, don't take anything for granted.

Python has a huge and wonderful community and it's very unlikely that, when you encounter a problem, you're the first one to see it, so open a browser and search. By doing so, your searching skills will also improve because you will have to trim the error down to the minimum but essential set of details that will make your search effective.

If you want to play and understand the traceback a bit better, in the standard library there is a module called, surprise surprise, `traceback` that you can use. It provides a standard interface to extract, format, and print stack traces of Python programs, mimicking exactly the behavior of the Python interpreter when it prints a stack trace.

Using the Python debugger

Another very effective way of debugging Python is to use the Python debugger: **pdb**. If you are addicted to the IPython console, like me, you should definitely check out the **ipdb** library. *ipdb* augments the standard *pdb* interface like IPython does with the Python console.

There are several different ways of using this debugger (whichever version, it is not important), but the most common one consists of simply setting a breakpoint and running the code. When Python reaches the breakpoint, execution is suspended and you get console access to that point so that you can inspect all the names, and so on. You can also alter data on the fly to change the flow of the program.

As a toy example, let's pretend we have a parser that is raising a `KeyError` because a key is missing in a dict. The dict is from a JSON payload that we cannot control, and we just want, for the time being, to cheat and pass that control, since we're interested in what comes afterwards. Let's see how we could intercept this moment, inspect the data, fix it and get to the bottom, with *ipdb*.

`ipdebugger.py`

```
# d comes from a JSON payload we don't control
d = {'first': 'v1', 'second': 'v2', 'fourth': 'v4'}
# keys also comes from a JSON payload we don't control
keys = ('first', 'second', 'third', 'fourth')

def do_something_with_value(value):
    print(value)

for key in keys:
    do_something_with_value(d[key])

print('Validation done.')
```

As you can see, this code will break when `key` gets the value `'third'`, which is missing in the dict. Remember, we're pretending that both `d` and `keys` come dynamically from a JSON payload we don't control, so we need to inspect them in order to fix `d` and pass the `for` loop. If we run the code as it is, we get the following:

```
$ python ipdebugger.py
v1
v2
Traceback (most recent call last):
  File "ipdebugger.py", line 10, in <module>
    do_something_with_value(d[key])
KeyError: 'third'
```

So we see that that `key` is missing from the dict, but since every time we run this code we may get a different dict or `keys` tuple, this information doesn't really help us. Let's inject a call to *ipdb*.

ipdebugger_ipdb.py

```
# d comes from a JSON payload we don't control
d = {'first': 'v1', 'second': 'v2', 'fourth': 'v4'}
# keys also comes from a JSON payload we don't control
keys = ('first', 'second', 'third', 'fourth')

def do_something_with_value(value):
    print(value)

import ipdb
ipdb.set_trace()  # we place a breakpoint here

for key in keys:
    do_something_with_value(d[key])

print('Validation done.')
```

If we now run this code, things get interesting (note that your output may vary a little and that all the comments in this output were added by me):

```
$ python ipdebugger_ipdb.py
> /home/fab/srv/l.p/ch11/ipdebugger_ipdb.py(12)<module>()
     11
---> 12 for key in keys:  # this is where the breakpoint comes
     13     do_something_with_value(d[key])
```

```
ipdb> keys  # let's inspect the keys tuple
('first', 'second', 'third', 'fourth')
ipdb> !d.keys()  # now the keys of d
dict_keys(['first', 'fourth', 'second'])  # we miss 'third'
ipdb> !d['third'] = 'something dark side...'  # let's put it in
ipdb> c  # ... and continue
v1
v2
something dark side...
v4
Validation done.
```

This is very interesting. First, note that, when you reach a breakpoint, you're served a console that tells you where you are (the Python module) and which line is the next one to be executed. You can, at this point, perform a bunch of exploratory actions, such as inspecting the code before and after the next line, printing a stacktrace, interacting with the objects, and so on. Please consult the official Python documentation on *pdb* to learn more about this. In our case, we first inspect the `keys` tuple. After that, we inspect the keys of `d`.

Have you noticed that exclamation mark I prepended to `d`? It's needed because `d` is a command in the *pdb* interface that moves the frame (*d*)own.

I indicate commands within the *ipdb* shell with this notation: each command is activated by one letter, which typically is the first letter of the command name. So, *d* for *down*, *n* for *next*, and *s* for *step* become, more concisely, (*d*)own, (*n*)ext and (*s*)tep.

I guess this is a good enough reason to have better names, right? Indeed, but I needed to show you this, so I chose to use `d`. In order to tell *pdb* that we're not yielding a (*d*)own command, we put `"!"` in front of `d` and we're fine.

After seeing the keys of `d`, we see that `'third'` is missing, so we put it in ourselves (could this be dangerous? think about it). Finally, now that all the keys are in, we type `c`, which means (*c*)ontinue.

pdb also gives you the ability to proceed with your code one line at a time using (*n*)ext, to (*s*)tep into a function for deeper analysis, or handling breaks with (*b*)reak. For a complete list of commands, please refer to the documentation or type (*h*)elp in the console.

You can see from the output that we could finally get to the end of the validation.

pdb (or *ipdb*) are invaluable tools that I use every day, I couldn't live without them. So, go and have fun, set a breakpoint somewhere and try and inspect, follow the official documentation and try the commands in your code to see their effect and learn them well.

Inspecting log files

Another way of debugging a misbehaving application is to inspect its log files. **Log files** are special files in which an application writes down all sorts of things, normally related to what's going on inside of it. If an important procedure is started, I would typically expect a line for that in the logs. It is the same when it finishes, and possibly for what happens inside of it.

Errors need to be logged so that when a problem happens we can inspect what went wrong by taking a look at the information in the log files.

There are many different ways to set up a logger in Python. Logging is very malleable and you can configure it. In a nutshell, there are normally four players in the game: loggers, handlers, filters, and formatters:

- **Loggers** expose the interface that the application code uses directly
- **Handlers** send the log records (created by loggers) to the appropriate destination
- **Filters** provide a finer grained facility for determining which log records to output
- **Formatters** specify the layout of the log records in the final output

Logging is performed by calling methods on instances of the `Logger` class. Each line you log has a level. The levels normally used are: `DEBUG`, `INFO`, `WARNING`, `ERROR`, and `CRITICAL`. You can import them from the `logging` module. They are in order of severity and it's very important to use them properly because they will help you filter the contents of a log file based on what you're searching for. Log files usually become extremely big so it's very important to have the information in them written properly so that you can find it quickly when it matters.

You can log to a file but you can also log to a network location, to a queue, to a console, and so on. In general, if you have an architecture that is deployed on one machine, logging to a file is acceptable, but when your architecture spans over multiple machines (such as in the case of **service-oriented architectures**), it's very useful to implement a centralized solution for logging so that all log messages coming from each service can be stored and investigated in a single place. It helps a lot, otherwise you can really go crazy trying to correlate giant files from several different sources to figure out what went wrong.

A **service-oriented architecture** (**SOA**) is an architectural pattern in software design in which application components provide services to other components via a communications protocol, typically over a network. The beauty of this system is that, when coded properly, each service can be written in the most appropriate language to serve its purpose. The only thing that matters is the communication with the other services, which needs to happen via a common format so that data exchange can be done.

Here, I will present you with a very simple logging example. We will log a few messages to a file:

`log.py`

```
import logging

logging.basicConfig(
    filename='ch11.log',
    level=logging.DEBUG,  # minimum level capture in the file
    format='[%(asctime)s] %(levelname)s:%(message)s',
    datefmt='%m/%d/%Y %I:%M:%S %p')

mylist = [1, 2, 3]
logging.info('Starting to process `mylist`...')

for position in range(4):
    try:
        logging.debug('Value at position {} is {}'.format(
            position, mylist[position]))
    except IndexError:
        logging.exception('Faulty position: {}'.format(position))

logging.info('Done parsing `mylist`.')
```

Let's go through it line by line. First, we import the `logging` module, then we set up a basic configuration. In general, a production logging configuration is much more complicated than this, but I wanted to keep things as easy as possible. We specify a filename, the minimum logging level we want to capture in the file, and the message format. We'll log the date and time information, the level, and the message.

I will start by logging an `info` message that tells me we're about to process our list. Then, I will log (this time using the `DEBUG` level, by using the `debug` function) which is the value at some position. I'm using `debug` here because I want to be able to filter out these logs in the future (by setting the minimum level to `logging.INFO` or more), because I might have to handle very big lists and I don't want to log all the values.

If we get an `IndexError` (and we do, since I'm looping over `range(4)`), we call `logging.exception()`, which is the same as `logging.error()`, but it also prints the traceback.

At the end of the code, I log another `info` message saying we're done. The result is this:

```
[10/08/2015 04:17:06 PM] INFO:Starting to process `mylist`...
[10/08/2015 04:17:06 PM] DEBUG:Value at position 0 is 1
[10/08/2015 04:17:06 PM] DEBUG:Value at position 1 is 2
[10/08/2015 04:17:06 PM] DEBUG:Value at position 2 is 3
[10/08/2015 04:17:06 PM] ERROR:Faulty position: 3
Traceback (most recent call last):
  File "log.py", line 15, in <module>
    position, mylist[position]))
IndexError: list index out of range
[10/08/2015 04:17:06 PM] INFO:Done parsing `mylist`.
```

This is exactly what we need to be able to debug an application that is running on a box, and not on our console. We can see what went on, the traceback of any exception raised, and so on.

The example presented here only scratches the surface of logging. For a more in-depth explanation, you can find a very nice introduction in the how to (`https://docs.python.org/3.4/howto/logging.html`) section of the official Python documentation.

Logging is an art, you need to find a good balance between logging everything and logging nothing. Ideally, you should log anything that you need to make sure your application is working correctly, and possibly all errors or exceptions.

Other techniques

In this final section, I'd like to demonstrate briefly a couple of techniques that you may find useful.

Profiling

We talked about profiling in *Chapter 7*, *Testing, Profiling, and Dealing with Exceptions*, and I'm only mentioning it here because profiling can sometimes explain weird errors that are due to a component being too slow. Especially when networking is involved, having an idea of the timings and latencies your application has to go through is very important in order to understand what may be going on when problems arise, therefore I suggest you get acquainted with profiling techniques also for a troubleshooting perspective.

Assertions

Assertions are a nice way to make your code ensure your assumptions are verified. If they are, all proceeds regularly but, if they are not, you get a nice exception that you can work with. Sometimes, instead of inspecting, it's quicker to drop a couple of assertions in the code just to exclude possibilities. Let's see an example:

`assertions.py`

```
mylist = [1, 2, 3]  # this ideally comes from some place
assert 4 == len(mylist)  # this will break
for position in range(4):
    print(mylist[position])
```

This code simulates a situation in which `mylist` isn't defined by us like that, of course, but we're assuming it has four elements. So we put an assertion there, and the result is this:

```
$ python assertions.py
Traceback (most recent call last):
  File "assertions.py", line 3, in <module>
    assert 4 == len(mylist)
AssertionError
```

This tells us exactly where the problem is.

Where to find information

In the Python official documentation, there is a section dedicated to debugging and profiling, where you can read up about the `bdb` debugger framework, and about modules such as `faulthandler`, `timeit`, `trace`, `tracemallock`, and of course *pdb*. Just head to the standard library section in the documentation and you'll find all this information very easily.

Troubleshooting guidelines

In this short section, I'll like to give you a few tips that come from my troubleshooting experience.

Using console editors

First, get comfortable using **vim** or **nano** as an editor, and learn the basics of the console. When things break bad you don't have the luxury of your editor with all the bells and whistles there. You have to connect to a box and work from there. So it's a very good idea to be comfortable browsing your production environment with console commands, and be able to edit files using console-based editors such as vi, vim, or nano. Don't let your usual development environment spoil you, because you'll have to pay a price if you do.

Where to inspect

My second suggestion is on where to place your debugging breakpoints. It doesn't matter if you are using `print`, a custom function, or *ipdb*, you still have to choose where to place the calls that provide you with the information, right?

Well, some places are better than others, and there are ways to handle the debugging progression that are better than others.

I normally avoid placing a breakpoint in an `if` clause because, if that clause is not exercised, I lose the chance of getting the information I wanted. Sometimes it's not easy or quick to get to the breakpoint, so think carefully before placing them.

Another important thing is where to start. Imagine that you have 100 lines of code that handle your data. Data comes in at line 1, and somehow it's wrong at line 100. You don't know where the bug is, so what do you do? You can place a breakpoint at line 1 and patiently go through all the lines, checking your data. In the worst case scenario, 99 lines later (and many coffee cups) you spot the bug. So, consider using a different approach.

You start at line 50, and inspect. If the data is good, it means the bug happens later, in which case you place your next breakpoint at line 75. If the data at line 50 is already bad, you go on by placing a breakpoint at line 25. Then, you repeat. Each time, you move either backwards or forwards, by half the jump you did last time.

In our worst case scenario, your debugging would go from 1, 2, 3, ..., 99 to 50, 75, 87, 93, 96, ..., 99 which is way faster. In fact, it's logarithmic. This searching technique is called **binary search**, it's based on a divide and conquer approach and it's very effective, so try to master it.

Using tests to debug

Do you remember *Chapter 7, Testing, Profiling, and Dealing with Exceptions*, about tests? Well, if we have a bug and all tests are passing, it means something is wrong or missing in our test codebase. So, one approach is to modify the tests in such a way that they cater for the new edge case that has been spotted, and then work your way through the code. This approach can be very beneficial, because it makes sure that your bug will be covered by a test when it's fixed.

Monitoring

Monitoring is also very important. Software applications can go completely crazy and have non-deterministic hiccups when they encounter edge case situations such as the network being down, a queue being full, an external component being unresponsive, and so on. In these cases, it's important to have an idea of what was the big picture when the problem happened and be able to correlate it to something related to it in a subtle, perhaps mysterious way.

You can monitor API endpoints, processes, web pages availability and load time, and basically almost everything that you can code. In general, when starting an application from scratch, it can be very useful to design it keeping in mind how you want to monitor it.

Summary

In this short chapter, we saw different techniques and suggestions to debug and troubleshoot our code. Debugging is an activity that is always part of a software developer's work, so it's important to be good at it.

If approached with the correct attitude, it can be fun and rewarding.

We saw techniques to inspect our code base on functions, logging, debuggers, traceback information, profiling, and assertions. We saw simple examples of most of them and we also talked about a set of guidelines that will help when it comes to face the fire.

Just *remember to always stay calm and focused*, and debugging will be easier already. This too, is a skill that needs to be learned and it's the most important. An agitated and stressed mind cannot work properly, logically and creatively, therefore, if you don't strengthen it, it will be hard for you to put all of your knowledge to good use.

In the next chapter, we will end the book with another small project whose goal is to leave you more thirsty than you were when you started this journey with me.

Ready?

12
Summing Up – A Complete Example

"Do not dwell in the past, do not dream of the future, concentrate the mind on the present moment."

– The Shakyamuni Buddha

In this chapter, I will show you one last project. If you've worked well in the rest of the book, this example should be easy. I tried my best to craft it in a way that it will neither be too hard for those who have only read the book, nor too simple for those who also took the time to work on the examples, and maybe have read up on the links and topics I suggested.

The challenge

One problem that we all have these days is remembering passwords. We have passwords for everything: websites, phones, cards, bank accounts, and so on. The amount of information we have to memorize is just too much, so many people end up using the same password over and over again. This is very bad, of course, so at some point, tools were invented to alleviate this problem. One of these tools is called **KeepassX**, and basically it works like this: you start the software by setting up a special password called **master password**. Once inside, you store a record for each password you need to memorize, for example, your e-mail account, the bank website, credit card information, and so on. When you close the software, it encrypts the database used to store all that information, so that the data can only be accessed by the owner of the master password. Therefore, kind of in a *Lord of The Rings* fashion, by just owning one password, you rule them all.

Our implementation

Our goal in this chapter is to create something similar but web-based, and the way I want to implement it is by writing two applications.

One will be an API written in Falcon. Its purpose will be twofold, it will be able to both generate and validate passwords. It will provide the caller with information about the validity and a score which should indicate how strong the password is.

The second application is a Django website, which will provide the interface to handle records. Each record will retain information such as the username, e-mail, password, URL, and so on. It will show a list of all the records, and it will allow the user to create, update and delete them. Passwords will be encrypted before being stored in the database.

The purpose of the whole project is, therefore, to mimic the way KeepassX works, even though it is in a much simpler fashion. It will be up to you, if you like this idea, to develop it further in order to add other features and make it more secure. I will make sure to give you some suggestions on how to extend it, towards the end.

This chapter will therefore be quite dense, code-wise. It's the price I have to pay for giving you an interesting example in a restricted amount of space.

Before we start, please make sure you are comfortable with the projects presented in *Chapter 10*, *Web Development Done Right* so that you're familiar with the basics of web development. Make sure also that you have installed all the `pip` packages needed for this project: `django`, `falcon`, `cryptography`, and `nose-parameterized`. If you download the source code for the book, you'll find everything you need to install in the `requirements` folder, while the code for this chapter will be in `ch12`.

Implementing the Django interface

I hope you're comfortable with the concepts presented in *Chapter 10*, *Web Development Done Right* which was mostly about Django. If you haven't read it, this is probably a good time, before reading on here.

The setup

In your root folder (`ch12`, for me), which will contain the root for the interface and the root for the API, start by running this command:

```
$ django-admin startproject pwdweb
```

This will create the structure for a Django project, which we know well by now. I'll show you the final structure of the interface project here:

```
$ tree -A pwdweb
pwdweb
├── db.sqlite3
├── manage.py
├── pwdweb
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── records
    ├── admin.py
    ├── forms.py
    ├── __init__.py
    ├── migrations
    │   ├── 0001_initial.py
    │   └── __init__.py
    ├── models.py
    ├── static
    │   └── records
    │       ├── css
    │       │   └── main.css
    │       └── js
    │           ├── api.js
    │           └── jquery-2.1.4.min.js
    ├── templates
    │   └── records
    │       ├── base.html
    │       ├── footer.html
    │       ├── home.html
    │       ├── list.html
    │       ├── messages.html
    │       ├── record_add_edit.html
    │       └── record_confirm_delete.html
    ├── templatetags
    │   └── record_extras.py
    ├── urls.py
    └── views.py
```

As usual, don't worry if you don't have all the files, we'll add them gradually. Change to the `pwdweb` folder, and make sure Django is correctly set up: `$ python manage.py runserver` (ignore the warning about unapplied migrations).

Shut down the server and create an app: `$ python manage.py startapp records`. That is excellent, now we can start coding. First things first, let's open `pwdweb/settings.py` and start by adding `'records'`, at the end of the `INSTALLED_APP` tuple (note that the comma is included in the code). Then, go ahead and fix the `LANGUAGE_CODE` and `TIME_ZONE` settings according to your preference and finally, add the following line at the bottom:

```
ENCRYPTION_KEY = b'qMhPGx-ROWUDr4veh0ybPRL6viIUNe0vcPDmy67x6CQ='
```

This is a custom encryption key that has nothing to do with Django settings, but we will need it later on, and this is the best place for it to be. Don't worry for now, we'll get back to it.

The model layer

We need to add just one model for the records application: `Record`. This model will represent each record we want to store in the database:

`records/models.py`

```
from cryptography.fernet import Fernet
from django.conf import settings
from django.db import models

class Record(models.Model):
    DEFAULT_ENCODING = 'utf-8'

    title = models.CharField(max_length=64, unique=True)
    username = models.CharField(max_length=64)
    email = models.EmailField(null=True, blank=True)
    url = models.URLField(max_length=255, null=True, blank=True)
    password = models.CharField(max_length=2048)
    notes = models.TextField(null=True, blank=True)
    created = models.DateTimeField(auto_now_add=True)
    last_modified = models.DateTimeField(auto_now=True)

    def encrypt_password(self):
        self.password = self.encrypt(self.password)

    def decrypt_password(self):
        self.password = self.decrypt(self.password)

    def encrypt(self, plaintext):
        return self.cypher('encrypt', plaintext)
```

```
    def decrypt(self, cyphertext):
        return self.cypher('decrypt', cyphertext)

    def cypher(self, cypher_func, text):
        fernet = Fernet(settings.ENCRYPTION_KEY)
        result = getattr(fernet, cypher_func)(
            self._to_bytes(text))
        return self._to_str(result)

    def _to_str(self, bytes_str):
        return bytes_str.decode(self.DEFAULT_ENCODING)

    def _to_bytes(self, s):
        return s.encode(self.DEFAULT_ENCODING)
```

Firstly, we set the `DEFAULT_ENCODING` class attribute to `'utf-8'`, which is the most popular type of encoding for the web (and not only the web). We set this attribute on the class to avoid hardcoding a string in more than one place.

Then, we proceed to set up all the model's fields. As you can see, Django allows you to specify very specific fields, such as `EmailField` and `URLField`. The reason why it's better to use these specific fields instead of a plain and simple `CharField` is we'll get e-mail and URL validation for free when we create a form for this model, which is brilliant.

All the options are quite standard, and we saw them in *Chapter 10*, *Web Development Done Right* but I want to point out a few things anyway. Firstly, `title` needs to be unique so that each `Record` object has a unique title and we don't want to risk having doubles. Each database treats strings a little bit differently, according to how it is set up, which engine it runs, and so on, so I haven't made the `title` field the primary key for this model, which would have been the natural thing to do. I prefer to avoid the pain of having to deal with weird string errors and I am happy with letting Django add a primary key to the model automatically.

Another option you should understand is the `null=True, blank=True` couple. The former allows the field to be `NULL`, which makes it non-mandatory, while the second allows it to be *blank* (that is to say, an empty string). Their use is quite peculiar in Django, so I suggest you to take a look at the official documentation to understand exactly how to use them.

Finally, the dates: `created` needs to have `auto_add_now=True`, which will set the current moment in time on the object when it's created. On the other hand, `last_modified` needs to be updated every time we save the model, hence we set `auto_now=True`.

After the field definitions, there are a few methods for encrypting and decrypting the password. It is always a very bad idea to save passwords as they are in a database, therefore you should always encrypt them before saving them.

Normally, when saving a password, you encrypt it using a **one way encryption** algorithm (also known as a **one way hash function**). This means that, once you have created the hash, there is no way for you to revert it back to the original password.

This kind of encryption is normally used for authentication: the user puts their username and password in a form and, on submission, the code fetches the hash from the user record in the database and compares it with the hash of the password the user has just put in the form. If the two hashes match, it means that they were produced by the same password, therefore authentication is granted.

In this case though, we need to be able to recover the passwords, otherwise this whole application wouldn't be very useful. Therefore, we will use a so-called **symmetric encryption** algorithm to encrypt them. The way this works is very simple: the password (called **plaintext**) is passed to an *encrypt* function, along with a *secret key*. The algorithm produces an encrypted string (called **cyphertext**) out of them, which is what you store in the database. When you want to recover the password, you will need the cyphertext and the secret key. You feed them to a *decrypt* function, and you get back your original password. This is exactly what we need.

In order to perform symmetric encryption, we need the `cryptography` package, which is why I instructed you to install it.

All the methods in the `Record` class are very simple. `encrypt_password` and `decrypt_password` are shortcuts to `encrypt` and `decrypt` the `password` field and reassign the result to itself.

The `encrypt` and `decrypt` methods are dispatchers for the `cypher` method, and `_to_str` and `_to_bytes` are just a couple of helpers. The `cryptography` library works with *bytes* objects, so we need those helpers to go back and forth between bytes and strings, using a common encoding.

The only interesting logic is in the `cypher` method. I could have coded it directly in the `encrypt` and `decrypt` ones, but that would have resulted in a bit of redundancy, and I wouldn't have had the chance to show you a different way of accessing an object's attribute, so let's analyze the body of `cypher`.

We start by creating an instance of the `Fernet` class, which provides us with the symmetric encryption functionality we need. We set the instance up by passing the secret key in the settings (`ENCRYPTION_KEY`). After creating `fernet`, we need to use it. We can use it to either encrypt or decrypt, according to what value is given to the `cypher_func` parameter. We use `getattr` to get an attribute from an object given the object itself and the name of the attribute. This technique allows us to fetch any attribute from an object dynamically.

The result of `getattr(fernet, cypher_func)`, with `cyper_func` being `'encrypt'`, for example, is the same as `fernet.encrypt`. The `getattr` function returns a method, which we then call with the bytes representation of the text argument. We then return the result, in string format.

Here's what this function is equivalent to when it's called by the encrypt dispatcher:

```
def cypher_encrypt(self, text):
        fernet = Fernet(settings.ENCRYPTION_KEY)
        result = fernet.encrypt(
            self._to_bytes(text))
        return self._to_str(result)
```

When you take the time to understand it properly, you'll see it's not as difficult as it sounds.

So, we have our model, hence it's time to migrate (I hope you remember that this will create the tables in the database for your application):

```
$ python manage.py makemigrations
$ python manage.py migrate
```

Now you should have a nice database with all the tables you need to run the interface application. Go ahead and create a superuser (`$ python manage.py createsuperuser`).

By the way, if you want to generate your own encryption key, it is as easy as this:

```
>>> from cryptography.fernet import Fernet
>>> Fernet.generate_key()
```

A simple form

We need a form for the `Record` model, so we'll use the `ModelForm` technique we saw in *Chapter 10, Web Development Done Right*.

`records/forms.py`

```
from django.forms import ModelForm, Textarea
from .models import Record

class RecordForm(ModelForm):
    class Meta:
        model = Record
        fields = ['title', 'username', 'email', 'url',
                  'password', 'notes']
        widgets = {'notes': Textarea(
            attrs={'cols': 40, 'rows': 4})}
```

We create a `RecordForm` class that inherits from `ModelForm`, so that the form is created automatically thanks to the introspection capabilities of Django. We only specify which model to use, which fields to display (we exclude the dates, which are handled automatically) and we provide minimal styling for the dimensions of the notes field, which will be displayed using a `Textarea` (which is a multiline text field in HTML).

The view layer

There are a total of five pages in the interface application: home, record list, record creation, record update, and record delete confirmation. Hence, there are five views that we have to write. As you'll see in a moment, Django helps us a lot by giving us views we can reuse with minimum customization. All the code that follows belongs to the `records/views.py` file.

Imports and home view

Just to break the ice, here are the imports and the view for the home page:

```
from django.contrib import messages
from django.contrib.messages.views import SuccessMessageMixin
from django.core.urlresolvers import reverse_lazy
from django.views.generic import TemplateView
from django.views.generic.edit import (
    CreateView, UpdateView, DeleteView)
from .forms import RecordForm
```

```
from .models import Record

class HomeView(TemplateView):
    template_name = 'records/home.html'
```

We import a few tools from Django. There are a couple of messaging-related objects, a URL lazy reverser, and four different types of view. We also import our `Record` model and `RecordForm`. As you can see, the `HomeView` class consists of only two lines since we only need to specify which template we want to use, the rest just reuses the code from `TemplateView`, as it is. It's so easy, it almost feels like cheating.

Listing all records

After the home view, we can write a view to list all the `Record` instances that we have in the database.

```
class RecordListView(TemplateView):
    template_name = 'records/list.html'

    def get(self, request, *args, **kwargs):
        context = self.get_context_data(**kwargs)
        records = Record.objects.all().order_by('title')  #1
        for record in records:
            record.plaintext = record.decrypt(record.password) #2
        context['records'] = records
        return self.render_to_response(context)
```

All we need to do is sub-class `TemplateView` again, and override the `get` method. We need to do a couple of things: we fetch all the records from the database and sort them by `title` (`#1`) and then parse all the records in order to add the attribute `plaintext` (`#2`) onto each of them, to show the original password on the page. Another way of doing this would be to add a read-only property to the `Record` model, to do the decryption on the fly. I'll leave it to you, as a fun exercise, to amend the code to do it.

After recovering and augmenting the records, we put them in the `context` dict and finish as usual by invoking `render_to_response`.

Creating records

Here's the code for the creation view:

```
class EncryptionMixin:
    def form_valid(self, form):
        self.encrypt_password(form)
        return super(EncryptionMixin, self).form_valid(form)
```

```
        def encrypt_password(self, form):
            self.object = form.save(commit=False)
            self.object.encrypt_password()
            self.object.save()

class RecordCreateView(
            EncryptionMixin, SuccessMessageMixin, CreateView):
    template_name = 'records/record_add_edit.html'
    form_class = RecordForm
    success_url = reverse_lazy('records:add')
    success_message = 'Record was created successfully'
```

A part of its logic has been factored out in order to be reused later on in the update view. Let's start with `EncryptionMixin`. All it does is override the `form_valid` method so that, prior to saving a new `Record` instance to the database, we make sure we call `encrypt_password` on the object that results from saving the form. In other words, when the user submits the form to create a new `Record`, if the form validates successfully, then the `form_valid` method is invoked. Within this method what usually happens is that an object is created out of the `ModelForm` instance, like this:

```
self.object = form.save()
```

We need to interfere with this behavior because running this code as it is would save the record with the original password, which isn't encrypted. So we change this to call `save` on the `form` passing `commit=False`, which creates the `Record` instance out of the `form`, but doesn't attempt to save it in the database. Immediately afterwards, we encrypt the password on that instance and then we can finally call save on it, actually committing it to the database.

Since we need this behavior both for creating and updating records, I have factored it out in a mixin.

Perhaps, a better solution for this password encryption logic is to create a custom `Field` (inheriting from `CharField` is the easiest way to do it) and add the necessary logic to it, so that when we handle `Record` instances from and to the database, the encryption and decryption logic is performed automatically for us. Though more elegant, this solution needs me to digress and explain a lot more about Django internals, which is too much for the extent of this example. As usual, you can try to do it yourself, if you feel like a challenge.

After creating the `EncryptionMixin` class, we can use it in the `RecordCreateView` class. We also inherit from two other classes: `SuccessMessageMixin` and `CreateView`. The message mixin provides us with the logic to quickly set up a message when creation is successful, and the `CreateView` gives us the necessary logic to create an object from a form.

You can see that all we have to code is some customization: the template name, the form class, and the success message and URL. Everything else is gracefully handled for us by Django.

Updating records

The code to update a `Record` instance is only a tiny bit more complicated. We just need to add some logic to decrypt the password before we populate the form with the record data.

```
class RecordUpdateView(
        EncryptionMixin, SuccessMessageMixin, UpdateView):
    template_name = 'records/record_add_edit.html'
    form_class = RecordForm
    model = Record
    success_message = 'Record was updated successfully'

    def get_context_data(self, **kwargs):
        kwargs['update'] = True
        return super(
            RecordUpdateView, self).get_context_data(**kwargs)

    def form_valid(self, form):
        self.success_url = reverse_lazy(
            'records:edit',
            kwargs={'pk': self.object.pk}
        )
        return super(RecordUpdateView, self).form_valid(form)

    def get_form_kwargs(self):
        kwargs = super(RecordUpdateView, self).get_form_kwargs()
        kwargs['instance'].decrypt_password()
        return kwargs
```

In this view, we still inherit from both `EncryptionMixin` and `SuccessMessageMixin`, but the view class we use is `UpdateView`.

The first four lines are customization as before, we set the template name, the form class, the `Record` model, and the success message. We cannot set the `success_url` as a class attribute because we want to redirect a successful edit to the same edit page for that record and, in order to do this, we need the ID of the instance we're editing. No worries, we'll do it another way.

First, we override `get_context_data` in order to set `'update'` to `True` in the `kwargs` argument, which means that a key `'update'` will end up in the `context` dict that is passed to the template for rendering the page. We do this because we want to use the same template for creating and updating a record, therefore we will use this variable in the context to be able to understand in which situation we are. There are other ways to do this but this one is quick and easy and I like it because it's explicit.

After overriding `get_context_data`, we need to take care of the URL redirection. We do this in the `form_valid` method since we know that, if we get there, it means the `Record` instance has been successfully updated. We reverse the `'records:edit'` view, which is exactly the view we're working on, passing the primary key of the object in question. We take that information from `self.object.pk`.

One of the reasons it's helpful to have the object saved on the view instance is that we can use it when needed without having to alter the signature of the many methods in the view in order to pass the object around. This design is very helpful and allows us to achieve a lot with very few lines of code.

The last thing we need to do is to decrypt the password on the instance before populating the form for the user. It's simple enough to do it in the `get_form_kwargs` method, where you can access the `Record` instance in the `kwargs` dict, and call `decrypt_password` on it.

This is all we need to do to update a record. If you think about it, the amount of code we had to write is really very little, thanks to Django class-based views.

A good way of understanding which is the best method to override, is to take a look at the Django official documentation or, even better in this case, check out the source code and look at the class-based views section. You'll be able to appreciate how much work has been done there by Django developers so that you only have to touch the smallest amounts of code to customize your views.

Deleting records

Of the three actions, deleting a record is definitely the easiest one. All we need is the following code:

```
class RecordDeleteView(SuccessMessageMixin, DeleteView):
    model = Record
    success_url = reverse_lazy('records:list')

    def delete(self, request, *args, **kwargs):
        messages.success(
            request, 'Record was deleted successfully')
        return super(RecordDeleteView, self).delete(
            request, *args, **kwargs)
```

We only need to inherit from `SuccessMessageMixin` and `DeleteView`, which gives us all we need. We set up the model and the success URL as class attributes, and then we override the `delete` method only to add a nice message that will be displayed in the list view (which is where we redirect to after deletion).

We don't need to specify the template name, since we'll use a name that Django infers by default: `record_confirm_delete.html`.

With this final view, we're all set to have a nice interface that we can use to handle `Record` instances.

Setting up the URLs

Before we move on to the template layer, let's set up the URLs. This time, I want to show you the inclusion technique I talked about in *Chapter 10, Web Development Done Right*.

`pwdweb/urls.py`

```
from django.conf.urls import include, url
from django.contrib import admin
from records import urls as records_url
from records.views import HomeView

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^records/', include(records_url, namespace='records')),
    url(r'^$', HomeView.as_view(), name='home'),
]
```

These are the URLs for the main project. We have the usual admin, a home page, and then for the records section, we include another `urls.py` file, which we define in the `records` application. This technique allows for apps to be reusable and self-contained. Note that, when including another `urls.py` file, you can pass namespace information, which you can then use in functions such as `reverse`, or the `url` template tag. For example, we've seen that the path to the `RecordUpdateView` was `'records:edit'`. The first part of that string is the namespace, and the second is the name that we have given to the view, as you can see in the following code:

`records/urls.py`

```
from django.conf.urls import include, url
from django.contrib import admin
from .views import (RecordCreateView, RecordUpdateView,
                    RecordDeleteView, RecordListView)

urlpatterns = [
    url(r'^add/$', RecordCreateView.as_view(), name='add'),
    url(r'^edit/(?P<pk>[0-9]+)/$', RecordUpdateView.as_view(),
        name='edit'),
    url(r'^delete/(?P<pk>[0-9]+)/$', RecordDeleteView.as_view(),
        name='delete'),
    url(r'^$', RecordListView.as_view(), name='list'),
]
```

We define four different `url` instances. There is one for adding a record, which doesn't need primary key information since the object doesn't exist yet. Then we have two `url` instances for updating and deleting a record, and for those we need to also specify primary key information to be passed to the view. Since `Record` instances have integer IDs, we can safely pass them on the URL, following good URL design practice. Finally, we define one `url` instance for the list of records.

All `url` instances have `name` information which is used in views and templates.

The template layer

Let's start with the template we'll use as the basis for the rest:

`records/templates/records/base.html`

```
{% load static from staticfiles %}
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
```

```
    <meta name="viewport"
          content="width=device-width, initial-scale=1.0">
    <link href="{% static "records/css/main.css" %}"
          rel="stylesheet">
    <title>{% block title %}Title{% endblock title %}</title>
  </head>

  <body>
    <div id="page-content">
      {% block page-content %}{% endblock page-content %}
    </div>
    <div id="footer">{% block footer %}{% endblock footer %}</div>
    {% block scripts %}
      <script
        src="{% static "records/js/jquery-2.1.4.min.js" %}">
      </script>
    {% endblock scripts %}
  </body>
</html>
```

It's very similar to the one I used in *Chapter 10*, *Web Development Done Right* although it is a bit more compressed and with one major difference. We will import jQuery in every page.

jQuery is the most popular JavaScript library out there. It allows you to write code that works on all the main browsers and it gives you many extra tools such as the ability to perform asynchronous calls (**AJAX**) from the browser itself. We'll use this library to perform the calls to the API, both to generate and validate our passwords. You can download it at `https://jquery.com/`, and put it in the `pwdweb/records/static/records/js/` folder (you may have to amend the import in the template).

I highlighted the only interesting part of this template for you. Note that we load the JavaScript library at the end. This is common practice, as JavaScript is used to manipulate the page, so loading libraries at the end helps in avoiding situations such as JavaScript code failing because the element needed hadn't been rendered on the page yet.

Home and footer templates

The home template is very simple:

`records/templates/records/home.html`

```
{% extends "records/base.html" %}
{% block title %}Welcome to the Records website.{% endblock %}

{% block page-content %}
  <h1>Welcome {{ user.first_name }}!</h1>
  <div class="home-option">To create a record click
    <a href="{% url "records:add" %}">here.</a>
  </div>
  <div class="home-option">To see all records click
    <a href="{% url "records:list" %}">here.</a>
  </div>
{% endblock page-content %}
```

There is nothing new here when compared to the `home.html` template we saw in *Chapter 10, Web Development Done Right*. The footer template is actually exactly the same:

`records/templates/records/footer.html`

```
<div class="footer">
  Go back <a href="{% url "home" %}">home</a>.
</div>
```

Listing all records

This template to list all records is fairly simple:

`records/templates/records/list.html`

```
{% extends "records/base.html" %}
{% load record_extras %}
{% block title %}Records{% endblock title %}

{% block page-content %}
  <h1>Records</h1><span name="top"></span>
  {% include "records/messages.html" %}

  {% for record in records %}
  <div class="record {% cycle 'row-light-blue' 'row-white' %}"
       id="record-{{ record.pk }}">
    <div class="record-left">
```

```
        <div class="record-list">
          <span class="record-span">Title</span>{{ record.title }}
        </div>
        <div class="record-list">
          <span class="record-span">Username</span>
          {{ record.username }}
        </div>
        <div class="record-list">
          <span class="record-span">Email</span>{{ record.email }}
        </div>
        <div class="record-list">
          <span class="record-span">URL</span>
            <a href="{{ record.url }}" target="_blank">
              {{ record.url }}</a>
        </div>
        <div class="record-list">
          <span class="record-span">Password</span>
          {% hide_password record.plaintext %}
        </div>
      </div>
      <div class="record-right">
        <div class="record-list">
          <span class="record-span">Notes</span>
          <textarea rows="3" cols="40" class="record-notes"
                    readonly>{{ record.notes }}</textarea>
        </div>
        <div class="record-list">
          <span class="record-span">Last modified</span>
          {{ record.last_modified }}
        </div>
        <div class="record-list">
          <span class="record-span">Created</span>
          {{ record.created }}
        </div>

      </div>
      <div class="record-list-actions">
        <a href="{% url "records:edit" pk=record.pk %}">» edit</a>
        <a href="{% url "records:delete" pk=record.pk %}">» delete
        </a>
      </div>
    </div>
    {% endfor %}
  {% endblock page-content %}
```

```
{% block footer %}
  <p><a href="#top">Go back to top</a></p>
  {% include "records/footer.html" %}
{% endblock footer %}
```

For this template as well, I have highlighted the parts I'd like you to focus on. Firstly, I load a custom tags module, `record_extras`, which we'll need later. I have also added an anchor at the top, so that we'll be able to put a link to it at the bottom of the page, to avoid having to scroll all the way up.

Then, I included a template to provide me with the HTML code to display Django messages. It's a very simple template which I'll show you shortly.

Then, we define a list of `div` elements. Each `Record` instance has a container `div`, in which there are two other main `div` elements: `record-left` and `record-right`. In order to display them side by side, I have set this class in the `main.css` file:

```
.record-left { float: left; width: 300px;}
```

The outermost `div` container (the one with class `record`), has an `id` attribute, which I have used as an anchor. This allows us to click on **cancel** on the record delete page, so that if we change our minds and don't want to delete the record, we can get back to the list page, and at the right position.

Each attribute of the record is then displayed in `div` elements whose class is `record-list`. Most of these classes are just there to allow me to set a bit of padding and dimensions on the HTML elements.

The next interesting bit is the `hide_password` tag, which takes the plaintext, which is the unencrypted password. The purpose of this custom tag is to display a sequence of `'*'` characters, as long as the original password, so that if someone is passing by while you're on the page, they won't see your passwords. However, hovering on that sequence of `'*'` characters will show you the original password in the tooltip. Here's the code for the `hide_password` tag:

`records/templatetags/record_extras.py`

```
from django import template
from django.utils.html import escape

register = template.Library()

@register.simple_tag
def hide_password(password):
    return '<span title="{0}">{1}</span>'.format(
        escape(password), '*' * len(password))
```

There is nothing fancy here. We just register this function as a simple tag and then we can use it wherever we want. It takes a `password` and puts it as a `tooltip` of a `span` element, whose main content is a sequence of `'*'` characters. Just note one thing: we need to escape the password, so that we're sure it won't break our HTML (think of what might happen if the password contained a double-quote `` `"` ``, for example).

As far as the `list.html` template is concerned, the next interesting bit is that we set the `readonly` attribute to the `textarea` element, so as not to give the impression to the user that they can modify notes on the fly.

Then, we set a couple of links for each `Record` instance, right at the bottom of the container `div`. There is one for the edit page, and another for the delete page. Note that we need to pass the `url` tag not only the `namespace:name` string, but also the primary key information, as required by the URL setup we made in the `urls.py` module for those views.

Finally, we import the footer and set the link to the anchor on top of the page.

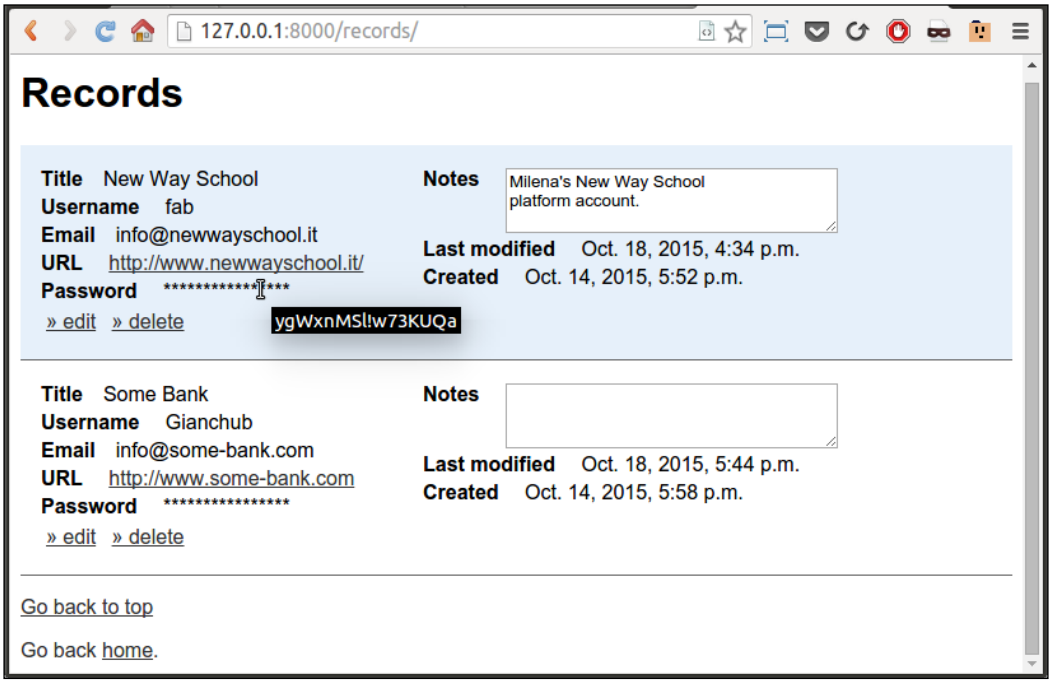
Now, as promised, here is the code for the messages:

`records/templates/records/messages.html`

```
{% if messages %}
  {% for message in messages %}
    <p class="{{ message.tags }}">{{ message }}</p>
  {% endfor %}
{% endif %}
```

This code takes care of displaying messages only when there is at least one to display. We give the `p` tag `class` information to display success messages in green and error messages in red.

If you grab the `main.css` file from the source code for the book, you will now be able to visualize the list page (yours will be blank, you still need to insert data into it), and it should look something like this:



As you can see, I have two records in the database at the moment. I'm hovering on the password of the first one, which is my platform account at my sister's school, and the password is displayed in the tooltip. The division in two `div` elements, *left* and *right*, helps in making rows smaller so that the overall result is more pleasing to the eye. The important information is on the left and the ancillary information is on the right. The row color alternates between a very light shade of blue and white.

Each row has an **edit** and **delete** link, at its bottom left. We'll show the pages for those two links right after we see the code for the templates that create them.

The CSS code that holds all the information for this interface is the following:

`records/static/records/css/main.css`

```
html, body, * {
  font-family: 'Trebuchet MS', Helvetica, sans-serif; }
a { color: #333; }
.record {
  clear: both; padding: 1em; border-bottom: 1px solid #666;}
```

```
.record-left { float: left; width: 300px;}
.record-list { padding: 2px 0; }
.fieldWrapper { padding: 5px; }
.footer { margin-top: 1em; color: #333; }
.home-option { padding: .6em 0; }
.record-span { font-weight: bold; padding-right: 1em; }
.record-notes { vertical-align: top; }
.record-list-actions { padding: 4px 0; clear: both; }
.record-list-actions a { padding: 0 4px; }
#pwd-info { padding: 0 6px; font-size: 1.1em; font-weight: bold;}
#id_notes { vertical-align: top; }
/* Messages */
.success, .errorlist {font-size: 1.2em; font-weight: bold; }
.success {color: #25B725; }
.errorlist {color: #B12B2B; }
/* colors */
.row-light-blue { background-color: #E6F0FA; }
.row-white { background-color: #fff; }
.green { color: #060; }
.orange { color: #FF3300; }
.red { color: #900; }
```

Please remember, I'm not a CSS guru so just take this file as it is, a fairly naive way to provide styling to our interface.

Creating and editing records

Now for the interesting part. Creating and updating a record. We'll use the same template for both, so we expect some decisional logic to be there that will tell us in which of the two situations we are. As it turns out, it will not be that much code. The most exciting part of this template, however, is its associated JavaScript file which we'll examine right afterwards.

records/templates/records/record_add_edit.html

```
{% extends "records/base.html" %}
{% load static from staticfiles %}
{% block title %}
  {% if update %}Update{% else %}Create{% endif %} Record
{% endblock title %}

{% block page-content %}
  <h1>{% if update %}Update a{% else %}Create a new{% endif %}
    Record
  </h1>
```

```
    {% include "records/messages.html" %}

    <form action="." method="post">{% csrf_token %}
      {{ form.non_field_errors }}

      <div class="fieldWrapper">{{ form.title.errors }}
        {{ form.title.label_tag }} {{ form.title }}</div>

      <div class="fieldWrapper">{{ form.username.errors }}
        {{ form.username.label_tag }} {{ form.username }}</div>

      <div class="fieldWrapper">{{ form.email.errors }}
        {{ form.email.label_tag }} {{ form.email }}</div>

      <div class="fieldWrapper">{{ form.url.errors }}
        {{ form.url.label_tag }} {{ form.url }}</div>

      <div class="fieldWrapper">{{ form.password.errors }}
        {{ form.password.label_tag }} {{ form.password }}
        <span id="pwd-info"></span></div>

      <button type="button" id="validate-btn">
        Validate Password</button>
      <button type="button" id="generate-btn">
        Generate Password</button>

      <div class="fieldWrapper">{{ form.notes.errors }}
        {{ form.notes.label_tag }} {{ form.notes }}</div>

      <input type="submit"
        value="{% if update %}Update{% else %}Insert{% endif %}">
    </form>
  {% endblock page-content %}

  {% block footer %}
    <br>{% include "records/footer.html" %}<br>
    Go to <a href="{% url "records:list" %}">the records list</a>.
  {% endblock footer %}

  {% block scripts %}
    {{ block.super }}
    <script src="{% static "records/js/api.js" %}"></script>
  {% endblock scripts %}
```

As usual, I have highlighted the important parts, so let's go through this code together.

You can see the first bit of decision logic in the `title` block. Similar decision logic is also displayed later on, in the header of the page (the `h1` HTML tag), and in the `submit` button at the end of the form.

Apart from this logic, what I'd like you to focus on is the form and what's inside it. We set the action attribute to a dot, which means *this page*, so that we don't need to customize it according to which view is serving the page. Also, we immediately take care of the *cross-site request forgery* token, as explained in *Chapter 10*, *Web Development Done Right*.

Note that, this time, we cannot leave the whole form rendering up to Django since we want to add in a couple of extra things, so we go down one level of granularity and ask Django to render each individual field for us, along with any errors, along with its label. This way we still save a lot of effort, and at the same time, we can also customize the form as we like. In situations like this, it's not uncommon to write a small template to render a field, in order to avoid repeating those three lines for each field. In this case though, the form is so small I decided to avoid raising the complexity level up any further.

The `span` element, `pwd-info`, contains the information about the password that we get from the API. The two buttons after that, `validate-btn` and `generate-btn`, are hooked up with the AJAX calls to the API.

At the end of the template, in the `scripts` block, we need to load the `api.js` JavaScript file which contains the code to work with the API. We also need to use `block.super`, which will load whatever code is in the same block in the parent template (for example, jQuery). `block.super` is basically the template equivalent of a call to `super(ClassName, self)` in Python. It's important to load jQuery before our library, since the latter is based on the former.

Talking to the API

Let's now take a look at that JavaScript. I don't expect you to understand everything. Firstly, this is a Python book and secondly, you're supposed to be a beginner (though by now, *ninja trained*), so fear not. However, as JavaScript has, by now, become essential if you're dealing with a web environment, having a working knowledge of it is extremely important even for a Python developer, so try and get the most out of what I'm about to show you. We'll see the password generation first:

records/static/records/js/api.js

```
var baseURL = 'http://127.0.0.1:5555/password';

var getRandomPassword = function() {
  var apiURL = '{url}/generate'.replace('{url}', baseURL);
  $.ajax({
    type: 'GET',
    url: apiURL,
    success: function(data, status, request) {
      $('#id_password').val(data[1]);
    },
    error: function() { alert('Unexpected error'); }
  });
}

$(function() {
  $('#generate-btn').click(getRandomPassword);
});
```

Firstly, we set a variable for the base API URL: `baseURL`. Then, we define the `getRandomPassword` function, which is very simple. At the beginning, it defines the `apiURL` extending `baseURL` with a replacement technique. Even if the syntax is different from that of Python, you shouldn't have any issues understanding this line.

After defining the `apiURL`, the interesting bit comes up. We call `$.ajax`, which is the jQuery function that performs the AJAX calls. That `$` is a shortcut for jQuery. As you can see in the body of the call, it's a `GET` request to `apiURL`. If it succeeds (`success`: ...), an anonymous function is run, which sets the value of the `id_password` text field to the second element of the returned data. We'll see the structure of the data when we examine the API code, so don't worry about that now. If an error occurs, we simply alert the user that there was an unexpected error.

The reason why the password field in the HTML has `id_password` as the ID is due to the way Django renders forms. You can customize this behavior using a custom prefix, for example. In this case, I'm happy with the Django defaults.

After the function definition, we run a couple of lines of code to bind the `click` event on the `generate-btn` button to the `getRandomPassword` function. This means that, after this code has been run by the browser engine, every time we click the `generate-btn` button, the `getRandomPassword` function is called.

That wasn't so scary, was it? So let's see what we need for the validation part.

Now there is a value in the **password** field and we want to validate it. We need to call the API and inspect its response. Since passwords can have weird characters, I don't want to pass them on the URL, therefore I will use a `POST` request, which allows me to put the password in its body. To do this, I need the following code:

```
var validatePassword = function() {
  var apiURL = '{url}/validate'.replace('{url}', baseURL);
  $.ajax({
    type: 'POST',
    url: apiURL,
    data: JSON.stringify({'password': $('#id_password').val()}),
    contentType: "text/plain",  // Avoid CORS preflight
    success: function(data, status, request) {
      var valid = data['valid'], infoClass, grade;
      var msg = (valid?'Valid':'Invalid') + ' password.';
      if (valid) {
        var score = data['score']['total'];
        grade = (score<10?'Poor':(score<18?'Medium':'Strong'));
        infoClass = (score<10?'red':(score<18?'orange':'green'));
        msg += ' (Score: {score}, {grade})'
          .replace('{score}', score).replace('{grade}', grade);
      }
      $('#pwd-info').html(msg);
      $('#pwd-info').removeClass().addClass(infoClass);
    },
    error: function(data) { alert('Unexpected error'); }
  });
}

$(function() {
  $('#validate-btn').click(validatePassword);
});
```

The concept is the same as before, only this time it's for the `validate-btn` button. The body of the AJAX call is similar. We use a `POST` instead of a `GET` request, and we define the data as a JSON object, which is the equivalent of using `json.dumps({'password': 'some_pwd'})` in Python.

The `contentType` line is a quick hack to avoid problems with the CORS preflight behavior of the browser. **Cross-origin resource sharing** (**CORS**) is a mechanism that allows restricted resources on a web page to be requested from another domain outside of the domain from which the request originated. In a nutshell, since the API is located at `127.0.0.1:5555` and the interface is running at `127.0.0.1:8000`, without this hack, the browser wouldn't allow us to perform the calls. In a production environment, you may want to check the documentation for JSONP, which is a much better (albeit more complex) solution to this issue.

The body of the anonymous function which is run if the call succeeds is apparently only a bit complicated. All we need to do is understand if the password is valid (from `data['valid']`), and assign it a grade and a CSS class based on its score. Validity and score information come from the API response.

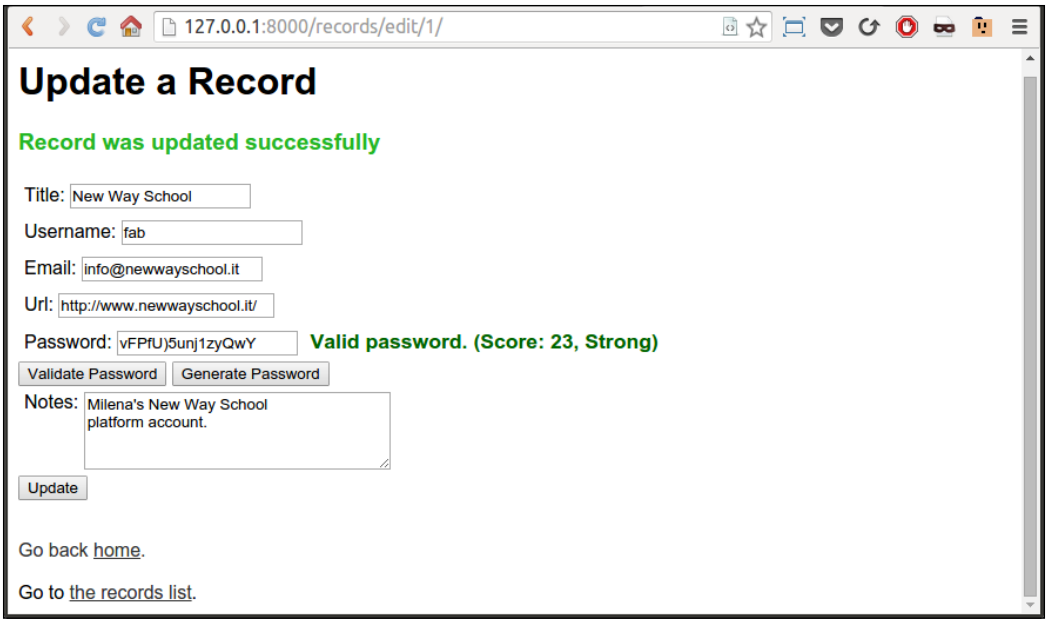
The only tricky bit in this code is the JavaScript ternary operator, so let's see a comparative example for it:

```
# Python
error = 'critical' if error_level > 50 else 'medium'
// JavaScript equivalent
error = (error_level > 50 ? 'critical' : 'medium');
```

With this example, you shouldn't have any issue reading the rest of the logic in the function. I know, I could have just used a regular `if (...)`, but JavaScript coders use the ternary operator all the time, so you should get used to it. It's good training to scratch our heads a bit harder in order to understand code.

Lastly, I'd like you to take a look at the end of that function. We set the `html` of the `pwd-info` span element to the message we assembled (`msg`), and then we style it. In one line, we remove all the CSS classes from that element (`removeClass()` with no parameters does that), and we add the `infoClass` to it. `infoClass` is either `'red'`, `'orange'`, or `'green'`. If you go back to the `main.css` file, you'll see them at the bottom.

Now that we've seen both the template code and the JavaScript to make the calls, let's see a screenshot of the page. We're going to edit the first record, the one about my sister's school.



In the picture, you can see that I updated the password by clicking on the **Generate Password** button. Then, I saved the record (so you could see the nice message on top), and, finally, I clicked on the **Validate Password** button.

The result is shown in green on the right-hand side of the **Password** field. It's strong (23 is actually the maximum score we can get) so the message is displayed in a nice shade of green.

Deleting records

To delete a record, go to the list and click on the **delete** link. You'll be redirected to a page that asks you for confirmation; you can then choose to proceed and delete the poor record, or to cancel the request and go back to the list page. The template code is the following:

`records/templates/records/record_confirm_delete.html`

```
{% extends "records/base.html" %}
{% block title %}Delete record{% endblock title %}

{% block page-content %}
  <h1>Confirm Record Deletion</h1>
  <form action="." method="post">{% csrf_token %}
    <p>Are you sure you want to delete "{{ object }}"?</p>
```

```
        <input type="submit" value="Confirm" /> 
        <a href="{% url "records:list" %}#record-{{ object.pk }}">
           » cancel</a>
      </form>
    {% endblock page-content %}
```

Since this is a template for a standard Django view, we need to use the naming conventions adopted by Django. Therefore, the record in question is called **object** in the template. The `{{ object }}` tag displays a string representation for the object, which is not exactly beautiful at the moment, since the whole line will read: **Are you sure you want to delete "Record object"?**.

This is because we haven't added a `__str__` method to our `Model` class yet, which means that Python has no idea of what to show us when we ask for a string representation of an instance. Let's change this by completing our model, adding the `__str__` method at the bottom of the class body:

`records/models.py`

```
    class Record(models.Model):
        ...

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

Restart the server and now the page will read: **Are you sure you want to delete "Some Bank"?** where *Some Bank* is the `title` of the record whose **delete** link I clicked on.

We could have just used `{{ object.title }}`, but I prefer to fix the root of the problem, not just the effect. Adding a `__str__` method is in fact something that you ought to do for all of your models.

The interesting bit in this last template is actually the link for canceling the operation. We use the `url` tag to go back to the list view (`records:list`), but we add anchor information to it so that it will eventually read something like this (this is for `pk=2`):

```
http://127.0.0.1:8000/records/#record-2
```

This will go back to the list page and scroll down to the container `div` that has ID record 2, which is nice.

This concludes the interface. Even though this section was similar to what we saw in *Chapter 10*, *Web Development Done Right*, we've been able to concentrate more on the code in this chapter. We've seen how useful Django class-based views are, and we even touched on some cool JavaScript. Run `$ python manage.py runserver` and your interface should be up and running at `http://127.0.0.1:8000`.

If you are wondering, `127.0.0.1` means the `localhost`—your computer—while `8000` is the port to which the server is bound, to listen for incoming requests.

Now it's time to spice things up a bit with the second part of this project.

Implementing the Falcon API

The structure of the Falcon project we're about to code is nowhere near as extended as the interface one. We'll code five files altogether. In your `ch12` folder, create a new one called `pwdapi`. This is its final structure:

```
$ tree -A pwdapi/
pwdapi/
├── core
│   ├── handlers.py
│   └── passwords.py
├── main.py
└── tests
    └── test_core
        ├── test_handlers.py
        └── test_passwords.py
```

The API was all coded using TDD, so we're also going to explore the tests. However, I think it's going to be easier for you to understand the tests if you first see the code, so we're going to start with that.

The main application

This is the code for the Falcon application:

`main.py`

```
import falcon
from core.handlers import (
    PasswordValidatorHandler,
    PasswordGeneratorHandler,
)

validation_handler = PasswordValidatorHandler()
generator_handler = PasswordGeneratorHandler()

app = falcon.API()
app.add_route('/password/validate/', validation_handler)
app.add_route('/password/generate/', generator_handler)
```

As in the example in *Chapter 10*, *Web Development Done Right*, we start by creating one instance for each of the handlers we need, then we create a `falcon.API` object and, by calling its `add_route` method, we set up the routing to the URLs of our API. We'll get to the definitions of the handlers in a moment. Firstly, we need a couple of helpers.

Writing the helpers

In this section, we will take a look at a couple of classes that we'll use in our handlers. It's always good to factor out some logic following the **Single Responsibility Principle**.

In OOP, the **Single Responsibility Principle** (**SRP**) states that every class should have responsibility for a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class. All of its services should be narrowly aligned with that responsibility.

The Single Responsibility Principle is the *S* in **S.O.L.I.D.**, an acronym for the first five OOP and software design principles introduced by Robert Martin.

I heartily suggest you to open a browser and read up on this subject, it is very important.

All the code in the helpers section belongs to the `core/passwords.py` module. Here's how it begins:

```
from math import ceil
from random import sample
from string import ascii_lowercase, ascii_uppercase, digits

punctuation = '!#$%&()*+-?@_|'
allchars = ''.join(
    (ascii_lowercase, ascii_uppercase, digits, punctuation))
```

We'll need to handle some randomized calculations but the most important part here is the allowed characters. We will allow letters, digits, and a set of punctuation characters. To ease writing the code, we will merge those parts into the `allchars` string.

Coding the password validator

The `PasswordValidator` class is my favorite bit of logic in the whole API. It exposes an `is_valid` and a `score` method. The latter runs all defined validators ("private" methods in the same class), and collects the scores into a single dict which is returned as a result. I'll write this class method by method so that it does not get too complicated:

```
class PasswordValidator:
    def __init__(self, password):
        self.password = password.strip()
```

It begins by setting `password` (with no leading or trailing spaces) as an instance attribute. This way we won't then have to pass it around from method to method. All the methods that will follow belong to this class.

```
    def is_valid(self):
        return (len(self.password) > 0 and
                all(char in allchars for char in self.password))
```

A password is valid when its length is greater than 0 and all of its characters belong to the `allchars` string. When you read the `is_valid` method, it's practically English (that's how amazing Python is). `all` is a built-in function that tells you if all the elements of the iterable you feed to it are `True`.

```
    def score(self):
        result = {
            'length': self._score_length(),
            'case': self._score_case(),
            'numbers': self._score_numbers(),
            'special': self._score_special(),
            'ratio': self._score_ratio(),
        }
        result['total'] = sum(result.values())
        return result
```

This is the other main method. It's very simple, it just prepares a dict with all the results from the validators. The only independent bit of logic happens at the end, when we sum the grades from each validator and assign it to a `'total'` key in the dict, just for convenience.

As you can see, we score a password by length, by letter case, by the presence of numbers, and special characters, and, finally, by the ratio between letters and numbers. Letters allow a character to be between *26 * 2 = 52* different possible choices, while digits allow only 10. Therefore, passwords whose letters to digits ratio is higher are more difficult to crack.

Let's see the length validator:

```
def _score_length(self):
    scores_list = ([0]*4) + ([1]*4) + ([3]*4) + ([5]*4)
    scores = dict(enumerate(scores_list))
    return scores.get(len(self.password), 7)
```

We assign 0 points to passwords whose length is less than four characters, 1 point for those whose length is less than 8, 3 for a length less than 12, 5 for a length less than 16, and 7 for a length of 16 or more.

In order to avoid a waterfall of `if/elif` clauses, I have adopted a functional style here. I prepared a `score_list`, which is basically `[0, 0, 0, 0, 1, 1, 1, 1, 3, ...]`. Then, by enumerating it, I got a (*length, score*) pair for each length less than 16. I put those pairs into a dict, which gives me the equivalent in dict form, so it should look like this: `{0:0, 1:0, 2:0, 3:0, 4:1, 5:1, ...}`. I then perform a `get` on this dict with the length of the password, setting a value of 7 as the default (which will be returned for lengths of 16 or more, which are not in the dict).

I have nothing against `if/elif` clauses, of course, but I wanted to take the opportunity to show you different coding styles in this final chapter, to help you get used to reading code which deviates from what you would normally expect. It's only beneficial.

```
def _score_case(self):
    lower = bool(set(ascii_lowercase) & set(self.password))
    upper = bool(set(ascii_uppercase) & set(self.password))
    return int(lower or upper) + 2 * (lower and upper)
```

The way we validate the case is again with a nice trick. `lower` is `True` when the intersection between the password and all lowercase characters is non-empty, otherwise it's `False`. `upper` behaves in the same way, only with uppercase characters.

To understand the evaluation that happens on the last line, let's use the inside-out technique once more: `lower or upper` is `True` when at least one of the two is `True`. When it's `True`, it will be converted to a `1` by the `int` class. This equates to saying, if there is at least one character, regardless of the casing, the score gets 1 point, otherwise it stays at 0.

Now for the second part: `lower and upper` is `True` when both of them are `True`, which means that we have at least one lowercase and one uppercase character. This means that, to crack the password, a brute-force algorithm would have to loop through 52 letters instead of just 26. Therefore, when that's `True`, we get an extra two points.

This validator therefore produces a result in the range (0, 1, 3), depending on what the password is.

```
def _score_numbers(self):
    return 2 if (set(self.password) & set(digits)) else 0
```

Scoring on the numbers is simpler. If we have at least one number, we get two points, otherwise we get 0. In this case, I used a ternary operator to return the result.

```
def _score_special(self):
    return 4 if (
        set(self.password) & set(punctuation)) else 0
```

The special characters validator has the same logic as the previous one but, since special characters add quite a bit of complexity when it comes to cracking a password, we have scored four points instead of just two.

The last one validates the ratio between the letters and the digits.

```
def _score_ratio(self):
    alpha_count = sum(
        1 if c.lower() in ascii_lowercase else 0
        for c in self.password)
    digits_count = sum(
        1 if c in digits else 0 for c in self.password)
    if digits_count == 0:
        return 0
    return min(ceil(alpha_count / digits_count), 7)
```

I highlighted the conditional logic in the expressions in the `sum` calls. In the first case, we get a 1 for each character whose lowercase version is in `ascii_lowercase`. This means that summing all those 1's up gives us exactly the count of all the letters. Then, we do the same for the digits, only we use the digits string for reference, and we don't need to lowercase the character. When `digits_count` is 0, `alpha_count / digits_count` would cause a `ZeroDivisionError`, therefore we check on `digits_count` and when it's 0 we return 0. If we have digits, we calculate the ceiling of the *letters:digits* ratio, and return it, capped at 7.

Of course, there are many different ways to calculate a score for a password. My aim here is not to give you the finest algorithm to do that, but to show you how you could go about implementing it.

Coding the password generator

The password generator is a much simpler class than the validator. However, I have coded it so that we won't need to create an instance to use it, just to show you yet again a different coding style.

```
class PasswordGenerator:

    @classmethod
    def generate(cls, length, bestof=10):
        candidates = sorted([
            cls._generate_candidate(length)
            for k in range(max(1, bestof))
        ])
        return candidates[-1]

    @classmethod
    def _generate_candidate(cls, length):
        password = cls._generate_password(length)
        score = PasswordValidator(password).score()
        return (score['total'], password)

    @classmethod
    def _generate_password(cls, length):
        chars = allchars * (ceil(length / len(allchars)))
        return ''.join(sample(chars, length))
```

Of the three methods, only the first one is meant to be used. Let's start our analysis with the last one: `_generate_password`.

This method simply takes a length, which is the desired length for the password we want, and calls the sample function to get a population of length elements out of the `chars` string. The return value of the sample function is a list of length elements, and we need to make it a string using `join`.

Before we can call `sample`, think about this, what if the desired length exceeds the length of `allchars`? The call would result in `ValueError: Sample larger than the population`.

Because of this, we create the `chars` string in a way that it is made by concatenating the `allchars` string to itself just enough times to cover the desired length. To give you an example, let's say we need a password of 27 characters, and let's pretend `allchars` is 10 characters long. `length / len(allchars)` gives 2.7, which, when passed to the `ceil` function, becomes 3. This means that we're going to assign `chars` to a triple concatenation of the `allchars` string, hence `chars` will be *10 * 3 = 30* characters long, which is enough to cover our requirements.

Note that, in order for these methods to be called without creating an instance of this class, we need to decorate them with the `classmethod` decorator. The convention is then to call the first argument, `cls`, instead of `self`, because Python, behind the scenes, will pass the class object to the call.

The code for `_generate_candidate` is also very simple. We just generate a password and, given the length, we calculate its score, and return a tuple (*score, password*).

We do this so that in the `generate` method we can generate 10 (by default) passwords each time the method is called and return the one that has the highest score. Since our generation logic is based on a random function, it's always a good way to employ a technique like this to avoid worst case scenarios.

This concludes the code for the helpers.

Writing the handlers

As you may have noticed, the code for the helpers isn't related to Falcon at all. It is just pure Python that we can reuse when we need it. On the other hand, the code for the handlers is of course based on Falcon. The code that follows belongs to the `core/handlers.py` module so, as we did before, let's start with the first few lines:

```
import json
import falcon
from .passwords import PasswordValidator, PasswordGenerator

class HeaderMixin:
    def set_access_control_allow_origin(self, resp):
        resp.set_header('Access-Control-Allow-Origin', '*')
```

That was very simple. We import `json`, `falcon`, and our helpers, and then we set up a mixin which we'll need in both handlers. The need for this mixin is to allow the API to serve requests that come from somewhere else. This is the other side of the CORS coin to what we saw in the JavaScript code for the interface. In this case, we boldly go where no security expert would ever dare, and allow requests to come from any domain (`'*'`). We do this because this is an exercise and, in this context, it is fine, but don't do it in production, okay?

Coding the password validator handler

This handler will have to respond to a `POST` request, therefore I have coded an `on_post` method, which is the way you react to a `POST` request in Falcon.

```
class PasswordValidatorHandler(HeaderMixin):

    def on_post(self, req, resp):
        self.process_request(req, resp)
        password = req.context.get('_body', {}).get('password')
        if password is None:
            resp.status = falcon.HTTP_BAD_REQUEST
            return None

        result = self.parse_password(password)
        resp.body = json.dumps(result)

    def parse_password(self, password):
        validator = PasswordValidator(password)
        return {
            'password': password,
            'valid': validator.is_valid(),
            'score': validator.score(),
        }

    def process_request(self, req, resp):
        self.set_access_control_allow_origin(resp)

        body = req.stream.read()
        if not body:
            raise falcon.HTTPBadRequest('Empty request body',
                'A valid JSON document is required.')
        try:
            req.context['_body'] = json.loads(
                body.decode('utf-8'))
        except (ValueError, UnicodeDecodeError):
            raise falcon.HTTPError(
                falcon.HTTP_753, 'Malformed JSON',
                'JSON incorrect or not utf-8 encoded.')
```

Let's start with the `on_post` method. First of all, we call the `process_request` method, which does a sanity check on the request body. I won't go into finest detail because it's taken from the Falcon documentation, and it's a standard way of processing a request. Let's just say that, if everything goes well (the highlighted part), we get the body of the request (already decoded from JSON) in `req.context['_body']`. If things go badly for any reason, we return an appropriate error response.

Let's go back to `on_post`. We fetch the password from the request context. At this point, `process_request` has succeeded, but we still don't know if the body was in the correct format. We're expecting something like: `{'password': 'my_password'}`.

So we proceed with caution. We get the value for the `'_body'` key and, if that is not present, we return an empty dict. We get the value for `'password'` from that. We use `get` instead of direct access to avoid `KeyError` issues.

If the password is `None`, we simply return a 400 error (bad request). Otherwise, we validate it and calculate its score, and then set the result as the body of our response.

You can see how easy it is to validate and calculate the score of the password in the `parse_password` method, by using our helpers.

We return a dict with three pieces of information: `password`, `valid`, and `score`. The password information is technically redundant because whoever made the request would know the password but, in this case, I think it's a good way of providing enough information for things such as logging, so I added it.

What happens if the JSON-decoded body is not a dict? I will leave it up to you to fix the code, adding some logic to cater for that edge case.

Coding the password generator handler

The generator handler has to handle a `GET` request with one query parameter: the desired password length.

```
class PasswordGeneratorHandler(HeaderMixin):

    def on_get(self, req, resp):
        self.process_request(req, resp)
        length = req.context.get('_length', 16)
        resp.body = json.dumps(
            PasswordGenerator.generate(length))

    def process_request(self, req, resp):
        self.set_access_control_allow_origin(resp)
        length = req.get_param('length')
        if length is None:
            return
        try:
            length = int(length)
            assert length > 0
            req.context['_length'] = length
        except (ValueError, TypeError, AssertionError):
            raise falcon.HTTPBadRequest('Wrong query parameter',
                '`length` must be a positive integer.')
```

We have a similar `process_request` method. It does a sanity check on the request, even though a bit differently from the previous handler. This time, we need to make sure that if the length is provided on the query string (which means, for example, `http://our-api-url/?length=23`), it's in the correct format. This means that `length` needs to be a positive integer.

So, to validate that, we do an `int` conversion (`req.get_param('length')` returns a string), then we assert that `length` is greater than zero and, finally, we put it in `context` under the `'_length'` key.

Doing the `int` conversion of a string which is not a suitable representation for an integer raises `ValueError`, while a conversion from a type that is not a string raises `TypeError`, therefore we catch those two in the `except` clause.

We also catch `AssertionError`, which is raised by the `assert length > 0` line when `length` is not a positive integer. We can then safely guarantee that the length is as desired with one single `try/except` block.

Note that, when coding a `try/except` block, you should usually try and be as specific as possible, separating instructions that would raise different exceptions if a problem arose. This would allow you more control over the issue, and easier debugging. In this case though, since this is a simple API, it's fine to have code which only reacts to a request for which `length` is not in the right format.

The code for the `on_get` method is quite straightforward. It starts by processing the request, then the length is fetched, falling back to 16 (the default value) when it's not passed, and then a password is generated and dumped to JSON, and then set to be the body of the response.

Running the API

In order to run this application, you need to remember that we set the base URL in the interface to `http://127.0.0.1:5555`. Therefore, we need the following command to start the API:

```
$ gunicorn -b 127.0.0.1:5555 main:app
```

Running that will start the app defined in the main module, binding the server instance to port `5555` on `localhost`. For more information about Gunicorn, please refer to either *Chapter 10*, *Web Development Done Right* or directly to the project's home page (`http://gunicorn.org/`).

The code for the API is now complete so if you have both the interface and the API running, you can try them out together. See if everything works as expected.

Testing the API

In this section, let's take a look at the tests I wrote for the helpers and for the handlers. Tests for the helpers are heavily based on the `nose_parameterized` library, as my favorite testing style is interface testing, with as little patching as possible. Using `nose_parameterized` allows me to write tests that are easier to read because the test cases are very visible.

On the other hand, tests for the handlers have to follow the testing conventions for the Falcon library, so they will be a bit different. This is, of course, ideal since it allows me to show you even more.

Due to the limited amount of pages I have left, I'll show you only a part of the tests, so make sure you check them out in full in the source code.

Testing the helpers

Let's see the tests for the `PasswordGenerator` class:

`tests/test_core/test_passwords.py`

```
    class PasswordGeneratorTestCase(TestCase):

        def test__generate_password_length(self):
            for length in range(300):
                assert_equal(
                    length,
                    len(PasswordGenerator._generate_password(length))
                )

        def test__generate_password_validity(self):
            for length in range(1, 300):
                password = PasswordGenerator._generate_password(
                    length)
                assert_true(PasswordValidator(password).is_valid())

        def test__generate_candidate(self):
            score, password = (
                PasswordGenerator._generate_candidate(42))
            expected_score = PasswordValidator(password).score()
            assert_equal(expected_score['total'], score)
```

```
@patch.object(PasswordGenerator, '_generate_candidate')
def test__generate(self, _generate_candidate_mock):
    # checks `generate` returns the highest score candidate
    _generate_candidate_mock.side_effect = [
        (16, '&a69Ly+0H4jZ'),
        (17, 'UXaF4stRfdlh'),
        (21, 'aB4Ge_KdTgwR'),  # the winner
        (12, 'IRLT*XEfcglm'),
        (16, '$P92-WZ5+DnG'),
        (18, 'Xi#36jcKA_qQ'),
        (19, '?p9avQzRMIK0'),
        (17, '4@sY&bQ9*H!+'),
        (12, 'Cx-QAYXG_Ejq'),
        (18, 'C)RAV(HP7j9n'),
    ]
    assert_equal(
        (21, 'aB4Ge_KdTgwR'), PasswordGenerator.generate(12))
```

Within `test__generate_password_length` we make sure the `_generate_password` method handles the length parameter correctly. We generate a password for each length in the range [0, 300), and verify that it has the correct length.

In the `test__generate_password_validity` test, we do something similar but, this time, we make sure that whatever length we ask for, the generated password is valid. We use the `PasswordValidator` class to check for validity.

Finally, we need to test the `generate` method. The password generation is random, therefore, in order to test this function, we need to mock `_generate_candidate`, thus controlling its output. We set the `side_effect` argument on its mock to be a list of 10 candidates, from which we expect the `generate` method to choose the one with the highest score. Setting `side_effect` on a mock to a list causes that mock to return the elements of that list, one at a time, each time it's called. To avoid ambiguity, the highest score is 21, and only one candidate has scored that high. We call the method and make sure that that particular one is the candidate which is returned.

If you are wondering why I used those double underscores in the test names, it's very simple: the first one is a separator and the second one is the leading underscore that is part of the name of the method under test.

Testing the `PasswordValidator` class requires many more lines of code, so I'll show only a portion of these tests:

`pwdapi/tests/test_core/test_passwords.py`

```
from unittest import TestCase
from unittest.mock import patch
from nose_parameterized import parameterized, param
from nose.tools import (
    assert_equal, assert_dict_equal, assert_true)
from core.passwords import PasswordValidator, PasswordGenerator

class PasswordValidatorTestCase(TestCase):

    @parameterized.expand([
        (False, ''),
        (False, '  '),
        (True, 'abcdefghijklmnopqrstuvwxyz'),
        (True, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'),
        (True, '0123456789'),
        (True, '!#$%&()*+-?@_|'),
    ])
    def test_is_valid(self, valid, password):
        validator = PasswordValidator(password)
        assert_equal(valid, validator.is_valid())
```

We start by testing the `is_valid` method. We test whether or not it returns `False` when it's fed an empty string, as well as a string made up of only spaces, which makes sure we're testing whether we're calling `.strip()` when we assign the password.

Then, we use all the characters that we want to be accepted to make sure the function accepts them.

I understand the syntax behind the `parameterize.expand` decorator can be challenging at first but really, all there is to it is that each tuple consists of an independent test case which, in turn, means that the `test_is_valid` test is run individually for each tuple, and that the two tuple elements are passed to the method as arguments: `valid` and `password`.

We then test for invalid characters. We expect them all to fail so we use `param.explicit`, which runs the test for each of the characters in that weird string.

```
@parameterized.expand(
    param.explicit(char) for char in '>]{<`\\;,[^/"\'~:}=.'
)
def test_is_valid_invalid_chars(self, password):
    validator = PasswordValidator(password)
    assert_equal(False, validator.is_valid())
```

They all evaluate to `False`, so we're good.

```
@parameterized.expand([
    (0, ''),  # 0-3: score 0
    (0, 'a'),  # 0-3: score 0
    (0, 'aa'),  # 0-3: score 0
    (0, 'aaa'),  # 0-3: score 0
    (1, 'aaab'),  # 4-7: score 1
    ...
    (5, 'aaabbbbccccddd'),  # 12-15: score 5
    (5, 'aaabbbbccccdddd'),  # 12-15: score 5
])
def test__score_length(self, score, password):
    validator = PasswordValidator(password)
    assert_equal(score, validator._score_length())
```

To test the `_score_length` method, I created 16 test cases for the lengths from 0 to 15. The body of the test simply makes sure that the score is assigned appropriately.

```
def test__score_length_sixteen_plus(self):
    # all password whose length is 16+ score 7 points
    password = 'x' * 255
    for length in range(16, len(password)):
        validator = PasswordValidator(password[:length])
        assert_equal(7, validator._score_length())
```

The preceding test is for lengths from 16 to 254. We only need to make sure that any length after 15 gets 7 as a score.

I will skip over the tests for the other internal methods and jump directly to the one for the score method. In order to test it, I want to control exactly what is returned by each of the `_score_*` methods so I mock them out and in the test, I set a return value for each of them. Note that to mock methods of a class, we use a variant of `patch`: `patch.object`. When you set return values on mocks, it's never good to have repetitions because you may not be sure which method returned what, and the test wouldn't fail in the case of a swap. So, always return different values. In my case, I am using the first few prime numbers to be sure there is no possibility of confusion.

```
@patch.object(PasswordValidator, '_score_length')
@patch.object(PasswordValidator, '_score_case')
@patch.object(PasswordValidator, '_score_numbers')
@patch.object(PasswordValidator, '_score_special')
@patch.object(PasswordValidator, '_score_ratio')
def test_score(
        self,
        _score_ratio_mock,
        _score_special_mock,
        _score_numbers_mock,
        _score_case_mock,
        _score_length_mock):

    _score_ratio_mock.return_value = 2
    _score_special_mock.return_value = 3
    _score_numbers_mock.return_value = 5
    _score_case_mock.return_value = 7
    _score_length_mock.return_value = 11

    expected_result = {
        'length': 11,
        'case': 7,
        'numbers': 5,
        'special': 3,
        'ratio': 2,
        'total': 28,
    }

    validator = PasswordValidator('')
    assert_dict_equal(expected_result, validator.score())
```

I want to point out explicitly that the `_score_*` methods are mocked, so I set up my `validator` instance by passing an empty string to the class constructor. This makes it even more evident to the reader that the internals of the class have been mocked out. Then, I just check if the result is the same as what I was expecting.

This last test is the only one in this class in which I used mocks. All the other tests for the `_score_*` methods are in an interface style, which reads better and usually produces better results.

Testing the handlers

Let's briefly see one example of a test for a handler:

`pwdapi/tests/test_core/test_handlers.py`

```
import json
from unittest.mock import patch
from nose.tools import assert_dict_equal, assert_equal
import falcon
import falcon.testing as testing
from core.handlers import (
    PasswordValidatorHandler, PasswordGeneratorHandler)

class PGHTest(PasswordGeneratorHandler):
    def process_request(self, req, resp):
        self.req, self.resp = req, resp
        return super(PGHTest, self).process_request(req, resp)

class PVHTest(PasswordValidatorHandler):
    def process_request(self, req, resp):
        self.req, self.resp = req, resp
        return super(PVHTest, self).process_request(req, resp)
```

Because of the tools Falcon gives you to test your handlers, I created a child for each of the classes I wanted to test. The only thing I changed (by overriding a method) is that in the `process_request` method, which is called by both classes, before processing the request I make sure I set the `req` and `resp` arguments on the instance. The normal behavior of the `process_request` method is thus not altered in any other way. By doing this, whatever happens over the course of the test, I'll be able to check against those objects.

It's quite common to use tricks like this when testing. We never change the code to adapt for a test, it would be bad practice. We find a way of adapting our tests to suit our needs.

```
class TestPasswordValidatorHandler(testing.TestBase):

    def before(self):
        self.resource = PVHTest()
        self.api.add_route('/password/validate/', self.resource)
```

The `before` method is called by the Falcon `TestBase` logic, and it allows us to set up the resource we want to test (the handler) and a route for it (which is not necessarily the same as the one we use in production).

```
    def test_post(self):
        self.simulate_request(
            '/password/validate/',
            body=json.dumps({'password': 'abcABC0123#&'}),
            method='POST')
        resp = self.resource.resp

        assert_equal('200 OK', resp.status)
        assert_dict_equal(
            {'password': 'abcABC0123#&',
             'score': {'case': 3, 'length': 5, 'numbers': 2,
                'special': 4, 'ratio': 2, 'total': 16},
             'valid': True},
             json.loads(resp.body))
```

This is the test for the happy path. All it does is simulate a `POST` request with a JSON payload as body. Then, we inspect the response object. In particular, we inspect its status and its body. We make sure that the handler has correctly called the validator and returned its results.

We also test the generator handler:

```
class TestPasswordGeneratorHandler(testing.TestBase):

    def before(self):
        self.resource = PGHTest()
        self.api.add_route('/password/generate/', self.resource)

    @patch('core.handlers.PasswordGenerator')
    def test_get(self, PasswordGenerator):
        PasswordGenerator.generate.return_value = (7, 'abc123')
        self.simulate_request(
            '/password/generate/',
            query_string='length=7',
            method='GET')
        resp = self.resource.resp

        assert_equal('200 OK', resp.status)
        assert_equal([7, 'abc123'], json.loads(resp.body))
```

For this one as well, I will only show you the test for the happy path. We mock out the `PasswordGenerator` class because we need to control which password it will generate and, unless we mock, we won't be able to do it, as it is a random process.

Once we have correctly set up its return value, we can simulate the request again. In this case, it's a `GET` request, with a desired length of 7. We use a technique similar to the one we used for the other handler, and check the response status and body.

These are not the only tests you could write against the API, and the style could be different as well. Some people mock often, I tend to mock only when I really have to. Just try to see if you can make some sense out of them. I know they're not really easy but they'll be good training for you. Tests are extremely important so give it your best shot.

Where do you go from here?

If you liked this project and you feel like expanding it, here are a few suggestions:

- Implement the encryption in the mechanism of a custom Django field.
- Amend the template for the record list so that you can search for a particular record.
- Amend the JavaScript to use JSONP with a callback to overcome the CORS issue.
- Amend the JavaScript to fire the validation call when the password field changes.
- Write a Django command that allows you to encrypt and decrypt the database file. When you do it from the command line, incorporate that behavior into the website, possibly on the home page, so that you don't have access to the records unless you are authenticated. This is definitely a hard challenge as it requires either another database with an authentication password stored properly with a one way hash, or some serious reworking of the data structure used to hold the record model data. Even if you don't have the means to do it now, try and think about how you would solve this problem.
- Set up PostgreSQL on your machine and switch to using it instead of the SQLite file that is the default.
- Add the ability to attach a file to a record.
- Play with the application, try to find out which features you want to add or change, and then do it.

Summary

In this chapter, we've worked on a final project that involves an interface and an API. We have used two different frameworks to accomplish our goal: Django and Falcon. They are very different and have allowed us to explore different concepts and techniques to craft our software and make this fun application come alive.

We have seen an example of symmetric encryption and explored code that was written in a more functional style, as opposed to a more classic control flow-oriented approach. We have reused and extended the Django class-based views, reducing to a minimum the amount of code we had to write.

When coding the API, we decoupled handling requests from password management. This way it's much easier to see which part of the code depends on the Falcon framework and which is independent from it.

Finally, we saw a few tests for the helpers and handlers of the API. We have briefly touched on a technique that I use to expand classes under test in order to be able to test against those parts of the code which would not normally be available.

My aim in this chapter was to provide you with an interesting example that could be expanded and improved in different ways. I also wanted to give you a few examples of different coding styles and techniques, which is why I chose to spread things apart and use different frameworks.

Module 2

Python 3 Object-Oriented Programming

Unleash the power of Python 3 objects

1
Object-oriented Design

In software development, design is often considered as the step done *before* programming. This isn't true; in reality, analysis, programming, and design tend to overlap, combine, and interweave. In this chapter, we will cover the following topics:

- What object-oriented means
- The difference between object-oriented design and object-oriented programming
- The basic principles of an object-oriented design
- Basic **Unified Modeling Language** (**UML**) and when it isn't evil

Introducing object-oriented

Everyone knows what an object is—a tangible thing that we can sense, feel, and manipulate. The earliest objects we interact with are typically baby toys. Wooden blocks, plastic shapes, and over-sized puzzle pieces are common first objects. Babies learn quickly that certain objects do certain things: bells ring, buttons press, and levers pull.

The definition of an object in software development is not terribly different. Software objects are not typically tangible things that you can pick up, sense, or feel, but they are models of something that can do certain things and have certain things done to them. Formally, an object is a collection of **data** and associated **behaviors**.

So, knowing what an object is, what does it mean to be object-oriented? Oriented simply means *directed toward*. So object-oriented means functionally directed towards modeling objects. This is one of the many techniques used for modeling complex systems by describing a collection of interacting objects via their data and behavior.

If you've read any hype, you've probably come across the terms object-oriented analysis, object-oriented design, object-oriented analysis and design, and object-oriented programming. These are all highly related concepts under the general object-oriented umbrella.

In fact, analysis, design, and programming are all stages of software development. Calling them object-oriented simply specifies what style of software development is being pursued.

Object-oriented analysis (**OOA**) is the process of looking at a problem, system, or task (that somebody wants to turn into an application) and identifying the objects and interactions between those objects. The analysis stage is all about *what* needs to be done.

The output of the analysis stage is a set of requirements. If we were to complete the analysis stage in one step, we would have turned a task, such as, I need a website, into a set of requirements. For example:

Website visitors need to be able to (*italic* represents actions, **bold** represents objects):

- *review* our **history**
- *apply* for **jobs**
- *browse*, *compare*, and *order* **products**

In some ways, analysis is a misnomer. The baby we discussed earlier doesn't analyze the blocks and puzzle pieces. Rather, it will explore its environment, manipulate shapes, and see where they might fit. A better turn of phrase might be object-oriented exploration. In software development, the initial stages of analysis include interviewing customers, studying their processes, and eliminating possibilities.

Object-oriented design (**OOD**) is the process of converting such requirements into an implementation specification. The designer must name the objects, define the behaviors, and formally specify which objects can activate specific behaviors on other objects. The design stage is all about *how* things should be done.

The output of the design stage is an implementation specification. If we were to complete the design stage in a single step, we would have turned the requirements defined during object-oriented analysis into a set of classes and interfaces that could be implemented in (ideally) any object-oriented programming language.

Object-oriented programming (**OOP**) is the process of converting this perfectly defined design into a working program that does exactly what the CEO originally requested.

Yeah, right! It would be lovely if the world met this ideal and we could follow these stages one by one, in perfect order, like all the old textbooks told us to. As usual, the real world is much murkier. No matter how hard we try to separate these stages, we'll always find things that need further analysis while we're designing. When we're programming, we find features that need clarification in the design.

Most twenty-first century development happens in an iterative development model. In iterative development, a small part of the task is modeled, designed, and programmed, then the program is reviewed and expanded to improve each feature and include new features in a series of short development cycles.

The rest of this book is about object-oriented programming, but in this chapter, we will cover the basic object-oriented principles in the context of design. This allows us to understand these (rather simple) concepts without having to argue with software syntax or Python interpreters.

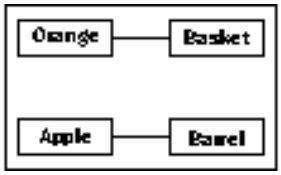
Objects and classes

So, an object is a collection of data with associated behaviors. How do we differentiate between types of objects? Apples and oranges are both objects, but it is a common adage that they cannot be compared. Apples and oranges aren't modeled very often in computer programming, but let's pretend we're doing an inventory application for a fruit farm. To facilitate the example, we can assume that apples go in barrels and oranges go in baskets.

Now, we have four kinds of objects: apples, oranges, baskets, and barrels. In object-oriented modeling, the term used for *kind of object* is **class**. So, in technical terms, we now have four classes of objects.

What's the difference between an object and a class? Classes describe objects. They are like blueprints for creating an object. You might have three oranges sitting on the table in front of you. Each orange is a distinct object, but all three have the attributes and behaviors associated with one class: the general class of oranges.

The relationship between the four classes of objects in our inventory system can be described using a **Unified Modeling Language** (invariably referred to as **UML**, because three letter acronyms never go out of style) class diagram. Here is our first class diagram:



This diagram shows that an **Orange** is somehow associated with a **Basket** and that an **Apple** is also somehow associated with a **Barrel**. Association is the most basic way for two classes to be related.

UML is very popular among managers, and occasionally disparaged by programmers. The syntax of a UML diagram is generally pretty obvious; you don't have to read a tutorial to (mostly) understand what is going on when you see one. UML is also fairly easy to draw, and quite intuitive. After all, many people, when describing classes and their relationships, will naturally draw boxes with lines between them. Having a standard based on these intuitive diagrams makes it easy for programmers to communicate with designers, managers, and each other.

However, some programmers think UML is a waste of time. Citing iterative development, they will argue that formal specifications done up in fancy UML diagrams are going to be redundant before they're implemented, and that maintaining these formal diagrams will only waste time and not benefit anyone.

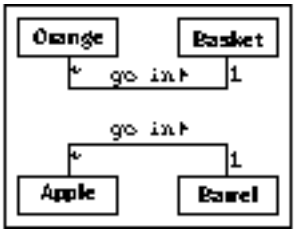
Depending on the corporate structure involved, this may or may not be true. However, every programming team consisting of more than one person will occasionally has to sit down and hash out the details of the subsystem it is currently working on. UML is extremely useful in these brainstorming sessions for quick and easy communication. Even those organizations that scoff at formal class diagrams tend to use some informal version of UML in their design meetings or team discussions.

Further, the most important person you will ever have to communicate with is yourself. We all think we can remember the design decisions we've made, but there will always be the *Why did I do that?* moments hiding in our future. If we keep the scraps of papers we did our initial diagramming on when we started a design, we'll eventually find them a useful reference.

This chapter, however, is not meant to be a tutorial in UML. There are many of these available on the Internet, as well as numerous books available on the topic. UML covers far more than class and object diagrams; it also has a syntax for use cases, deployment, state changes, and activities. We'll be dealing with some common class diagram syntax in this discussion of object-oriented design. You'll find that you can pick up the structure by example, and you'll subconsciously choose the UML-inspired syntax in your own team or personal design sessions.

Our initial diagram, while correct, does not remind us that apples go in barrels or how many barrels a single apple can go in. It only tells us that apples are somehow associated with barrels. The association between classes is often obvious and needs no further explanation, but we have the option to add further clarification as needed.

The beauty of UML is that most things are optional. We only need to specify as much information in a diagram as makes sense for the current situation. In a quick whiteboard session, we might just quickly draw lines between boxes. In a formal document, we might go into more detail. In the case of apples and barrels, we can be fairly confident that the association is, **many apples go in one barrel**, but just to make sure nobody confuses it with, **one apple spoils one barrel**, we can enhance the diagram as shown:



This diagram tells us that oranges **go in** baskets with a little arrow showing what goes in what. It also tells us the number of that object that can be used in the association on both sides of the relationship. One **Basket** can hold many (represented by a *) **Orange** objects. Any one **Orange** can go in exactly one **Basket**. This number is referred to as the multiplicity of the object. You may also hear it described as the cardinality. These are actually slightly distinct terms. Cardinality refers to the actual number of items in the set, whereas multiplicity specifies how small or how large this number could be.

I frequently forget which side of a relationship the multiplicity goes on. The multiplicity nearest to a class is the number of objects of that class that can be associated with any one object at the other end of the association. For the apple goes in barrel association, reading from left to right, many instances of the **Apple** class (that is many **Apple** objects) can go in any one **Barrel**. Reading from right to left, exactly one **Barrel** can be associated with any one **Apple**.

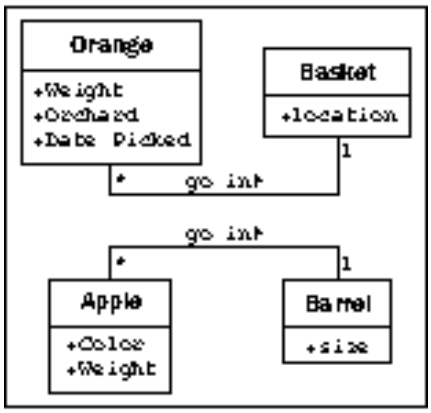
Specifying attributes and behaviors

We now have a grasp of some basic object-oriented terminology. Objects are instances of classes that can be associated with each other. An object instance is a specific object with its own set of data and behaviors; a specific orange on the table in front of us is said to be an instance of the general class of oranges. That's simple enough, but what are these data and behaviors that are associated with each object?

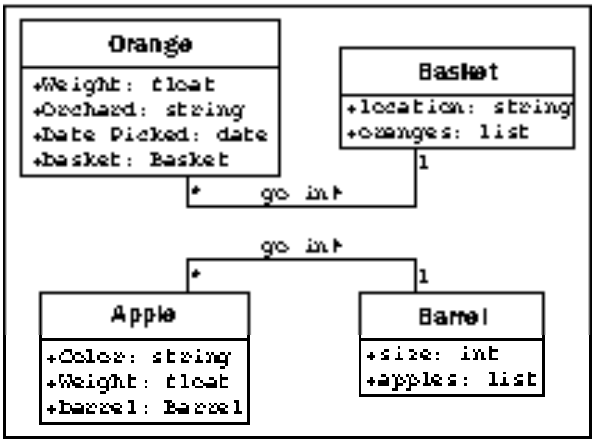
Data describes objects

Let's start with data. Data typically represents the individual characteristics of a certain object. A class can define specific sets of characteristics that are shared by all objects of that class. Any specific object can have different data values for the given characteristics. For example, our three oranges on the table (if we haven't eaten any) could each weigh a different amount. The orange class could then have a weight **attribute**. All instances of the orange class have a weight attribute, but each orange has a different value for this attribute. Attributes don't have to be unique, though; any two oranges may weigh the same amount. As a more realistic example, two objects representing different customers might have the same value for a first name attribute.

Attributes are frequently referred to as **members** or **properties**. Some authors suggest that the terms have different meanings, usually that attributes are settable, while properties are read-only. In Python, the concept of "read-only" is rather pointless, so throughout this book, we'll see the two terms used interchangeably. In addition, as we'll discuss in *Chapter 5*, *When to Use Object-oriented Programming*, the property keyword has a special meaning in Python for a particular kind of attribute.



In our fruit inventory application, the fruit farmer may want to know what orchard the orange came from, when it was picked, and how much it weighs. They might also want to keep track of where each basket is stored. Apples might have a color attribute, and barrels might come in different sizes. Some of these properties may also belong to multiple classes (we may want to know when apples are picked, too), but for this first example, let's just add a few different attributes to our class diagram:



Depending on how detailed our design needs to be, we can also specify the type for each attribute. Attribute types are often primitives that are standard to most programming languages, such as integer, floating-point number, string, byte, or Boolean. However, they can also represent data structures such as lists, trees, or graphs, or most notably, other classes. This is one area where the design stage can overlap with the programming stage. The various primitives or objects available in one programming language may be somewhat different from what is available in other languages.

Usually, we don't need to be overly concerned with data types at the design stage, as implementation-specific details are chosen during the programming stage. Generic names are normally sufficient for design. If our design calls for a list container type, the Java programmers can choose to use a `LinkedList` or an `ArrayList` when implementing it, while the Python programmers (that's us!) can choose between the `list` built-in and a `tuple`.

In our fruit-farming example so far, our attributes are all basic primitives. However, there are some implicit attributes that we can make explicit—the associations. For a given orange, we might have an attribute containing the basket that holds that orange.

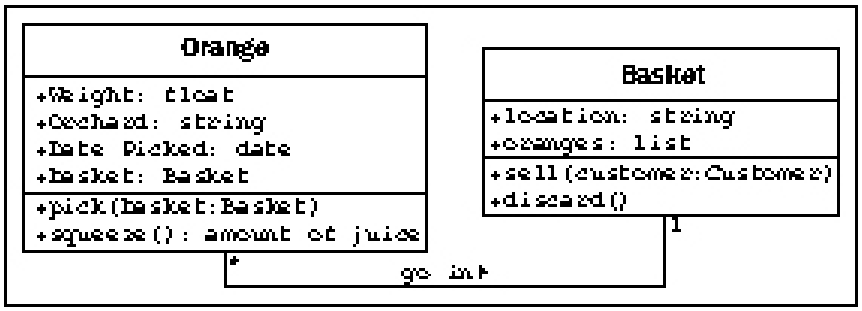
Behaviors are actions

Now, we know what data is, but what are behaviors? Behaviors are actions that can occur on an object. The behaviors that can be performed on a specific class of objects are called **methods**. At the programming level, methods are like functions in structured programming, but they magically have access to all the data associated with this object. Like functions, methods can also accept **parameters** and return **values**.

Parameters to a method are a list of objects that need to be **passed** into the method that is being called (the objects that are passed in from the calling object are usually referred to as **arguments**). These objects are used by the method to perform whatever behavior or task it is meant to do. Returned values are the results of that task.

We've stretched our "comparing apples and oranges" example into a basic (if far-fetched) inventory application. Let's stretch it a little further and see if it breaks. One action that can be associated with oranges is the **pick** action. If you think about implementation, **pick** would place the orange in a basket by updating the **basket** attribute of the orange, and by adding the orange to the **oranges** list on the **Basket**. So, **pick** needs to know what basket it is dealing with. We do this by giving the **pick** method a **basket** parameter. Since our fruit farmer also sells juice, we can add a **squeeze** method to **Orange**. When squeezed, **squeeze** might return the amount of juice retrieved, while also removing the **Orange** from the **basket** it was in.

Basket can have a **sell** action. When a basket is sold, our inventory system might update some data on as-yet unspecified objects for accounting and profit calculations. Alternatively, our basket of oranges might go bad before we can sell them, so we add a **discard** method. Let's add these methods to our diagram:



Adding models and methods to individual objects allows us to create a **system** of interacting objects. Each object in the system is a member of a certain class. These classes specify what types of data the object can hold and what methods can be invoked on it. The data in each object can be in a different state from other objects of the same class, and each object may react to method calls differently because of the differences in state.

Object-oriented analysis and design is all about figuring out what those objects are and how they should interact. The next section describes principles that can be used to make those interactions as simple and intuitive as possible.

Hiding details and creating the public interface

The key purpose of modeling an object in object-oriented design is to determine what the public **interface** of that object will be. The interface is the collection of attributes and methods that other objects can use to interact with that object. They do not need, and are often not allowed, to access the internal workings of the object. A common real-world example is the television. Our interface to the television is the remote control. Each button on the remote control represents a method that can be called on the television object. When we, as the calling object, access these methods, we do not know or care if the television is getting its signal from an antenna, a cable connection, or a satellite dish. We don't care what electronic signals are being sent to adjust the volume, or whether the sound is destined to speakers or headphones. If we open the television to access the internal workings, for example, to split the output signal to both external speakers and a set of headphones, we will void the warranty.

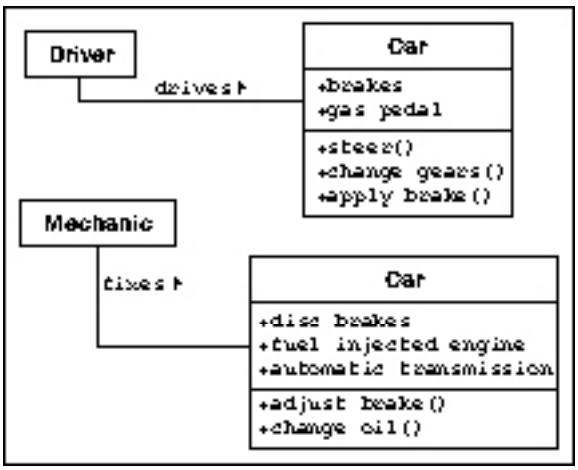
This process of hiding the implementation, or functional details, of an object is suitably called **information hiding**. It is also sometimes referred to as **encapsulation**, but encapsulation is actually a more all-encompassing term. Encapsulated data is not necessarily hidden. Encapsulation is, literally, creating a capsule and so think of creating a time capsule. If you put a bunch of information into a time capsule, lock and bury it, it is both encapsulated and the information is hidden. On the other hand, if the time capsule has not been buried and is unlocked or made of clear plastic, the items inside it are still encapsulated, but there is no information hiding.

The distinction between encapsulation and information hiding is largely irrelevant, especially at the design level. Many practical references use these terms interchangeably. As Python programmers, we don't actually have or need true information hiding, (we'll discuss the reasons for this in *Chapter 2*, *Objects in Python*) so the more encompassing definition for encapsulation is suitable.

The public interface, however, is very important. It needs to be carefully designed as it is difficult to change it in the future. Changing the interface will break any client objects that are calling it. We can change the internals all we like, for example, to make it more efficient, or to access data over the network as well as locally, and the client objects will still be able to talk to it, unmodified, using the public interface. On the other hand, if we change the interface by changing attribute names that are publicly accessed, or by altering the order or types of arguments that a method can accept, all client objects will also have to be modified. While on the topic of public interfaces, keep it simple. Always design the interface of an object based on how easy it is to use, not how hard it is to code (this advice applies to user interfaces as well).

Remember, program objects may represent real objects, but that does not make them real objects. They are models. One of the greatest gifts of modeling is the ability to ignore irrelevant details. The model car I built as a child may look like a real 1956 Thunderbird on the outside, but it doesn't run and the driveshaft doesn't turn. These details were overly complex and irrelevant before I started driving. The model is an **abstraction** of a real concept.

Abstraction is another object-oriented concept related to encapsulation and information hiding. Simply put, abstraction means dealing with the level of detail that is most appropriate to a given task. It is the process of extracting a public interface from the inner details. A driver of a car needs to interact with steering, gas pedal, and brakes. The workings of the motor, drive train, and brake subsystem don't matter to the driver. A mechanic, on the other hand, works at a different level of abstraction, tuning the engine and bleeding the breaks. Here's an example of two abstraction levels for a car:



Now, we have several new terms that refer to similar concepts. Condensing all this jargon into a couple of sentences: abstraction is the process of encapsulating information with separate public and private interfaces. The private interfaces can be subject to information hiding.

The important lesson to take from all these definitions is to make our models understandable to other objects that have to interact with them. This means paying careful attention to small details. Ensure methods and properties have sensible names. When analyzing a system, objects typically represent nouns in the original problem, while methods are normally verbs. Attributes can often be picked up as adjectives, although if the attribute refers to another object that is part of the current object, it will still likely be a noun. Name classes, attributes, and methods accordingly.

Don't try to model objects or actions that *might* be useful in the future. Model exactly those tasks that the system needs to perform, and the design will naturally gravitate towards the one that has an appropriate level of abstraction. This is not to say we should not think about possible future design modifications. Our designs should be open ended so that future requirements can be satisfied. However, when abstracting interfaces, try to model exactly what needs to be modeled and nothing more.

When designing the interface, try placing yourself in the object's shoes and imagine that the object has a strong preference for privacy. Don't let other objects have access to data about you unless you feel it is in your best interest for them to have it. Don't give them an interface to force you to perform a specific task unless you are certain you want them to be able to do that to you.

Composition

So far, we learned to design systems as a group of interacting objects, where each interaction involves viewing objects at an appropriate level of abstraction. But we don't know yet how to create these levels of abstraction. There are a variety of ways to do this; we'll discuss some advanced design patterns in *Chapter 8*, *Strings and Serialization* and *Chapter 9*, *The Iterator Pattern*. But even most design patterns rely on two basic object-oriented principles known as **composition** and **inheritance**. Composition is simpler, so let's start with it.

Composition is the act of collecting several objects together to create a new one. Composition is usually a good choice when one object is part of another object. We've already seen a first hint of composition in the mechanic example. A car is composed of an engine, transmission, starter, headlights, and windshield, among numerous other parts. The engine, in turn, is composed of pistons, a crank shaft, and valves. In this example, composition is a good way to provide levels of abstraction. The car object can provide the interface required by a driver, while also providing access to its component parts, which offers the deeper level of abstraction suitable for a mechanic. Those component parts can, of course, be further broken down if the mechanic needs more information to diagnose a problem or tune the engine.

This is a common introductory example of composition, but it's not overly useful when it comes to designing computer systems. Physical objects are easy to break into component objects. People have been doing this at least since the ancient Greeks originally postulated that atoms were the smallest units of matter (they, of course, didn't have access to particle accelerators). Computer systems are generally less complicated than physical objects, yet identifying the component objects in such systems does not happen as naturally.

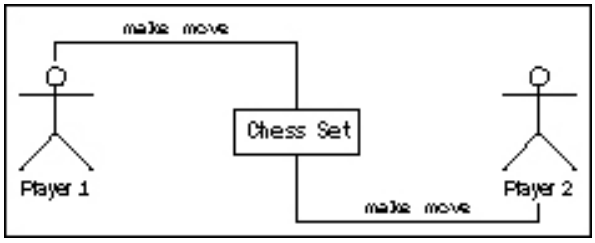
The objects in an object-oriented system occasionally represent physical objects such as people, books, or telephones. More often, however, they represent abstract ideas. People have names, books have titles, and telephones are used to make calls. Calls, titles, accounts, names, appointments, and payments are not usually considered objects in the physical world, but they are all frequently-modeled components in computer systems.

Let's try modeling a more computer-oriented example to see composition in action. We'll be looking at the design of a computerized chess game. This was a very popular pastime among academics in the 80s and 90s. People were predicting that computers would one day be able to defeat a human chess master. When this happened in 1997 (IBM's Deep Blue defeated world chess champion, Gary Kasparov), interest in the problem waned, although there are still contests between computer and human chess players. (The computers usually win.)

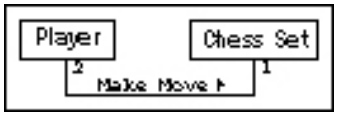
As a basic, high-level analysis, a game of chess is played between two players, using a chess set featuring a board containing sixty-four positions in an 8 X 8 grid. The board can have two sets of sixteen pieces that can be moved, in alternating turns by the two players in different ways. Each piece can take other pieces. The board will be required to draw itself on the computer screen after each turn.

I've identified some of the possible objects in the description using *italics*, and a few key methods using **bold**. This is a common first step in turning an object-oriented analysis into a design. At this point, to emphasize composition, we'll focus on the board, without worrying too much about the players or the different types of pieces.

Let's start at the highest level of abstraction possible. We have two players interacting with a chess set by taking turns making moves:



What is this? It doesn't quite look like our earlier class diagrams. That's because it isn't a class diagram! This is an **object diagram**, also called an instance diagram. It describes the system at a specific state in time, and is describing specific instances of objects, not the interaction between classes. Remember, both players are members of the same class, so the class diagram looks a little different:



The diagram shows that exactly two players can interact with one chess set. It also indicates that any one player can be playing with only one chess set at a time.

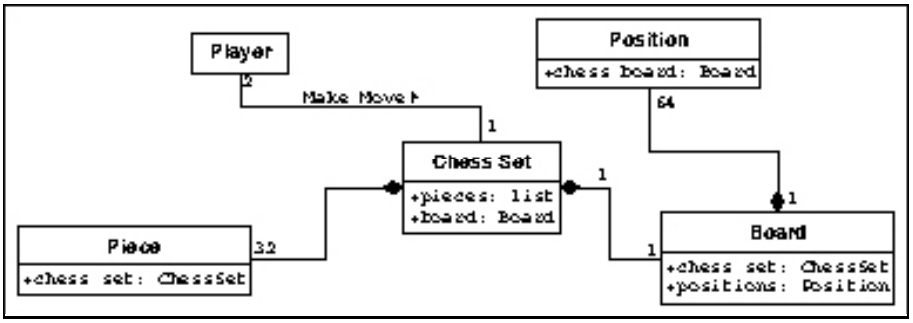
However, we're discussing composition, not UML, so let's think about what the **Chess Set** is composed of. We don't care what the player is composed of at this time. We can assume that the player has a heart and brain, among other organs, but these are irrelevant to our model. Indeed, there is nothing stopping said player from being Deep Blue itself, which has neither a heart nor a brain.

The chess set, then, is composed of a board and 32 pieces. The board further comprises 64 positions. You could argue that pieces are not part of the chess set because you could replace the pieces in a chess set with a different set of pieces. While this is unlikely or impossible in a computerized version of chess, it introduces us to **aggregation**.

Aggregation is almost exactly like composition. The difference is that aggregate objects can exist independently. It would be impossible for a position to be associated with a different chess board, so we say the board is composed of positions. But the pieces, which might exist independently of the chess set, are said to be in an aggregate relationship with that set.

Another way to differentiate between aggregation and composition is to think about the lifespan of the object. If the composite (outside) object controls when the related (inside) objects are created and destroyed, composition is most suitable. If the related object is created independently of the composite object, or can outlast that object, an aggregate relationship makes more sense. Also, keep in mind that composition is aggregation; aggregation is simply a more general form of composition. Any composite relationship is also an aggregate relationship, but not vice versa.

Let's describe our current chess set composition and add some attributes to the objects to hold the composite relationships:



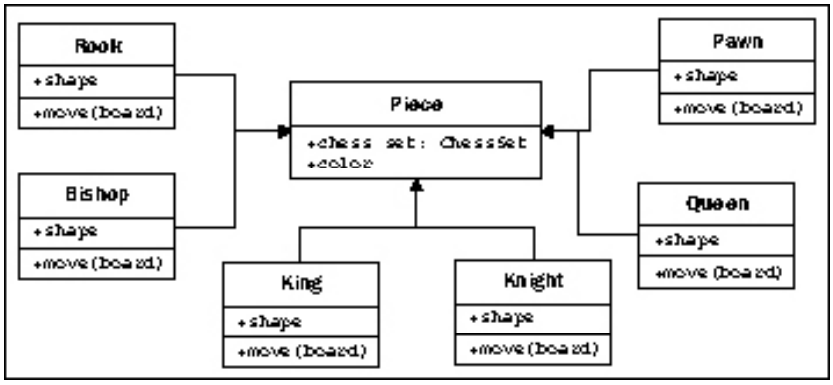
The composition relationship is represented in UML as a solid diamond. The hollow diamond represents the aggregate relationship. You'll notice that the board and pieces are stored as part of the chess set in exactly the same way a reference to them is stored as an attribute on the chess set. This shows that, once again, in practice, the distinction between aggregation and composition is often irrelevant once you get past the design stage. When implemented, they behave in much the same way. However, it can help to differentiate between the two when your team is discussing how the different objects interact. Often, you can treat them as the same thing, but when you need to distinguish between them, it's great to know the difference (this is abstraction at work).

Inheritance

We discussed three types of relationships between objects: association, composition, and aggregation. However, we have not fully specified our chess set, and these tools don't seem to give us all the power we need. We discussed the possibility that a player might be a human or it might be a piece of software featuring artificial intelligence. It doesn't seem right to say that a player is *associated* with a human, or that the artificial intelligence implementation is *part of* the player object. What we really need is the ability to say that "Deep Blue *is a* player" or that "Gary Kasparov *is a* player".

The *is a* relationship is formed by **inheritance**. Inheritance is the most famous, well-known, and over-used relationship in object-oriented programming. Inheritance is sort of like a family tree. My grandfather's last name was Phillips and my father inherited that name. I inherited it from him (along with blue eyes and a penchant for writing). In object-oriented programming, instead of inheriting features and behaviors from a person, one class can inherit attributes and methods from another class.

For example, there are 32 chess pieces in our chess set, but there are only six different types of pieces (pawns, rooks, bishops, knights, king, and queen), each of which behaves differently when it is moved. All of these classes of piece have properties, such as color and the chess set they are part of, but they also have unique shapes when drawn on the chess board, and make different moves. Let's see how the six types of pieces can inherit from a **Piece** class:



The hollow arrows indicate that the individual classes of pieces inherit from the **Piece** class. All the subtypes automatically have a **chess_set** and **color** attribute inherited from the base class. Each piece provides a different shape property (to be drawn on the screen when rendering the board), and a different **move** method to move the piece to a new position on the board at each turn.

We actually know that all subclasses of the **Piece** class need to have a **move** method; otherwise, when the board tries to move the piece, it will get confused. It is possible that we would want to create a new version of the game of chess that has one additional piece (the wizard). Our current design allows us to design this piece without giving it a **move** method. The board would then choke when it asked the piece to move itself.

We can implement this by creating a dummy move method on the **Piece** class. The subclasses can then **override** this method with a more specific implementation. The default implementation might, for example, pop up an error message that says: **That piece cannot be moved**.

Overriding methods in subtypes allows very powerful object-oriented systems to be developed. For example, if we wanted to implement a player class with artificial intelligence, we might provide a `calculate_move` method that takes a **Board** object and decides which piece to move where. A very basic class might randomly choose a piece and direction and move it accordingly. We could then override this method in a subclass with the Deep Blue implementation. The first class would be suitable for play against a raw beginner, the latter would challenge a grand master. The important thing is that other methods in the class, such as the ones that inform the board as to which move was chose need not be changed; this implementation can be shared between the two classes.

In the case of chess pieces, it doesn't really make sense to provide a default implementation of the move method. All we need to do is specify that the move method is required in any subclasses. This can be done by making **Piece** an **abstract class** with the move method declared **abstract**. Abstract methods basically say, "We demand this method exist in any non-abstract subclass, but we are declining to specify an implementation in this class."

Indeed, it is possible to make a class that does not implement any methods at all. Such a class would simply tell us what the class should do, but provides absolutely no advice on how to do it. In object-oriented parlance, such classes are called **interfaces**.

Inheritance provides abstraction

Let's explore the longest word in object-oriented argot. **Polymorphism** is the ability to treat a class differently depending on which subclass is implemented. We've already seen it in action with the pieces system we've described. If we took the design a bit further, we'd probably see that the **Board** object can accept a move from the player and call the **move** function on the piece. The board need not ever know what type of piece it is dealing with. All it has to do is call the **move** method, and the proper subclass will take care of moving it as a **Knight** or a **Pawn**.

Polymorphism is pretty cool, but it is a word that is rarely used in Python programming. Python goes an extra step past allowing a subclass of an object to be treated like a parent class. A board implemented in Python could take any object that has a **move** method, whether it is a bishop piece, a car, or a duck. When **move** is called, the **Bishop** will move diagonally on the board, the car will drive someplace, and the duck will swim or fly, depending on its mood.

This sort of polymorphism in Python is typically referred to as **duck typing**: "If it walks like a duck or swims like a duck, it's a duck". We don't care if it really *is a* duck (inheritance), only that it swims or walks. Geese and swans might easily be able to provide the duck-like behavior we are looking for. This allows future designers to create new types of birds without actually specifying an inheritance hierarchy for aquatic birds. It also allows them to create completely different drop-in behaviors that the original designers never planned for. For example, future designers might be able to make a walking, swimming penguin that works with the same interface without ever suggesting that penguins are ducks.

Multiple inheritance

When we think of inheritance in our own family tree, we can see that we inherit features from more than just one parent. When strangers tell a proud mother that her son has, "his fathers eyes", she will typically respond along the lines of, "yes, but he got my nose."

Object-oriented design can also feature such **multiple inheritance**, which allows a subclass to inherit functionality from multiple parent classes. In practice, multiple inheritance can be a tricky business, and some programming languages (most notably, Java) strictly prohibit it. However, multiple inheritance can have its uses. Most often, it can be used to create objects that have two distinct sets of behaviors. For example, an object designed to connect to a scanner and send a fax of the scanned document might be created by inheriting from two separate `scanner` and `faxer` objects.

As long as two classes have distinct interfaces, it is not normally harmful for a subclass to inherit from both of them. However, it gets messy if we inherit from two classes that provide overlapping interfaces. For example, if we have a motorcycle class that has a `move` method, and a boat class also featuring a `move` method, and we want to merge them into the ultimate amphibious vehicle, how does the resulting class know what to do when we call `move`? At the design level, this needs to be explained, and at the implementation level, each programming language has different ways of deciding which parent class's method is called, or in what order.

Often, the best way to deal with it is to avoid it. If you have a design showing up like this, you're *probably* doing it wrong. Take a step back, analyze the system again, and see if you can remove the multiple inheritance relationship in favor of some other association or composite design.

Inheritance is a very powerful tool for extending behavior. It is also one of the most marketable advancements of object-oriented design over earlier paradigms. Therefore, it is often the first tool that object-oriented programmers reach for. However, it is important to recognize that owning a hammer does not turn screws into nails. Inheritance is the perfect solution for obvious *is a* relationships, but it can be abused. Programmers often use inheritance to share code between two kinds of objects that are only distantly related, with no *is a* relationship in sight. While this is not necessarily a bad design, it is a terrific opportunity to ask just why they decided to design it that way, and whether a different relationship or design pattern would have been more suitable.

Case study

Let's tie all our new object-oriented knowledge together by going through a few iterations of object-oriented design on a somewhat real-world example. The system we'll be modeling is a library catalog. Libraries have been tracking their inventory for centuries, originally using card catalogs, and more recently, electronic inventories. Modern libraries have web-based catalogs that we can query from our homes.

Let's start with an analysis. The local librarian has asked us to write a new card catalog program because their ancient DOS-based program is ugly and out of date. That doesn't give us much detail, but before we start asking for more information, let's consider what we already know about library catalogs.

Catalogs contain lists of books. People search them to find books on certain subjects, with specific titles, or by a particular author. Books can be uniquely identified by an **International Standard Book Number** (**ISBN**). Each book has a **Dewey Decimal System** (**DDS**) number assigned to help find it on a particular shelf.

This simple analysis tells us some of the obvious objects in the system. We quickly identify **Book** as the most important object, with several attributes already mentioned, such as author, title, subject, ISBN, and DDS number, and catalog as a sort of manager for books.

We also notice a few other objects that may or may not need to be modeled in the system. For cataloging purposes, all we need to search a book by author is an `author_name` attribute on the book. However, authors are also objects, and we might want to store some other data about the author. As we ponder this, we might remember that some books have multiple authors. Suddenly, the idea of having a single `author_name` attribute on objects seems a bit silly. A list of authors associated with each book is clearly a better idea.

The relationship between author and book is clearly association, since you would never say, "a book is an author" (it's not inheritance), and saying "a book has an author", though grammatically correct, does not imply that authors are part of books (it's not aggregation). Indeed, any one author may be associated with multiple books.

We should also pay attention to the noun (nouns are always good candidates for objects) *shelf*. Is a shelf an object that needs to be modeled in a cataloging system? How do we identify an individual shelf? What happens if a book is stored at the end of one shelf, and later moved to the beginning of the next shelf because another book was inserted in the previous shelf?

DDS was designed to help locate physical books in a library. As such, storing a DDS attribute with the book should be enough to locate it, regardless of which shelf it is stored on. So we can, at least for the moment, remove shelf from our list of contending objects.

Another questionable object in the system is the user. Do we need to know anything about a specific user, such as their name, address, or list of overdue books? So far, the librarian has told us only that they want a catalog; they said nothing about tracking subscriptions or overdue notices. In the back of our minds, we also note that authors and users are both specific kinds of people; there might be a useful inheritance relationship here in the future.

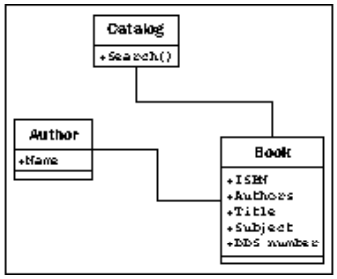
For cataloging purposes, we decide we don't need to identify the user for now. We can assume that a user will be searching the catalog, but we don't have to actively model them in the system, beyond providing an interface that allows them to search.

We have identified a few attributes on the book, but what properties does a catalog have? Does any one library have more than one catalog? Do we need to uniquely identify them? Obviously, the catalog has to have a collection of the books it contains, somehow, but this list is probably not part of the public interface.

What about behaviors? The catalog clearly needs a search method, possibly separate ones for authors, titles, and subjects. Are there any behaviors on books? Would it need a preview method? Or could preview be identified by a first pages attribute instead of a method?

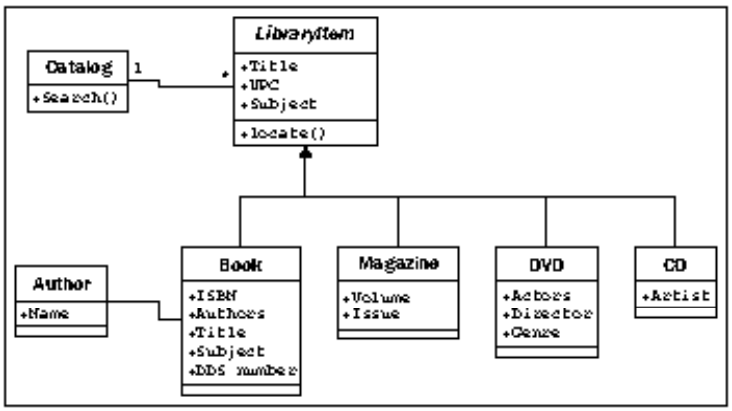
The questions in the preceding discussion are all part of the object-oriented analysis phase. But intermixed with the questions, we have already identified a few key objects that are part of the design. Indeed, what you have just seen are several microiterations between analysis and design.

Likely, these iterations would all occur in an initial meeting with the librarian. Before this meeting, however, we can already sketch out a most basic design for the objects we have concretely identified:



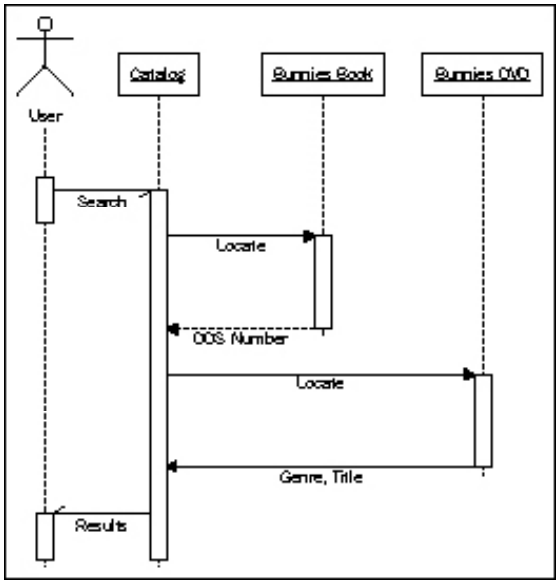
Armed with this basic diagram and a pencil to interactively improve it, we meet up with the librarian. They tell us that this is a good start, but libraries don't serve only books, they also have DVDs, magazines, and CDs, none of which have an ISBN or DDS number. All of these types of items can be uniquely identified by a UPC number though. We remind the librarian that they have to find the items on the shelf, and these items probably aren't organized by UPC. The librarian explains that each type is organized in a different way. The CDs are mostly audio books, and they only have a couple of dozen in stock, so they are organized by the author's last name. DVDs are divided into genre and further organized by title. Magazines are organized by title and then refined by the volume and issue number. Books are, as we had guessed, organized by the DDS number.

With no previous object-oriented design experience, we might consider adding separate lists of DVDs, CDs, magazines, and books to our catalog, and search each one in turn. The trouble is, except for certain extended attributes, and identifying the physical location of the item, these items all behave as much the same. This is a job for inheritance! We quickly update our UML diagram:



The librarian understands the gist of our sketched diagram, but is a bit confused by the **locate** functionality. We explain using a specific use case where the user is searching for the word "bunnies". The user first sends a search request to the catalog. The catalog queries its internal list of items and finds a book and a DVD with "bunnies" in the title. At this point, the catalog doesn't care if it is holding a DVD, book, CD, or magazine; all items are the same, as far as the catalog is concerned. However, the user wants to know how to find the physical items, so the catalog would be remiss if it simply returned a list of titles. So, it calls the **locate** method on the two items it has uncovered. The book's **locate** method returns a DDS number that can be used to find the shelf holding the book. The DVD is located by returning the genre and title of the DVD. The user can then visit the DVD section, find the section containing that genre, and find the specific DVD as sorted by the titles.

As we explain, we sketch a UML **sequence diagram** explaining how the various objects are communicating:



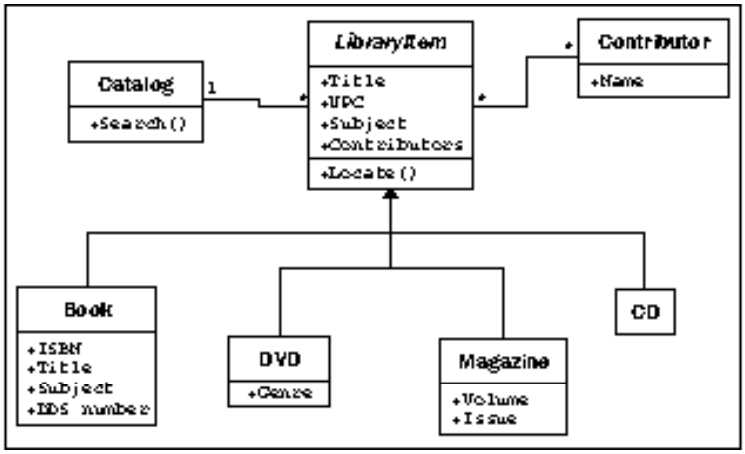
Where, class diagrams describe the relationships between classes, and sequence diagrams describe specific sequences of messages passed between objects. The dashed line hanging from each object is a **lifeline** describing the lifetime of the object. The wider boxes on each lifeline represent active processing in that object (where there's no box, the object is basically sitting idle, waiting for something to happen). The horizontal arrows between the lifelines indicate specific messages. The solid arrows represent methods being called, while the dashed arrows with solid heads represent the method return values.

The half arrowheads indicate asynchronous messages sent to or from an object. An asynchronous message typically means the first object calls a method on the second object, which returns immediately. After some processing, the second object calls a method on the first object to give it a value. This is in contrast to normal method calls, which do the processing in the method, and return a value immediately.

Sequence diagrams, like all UML diagrams, are best used only when they are needed. There is no point in drawing a UML diagram for the sake of drawing a diagram. However, when you need to communicate a series of interactions between two objects, the sequence diagram is a very useful tool.

Unfortunately, our class diagram so far is still a messy design. We notice that actors on DVDs and artists on CDs are all types of people, but are being treated differently from the book authors. The librarian also reminds us that most of their CDs are audio books, which have authors instead of artists.

How can we deal with different kinds of people that contribute to a title? An obvious implementation is to create a `Person` class with the person's name and other relevant details, and then create subclasses of this for the artists, authors, and actors. However, is inheritance really necessary here? For searching and cataloging purposes, we don't really care that acting and writing are two very different activities. If we were doing an economic simulation, it would make sense to give separate actor and author classes, and different `calculate_income` and `perform_job` methods, but for cataloging purposes, it is probably enough to know how the person contributed to the item. We recognize that all items have one or more `Contributor` objects, so we move the author relationship from the book to its parent class:

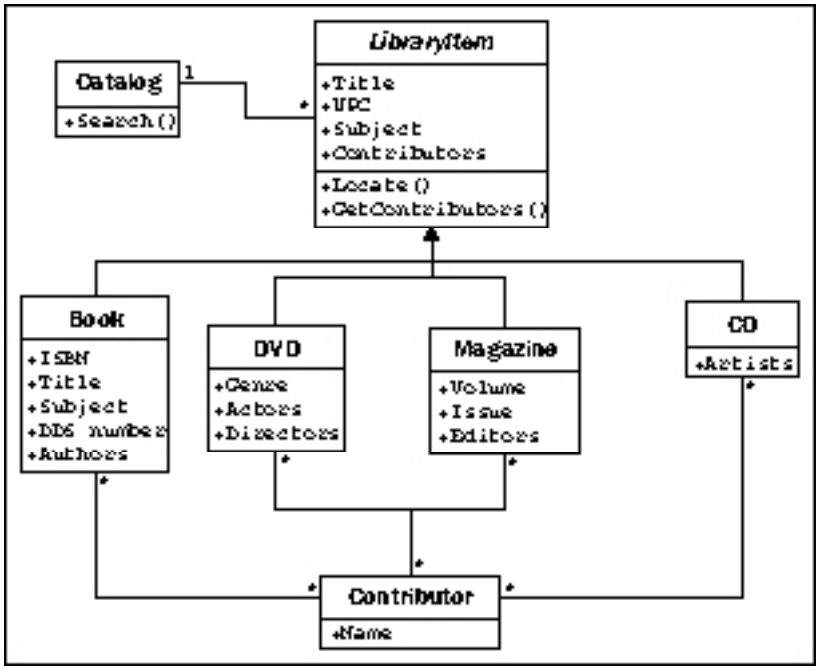


The multiplicity of the **Contributor/LibraryItem** relationship is **many-to-many**, as indicated by the * character at both ends of one relationship. Any one library item might have more than one contributor (for example, several actors and a director on a DVD). And many authors write many books, so they would be attached to multiple library items.

This little change, while it looks a bit cleaner and simpler, has lost some vital information. We can still tell who contributed to a specific library item, but we don't know how they contributed. Were they the director or an actor? Did they write the audio book, or were they the voice that narrated the book?

It would be nice if we could just add a `contributor_type` attribute on the **Contributor** class, but this will fall apart when dealing with multitalented people who have both authored books and directed movies.

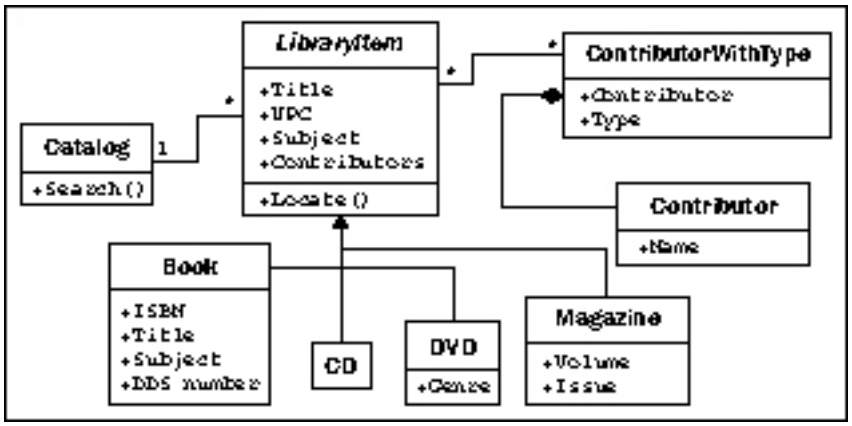
One option is to add attributes to each of our **LibraryItem** subclasses that hold the information we need, such as **Author** on **Book**, or **Artist** on **CD**, and then make the relationship to those properties all point to the **Contributor** class. The problem with this is that we lose a lot of polymorphic elegance. If we want to list the contributors to an item, we have to look for specific attributes on that item, such as **Authors** or **Actors**. We can alleviate this by adding a **GetContributors** method on the **LibraryItem** class that subclasses can override. Then the catalog never has to know what attributes the objects are querying; we've abstracted the public interface:



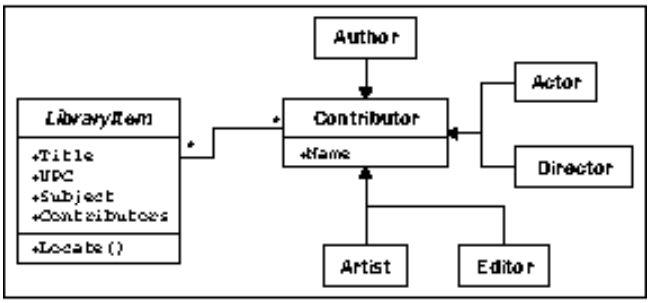
Just looking at this class diagram, it feels like we are doing something wrong. It is bulky and fragile. It may do everything we need, but it feels like it will be hard to maintain or extend. There are too many relationships, and too many classes would be affected by modifications to any one class. It looks like spaghetti and meatballs.

Now that we've explored inheritance as an option, and found it wanting, we might look back at our previous composition-based diagram, where **Contributor** was attached directly to **LibraryItem**. With some thought, we can see that we actually only need to add one more relationship to a brand-new class to identify the type of contributor. This is an important step in object-oriented design. We are now adding a class to the design that is intended to *support* the other objects, rather than modeling any part of the initial requirements. We are **refactoring** the design to facilitate the objects in the system, rather than objects in real life. Refactoring is an essential process in the maintenance of a program or design. The goal of refactoring is to improve the design by moving code around, removing duplicate code or complex relationships in favor of simpler, more elegant designs.

This new class is composed of a **Contributor** and an extra attribute identifying the type of contribution the person has made to the given **LibraryItem**. There can be many such contributions to a particular **LibraryItem**, and one contributor can contribute in the same way to different items. The diagram communicates this design very well:



At first, this composition relationship looks less natural than the inheritance-based relationships. However, it has the advantage of allowing us to add new types of contributions without adding a new class to the design. Inheritance is most useful when the subclasses have some kind of specialization. Specialization is creating or changing attributes or behaviors on the subclass to make it somehow different from the parent class. It seems silly to create a bunch of empty classes solely for identifying different types of objects (this attitude is less prevalent among Java and other "everything is an object" programmers, but it is common among more practical Python designers). If we look at the inheritance version of the diagram, we can see a bunch of subclasses that don't actually do anything:



Sometimes it is important to recognize when not to use object-oriented principles. This example of when not to use inheritance is a good reminder that objects are just tools, and not rules.

Exercises

This is a practical book, not a textbook. As such, I'm not about to assign you a bunch of fake object-oriented analysis problems to create designs for bunch of fake object-oriented problems to analyze and design. Instead, I want to give you some thoughts that you can apply to your own projects. If you have previous object-oriented experience, you won't need to put much effort into these. However, they are useful mental exercises if you've been using Python for a while, but never really cared about all that class stuff.

First, think about a recent programming project you've completed. Identify the most prominent object in the design. Try to think of as many attributes for this object as possible. Did it have: Color? Weight? Size? Profit? Cost? Name? ID number? Price? Style? Think about the attribute types. Were they primitives or classes? Were some of those attributes actually behaviors in disguise? Sometimes what looks like data is actually calculated from other data on the object, and you can use a method to do those calculations. What other methods or behaviors did the object have? Which objects called those methods? What kinds of relationships did they have with this object?

Now, think about an upcoming project. It doesn't matter what the project is; it might be a fun free-time project or a multimillion dollar contract. It doesn't have to be a complete application; it could just be one subsystem. Perform a basic object-oriented analysis. Identify the requirements and the interacting objects. Sketch out a class diagram featuring the highest level of abstraction on that system. Identify the major interacting objects. Identify minor supporting objects. Go into detail for the attributes and methods of some of the most interesting ones. Take different objects to different levels of abstraction. Look for places you can use inheritance or composition. Look for places you should avoid inheritance.

The goal is not to design a system (although you're certainly welcome to do so if inclination meets both ambition and available time). The goal is to think about object-oriented designs. Focusing on projects that you have worked on, or are expecting to work on in the future, simply makes it real.

Now, visit your favorite search engine and look up some tutorials on UML. There are dozens, so find the one that suits your preferred method of study. Sketch some class diagrams or a sequence diagram for the objects you identified earlier. Don't get too hung up on memorizing the syntax (after all, if it is important, you can always look it up again), just get a feel for the language. Something will stay lodged in your brain, and it can make communicating a bit easier if you can quickly sketch a diagram for your next OOP discussion.

Summary

In this chapter, we took a whirlwind tour through the terminology of the object-oriented paradigm, focusing on object-oriented design. We can separate different objects into a taxonomy of different classes and describe the attributes and behaviors of those objects via the class interface. Classes describe objects, abstraction, encapsulation, and information hiding are highly related concepts. There are many different kinds of relationships between objects, including association, composition, and inheritance. UML syntax can be useful for fun and communication.

In the next chapter, we'll explore how to implement classes and methods in Python.

2
Objects in Python

So, we now have a design in hand and are ready to turn that design into a working program! Of course, it doesn't usually happen this way. We'll be seeing examples and hints for good software design throughout the book, but our focus is object-oriented programming. So, let's have a look at the Python syntax that allows us to create object-oriented software.

After completing this chapter, we will understand:

- How to create classes and instantiate objects in Python
- How to add attributes and behaviors to Python objects
- How to organize classes into packages and modules
- How to suggest people don't clobber our data

Creating Python classes

We don't have to write much Python code to realize that Python is a very "clean" language. When we want to do something, we just do it, without having to go through a lot of setup. The ubiquitous "hello world" in Python, as you've likely seen, is only one line.

Similarly, the simplest class in Python 3 looks like this:

```
class MyFirstClass:
    pass
```

There's our first object-oriented program! The class definition starts with the `class` keyword. This is followed by a name (of our choice) identifying the class, and is terminated with a colon.

The class name must follow standard Python variable naming rules (it must start with a letter or underscore, and can only be comprised of letters, underscores, or numbers). In addition, the Python style guide (search the web for "PEP 8") recommends that classes should be named using **CamelCase** notation (start with a capital letter; any subsequent words should also start with a capital).

The class definition line is followed by the class contents indented. As with other Python constructs, indentation is used to delimit the classes, rather than braces or brackets as many other languages use. Use four spaces for indentation unless you have a compelling reason not to (such as fitting in with somebody else's code that uses tabs for indents). Any decent programming editor can be configured to insert four spaces whenever the *Tab* key is pressed.

Since our first class doesn't actually do anything, we simply use the `pass` keyword on the second line to indicate that no further action needs to be taken.

We might think there isn't much we can do with this most basic class, but it does allow us to instantiate objects of that class. We can load the class into the Python 3 interpreter, so we can interactively play with it. To do this, save the class definition mentioned earlier into a file named `first_class.py` and then run the command `python -i first_class.py`. The `-i` argument tells Python to "run the code and then drop to the interactive interpreter". The following interpreter session demonstrates basic interaction with this class:

```
>>> a = MyFirstClass()
>>> b = MyFirstClass()
>>> print(a)
<__main__.MyFirstClass object at 0xb7b7faec>
>>> print(b)
<__main__.MyFirstClass object at 0xb7b7fbac>
>>>
```

This code instantiates two objects from the new class, named `a` and `b`. Creating an instance of a class is a simple matter of typing the class name followed by a pair of parentheses. It looks much like a normal function call, but Python knows we're "calling" a class and not a function, so it understands that its job is to create a new object. When printed, the two objects tell us which class they are and what memory address they live at. Memory addresses aren't used much in Python code, but here, they demonstrate that there are two distinct objects involved.

Adding attributes

Now, we have a basic class, but it's fairly useless. It doesn't contain any data, and it doesn't do anything. What do we have to do to assign an attribute to a given object?

It turns out that we don't have to do anything special in the class definition. We can set arbitrary attributes on an instantiated object using the dot notation:

```
class Point:
    pass

p1 = Point()
p2 = Point()

p1.x = 5
p1.y = 4

p2.x = 3
p2.y = 6

print(p1.x, p1.y)
print(p2.x, p2.y)
```

If we run this code, the two `print` statements at the end tell us the new attribute values on the two objects:

```
5 4
3 6
```

This code creates an empty `Point` class with no data or behaviors. Then it creates two instances of that class and assigns each of those instances `x` and `y` coordinates to identify a point in two dimensions. All we need to do to assign a value to an attribute on an object is use the <*object*>.<*attribute*> = <*value*> syntax. This is sometimes referred to as **dot notation**. The value can be anything: a Python primitive, a built-in data type, or another object. It can even be a function or another class!

Making it do something

Now, having objects with attributes is great, but object-oriented programming is really about the interaction between objects. We're interested in invoking actions that cause things to happen to those attributes. It is time to add behaviors to our classes.

Let's model a couple of actions on our `Point` class. We can start with a called `reset` that moves the point to the origin (the origin is the point where `x` and `y` are both zero). This is a good introductory action because it doesn't require any parameters:

```
class Point:
    def reset(self):
        self.x = 0
        self.y = 0

p = Point()
p.reset()
print(p.x, p.y)
```

This `print` statement shows us the two zeros on the attributes:

```
0 0
```

A method in Python is formatted identically to a function. It starts with the keyword `def` followed by a space and the name of the method. This is followed by a set of parentheses containing the parameter list (we'll discuss that `self` parameter in just a moment), and terminated with a colon. The next line is indented to contain the statements inside the method. These statements can be arbitrary Python code operating on the object itself and any parameters passed in as the method sees fit.

Talking to yourself

The one difference between methods and normal functions is that all methods have one required argument. This argument is conventionally named `self`; I've never seen a programmer use any other name for this variable (convention is a very powerful thing). There's nothing stopping you, however, from calling it `this` or even `Martha`.

The `self` argument to a method is simply a reference to the object that the method is being invoked on. We can access attributes and methods of that object as if it were any another object. This is exactly what we do inside the `reset` method when we set the `x` and `y` attributes of the `self` object.

Notice that when we call the `p.reset()` method, we do not have to pass the `self` argument into it. Python automatically takes care of this for us. It knows we're calling a method on the `p` object, so it automatically passes that object to the method.

However, the method really is just a function that happens to be on a class. Instead of calling the method on the object, we can invoke the function on the class, explicitly passing our object as the `self` argument:

```
p = Point()
Point.reset(p)
print(p.x, p.y)
```

The output is the same as the previous example because internally, the exact same process has occurred.

What happens if we forget to include the `self` argument in our class definition? Python will bail with an error message:

```
>>> class Point:
...     def reset():
...         pass
...
>>> p = Point()
>>> p.reset()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: reset() takes no arguments (1 given)
```

The error message is not as clear as it could be ("You silly fool, you forgot the `self` argument" would be more informative). Just remember that when you see an error message that indicates missing arguments, the first thing to check is whether you forgot `self` in the method definition.

More arguments

So, how do we pass multiple arguments to a method? Let's add a new method that allows us to move a point to an arbitrary position, not just to the origin. We can also include one that accepts another `Point` object as input and returns the distance between them:

```
import math

class Point:
```

```
        def move(self, x, y):
            self.x = x
            self.y = y

        def reset(self):
            self.move(0, 0)

        def calculate_distance(self, other_point):
            return math.sqrt(
                    (self.x - other_point.x)**2 +
                    (self.y - other_point.y)**2)

    # how to use it:
    point1 = Point()
    point2 = Point()

    point1.reset()
    point2.move(5,0)
    print(point2.calculate_distance(point1))
    assert (point2.calculate_distance(point1) ==
            point1.calculate_distance(point2))
    point1.move(3,4)
    print(point1.calculate_distance(point2))
    print(point1.calculate_distance(point1))
```

The `print` statements at the end give us the following output:

```
5.0
4.472135955
0.0
```

A lot has happened here. The class now has three methods. The `move` method accepts two arguments, `x` and `y`, and sets the values on the `self` object, much like the old `reset` method from the previous example. The old `reset` method now calls `move`, since a reset is just a move to a specific known location.

The `calculate_distance` method uses the not-too-complex Pythagorean theorem to calculate the distance between two points. I hope you understand the math (`**` means squared, and `math.sqrt` calculates a square root), but it's not a requirement for our current focus, learning how to write methods.

The sample code at the end of the preceding example shows how to call a method with arguments: simply include the arguments inside the parentheses, and use the same dot notation to access the method. I just picked some random positions to test the methods. The test code calls each method and prints the results on the console. The `assert` function is a simple test tool; the program will bail if the statement after `assert` is `False` (or zero, empty, or `None`). In this case, we use it to ensure that the distance is the same regardless of which point called the other point's `calculate_distance` method.

Initializing the object

If we don't explicitly set the `x` and `y` positions on our `Point` object, either using `move` or by accessing them directly, we have a broken point with no real position. What will happen when we try to access it?

Well, let's just try it and see. "Try it and see" is an extremely useful tool for Python study. Open up your interactive interpreter and type away. The following interactive session shows what happens if we try to access a missing attribute. If you saved the previous example as a file or are using the examples distributed with the book, you can load it into the Python interpreter with the command `python -i filename.py`:

```
>>> point = Point()
>>> point.x = 5
>>> print(point.x)
5
>>> print(point.y)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Point' object has no attribute 'y'
```

Well, at least it threw a useful exception. We'll cover exceptions in detail in *Chapter 4*, *Expecting the Unexpected*. You've probably seen them before (especially the ubiquitous **SyntaxError**, which means you typed something incorrectly!). At this point, simply be aware that it means something went wrong.

The output is useful for debugging. In the interactive interpreter, it tells us the error occurred at **line 1**, which is only partially true (in an interactive session, only one line is executed at a time). If we were running a script in a file, it would tell us the exact line number, making it easy to find the offending code. In addition, it tells us the error is an `AttributeError`, and gives a helpful message telling us what that error means.

We can catch and recover from this error, but in this case, it feels like we should have specified some sort of default value. Perhaps every new object should be `reset()` by default, or maybe it would be nice if we could force the user to tell us what those positions should be when they create the object.

Most object-oriented programming languages have the concept of a **constructor**, a special method that creates and initializes the object when it is created. Python is a little different; it has a constructor *and* an initializer. The constructor function is rarely used unless you're doing something exotic. So, we'll start our discussion with the initialization method.

The Python initialization method is the same as any other method, except it has a special name, `__init__`. The leading and trailing double underscores mean this is a special method that the Python interpreter will treat as a special case.

Never name a function of your own with leading and trailing double underscores. It may mean nothing to Python, but there's always the possibility that the designers of Python will add a function that has a special purpose with that name in the future, and when they do, your code will break.

Let's start with an initialization function on our `Point` class that requires the user to supply `x` and `y` coordinates when the `Point` object is instantiated:

```
class Point:
    def __init__(self, x, y):
        self.move(x, y)

    def move(self, x, y):
        self.x = x
        self.y = y

    def reset(self):
        self.move(0, 0)

# Constructing a Point
point = Point(3, 5)
print(point.x, point.y)
```

Now, our point can never go without a `y` coordinate! If we try to construct a point without including the proper initialization parameters, it will fail with a **not enough arguments** error similar to the one we received earlier when we forgot the `self` argument.

What if we don't want to make those two arguments required? Well, then we can use the same syntax Python functions use to provide default arguments. The keyword argument syntax appends an equals sign after each variable name. If the calling object does not provide this argument, then the default argument is used instead. The variables will still be available to the function, but they will have the values specified in the argument list. Here's an example:

```
class Point:
    def __init__(self, x=0, y=0):
        self.move(x, y)
```

Most of the time, we put our initialization statements in an `__init__` function. But as mentioned earlier, Python has a constructor in addition to its initialization function. You may never need to use the other Python constructor, but it helps to know it exists, so we'll cover it briefly.

The constructor function is called `__new__` as opposed to `__init__`, and accepts exactly one argument; the class that is being constructed (it is called *before* the object is constructed, so there is no `self` argument). It also has to return the newly created object. This has interesting possibilities when it comes to the complicated art of metaprogramming, but is not very useful in day-to-day programming. In practice, you will rarely, if ever, need to use `__new__` and `__init__` will be sufficient.

Explaining yourself

Python is an extremely easy-to-read programming language; some might say it is self-documenting. However, when doing object-oriented programming, it is important to write API documentation that clearly summarizes what each object and method does. Keeping documentation up-to-date is difficult; the best way to do it is to write it right into our code.

Python supports this through the use of **docstrings**. Each class, function, or method header can have a standard Python string as the first line following the definition (the line that ends in a colon). This line should be indented the same as the following code.

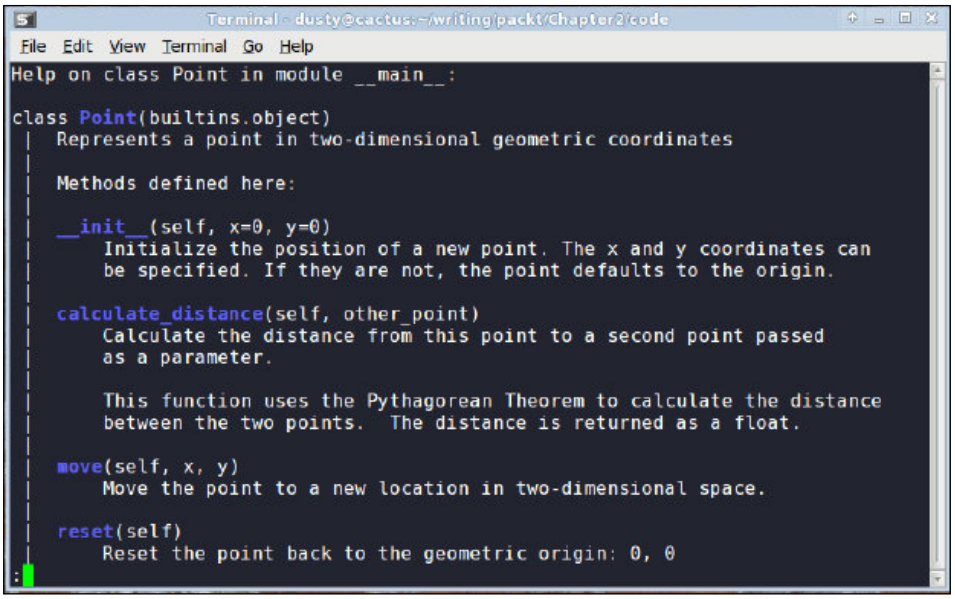
Docstrings are simply Python strings enclosed with apostrophe (`'`) or quote (`"`) characters. Often, docstrings are quite long and span multiple lines (the style guide suggests that the line length should not exceed 80 characters), which can be formatted as multi-line strings, enclosed in matching triple apostrophe (`'''`) or triple quote (`"""`) characters.

A docstring should clearly and concisely summarize the purpose of the class or method it is describing. It should explain any parameters whose usage is not immediately obvious, and is also a good place to include short examples of how to use the API. Any caveats or problems an unsuspecting user of the API should be aware of should also be noted.

To illustrate the use of docstrings, we will end this section with our completely documented `Point` class:

```
import math

class Point:
    'Represents a point in two-dimensional geometric coordinates'

    def __init__(self, x=0, y=0):
        '''Initialize the position of a new point. The x and y
coordinates can be specified. If they are not, the
           point defaults to the origin.'''
        self.move(x, y)

    def move(self, x, y):
        "Move the point to a new location in 2D space."
        self.x = x
        self.y = y

    def reset(self):
        'Reset the point back to the geometric origin: 0, 0'
        self.move(0, 0)

    def calculate_distance(self, other_point):
        """Calculate the distance from this point to a second
        point passed as a parameter.

        This function uses the Pythagorean Theorem to calculate
the distance between the two points. The distance is
        returned as a float."""

        return math.sqrt(
                (self.x - other_point.x)**2 +
                (self.y - other_point.y)**2)
```

Try typing or loading (remember, it's `python -i filename.py`) this file into the interactive interpreter. Then, enter `help(Point)<enter>` at the Python prompt. You should see nicely formatted documentation for the class, as shown in the following screenshot:



Modules and packages

Now, we know how to create classes and instantiate objects, but how do we organize them? For small programs, we can just put all our classes into one file and add a little script at the end of the file to start them interacting. However, as our projects grow, it can become difficult to find the one class that needs to be edited among the many classes we've defined. This is where **modules** come in. Modules are simply Python files, nothing more. The single file in our small program is a module. Two Python files are two modules. If we have two files in the same folder, we can load a class from one module for use in the other module.

For example, if we are building an e-commerce system, we will likely be storing a lot of data in a database. We can put all the classes and functions related to database access into a separate file (we'll call it something sensible: `database.py`). Then, our other modules (for example, customer models, product information, and inventory) can import classes from that module in order to access the database.

The `import` statement is used for importing modules or specific classes or functions from modules. We've already seen an example of this in our `Point` class in the previous section. We used the `import` statement to get Python's built-in `math` module and use its `sqrt` function in our `distance` calculation.

Here's a concrete example. Assume we have a module called `database.py` that contains a class called `Database`, and a second module called `products.py` that is responsible for product-related queries. At this point, we don't need to think too much about the contents of these files. What we know is that `products.py` needs to instantiate the `Database` class from `database.py` so that it can execute queries on the product table in the database.

There are several variations on the `import` statement syntax that can be used to access the class:

```
import database
db = database.Database()
# Do queries on db
```

This version imports the `database` module into the `products` namespace (the list of names currently accessible in a module or function), so any class or function in the `database` module can be accessed using the `database.<something>` notation. Alternatively, we can import just the one class we need using the `from...import` syntax:

```
from database import Database
db = Database()
# Do queries on db
```

If, for some reason, `products` already has a class called `Database`, and we don't want the two names to be confused, we can rename the class when used inside the `products` module:

```
from database import Database as DB
db = DB()
# Do queries on db
```

We can also import multiple items in one statement. If our `database` module also contains a `Query` class, we can import both classes using:

```
from database import Database, Query
```

Some sources say that we can import all classes and functions from the `database` module using this syntax:

```
from database import *
```

Don't do this. Every experienced Python programmer will tell you that you should never use this syntax. They'll use obscure justifications such as "it clutters up the namespace", which doesn't make much sense to beginners. One way to learn why to avoid this syntax is to use it and try to understand your code two years later. But we can save some time and two years of poorly written code with a quick explanation now!

When we explicitly import the `database` class at the top of our file using `from database import Database`, we can easily see where the `Database` class comes from. We might use `db = Database()` 400 lines later in the file, and we can quickly look at the imports to see where that `Database` class came from. Then, if we need clarification as to how to use the `Database` class, we can visit the original file (or import the module in the interactive interpreter and use the `help(database.Database)` command). However, if we use the `from database import *` syntax, it takes a lot longer to find where that class is located. Code maintenance becomes a nightmare.

In addition, most editors are able to provide extra functionality, such as reliable code completion, the ability to jump to the definition of a class, or inline documentation, if normal imports are used. The `import *` syntax usually completely destroys their ability to do this reliably.

Finally, using the `import *` syntax can bring unexpected objects into our local namespace. Sure, it will import all the classes and functions defined in the module being imported from, but it will also import any classes or modules that were themselves imported into that file!

Every name used in a module should come from a well-specified place, whether it is defined in that module, or explicitly imported from another module. There should be no magic variables that seem to come out of thin air. We should *always* be able to immediately identify where the names in our current namespace originated. I promise that if you use this evil syntax, you will one day have extremely frustrating moments of "where on earth can this class be coming from?".

Organizing the modules

As a project grows into a collection of more and more modules, we may find that we want to add another level of abstraction, some kind of nested hierarchy on our modules' levels. However, we can't put modules inside modules; one file can hold only one file after all, and modules are nothing more than Python files.

Files, however, can go in folders and so can modules. A **package** is a collection of modules in a folder. The name of the package is the name of the folder. All we need to do to tell Python that a folder is a package is place a (normally empty) file in the folder named `__init__.py`. If we forget this file, we won't be able to import modules from that folder.

Let's put our modules inside an `ecommerce` package in our working folder, which will also contain a `main.py` file to start the program. Let's additionally add another package in the `ecommerce` package for various payment options. The folder hierarchy will look like this:

```
parent_directory/
    main.py
    ecommerce/
        __init__.py
        database.py
        products.py
        payments/
            __init__.py
            square.py
            stripe.py
```

When importing modules or classes between packages, we have to be cautious about the syntax. In Python 3, there are two ways of importing modules: absolute imports and relative imports.

Absolute imports

Absolute imports specify the complete path to the module, function, or path we want to import. If we need access to the `Product` class inside the `products` module, we could use any of these syntaxes to do an absolute import:

```
import ecommerce.products
product = ecommerce.products.Product()
```

or

```
from ecommerce.products import Product
product = Product()
```

or

```
from ecommerce import products
product = products.Product()
```

The `import` statements use the period operator to separate packages or modules.

These statements will work from any module. We could instantiate a `Product` class using this syntax in `main.py`, in the `database` module, or in either of the two payment modules. Indeed, assuming the packages are available to Python, it will be able to import them. For example, the packages can also be installed to the Python site packages folder, or the `PYTHONPATH` environment variable could be customized to dynamically tell Python what folders to search for packages and modules it is going to import.

So, with these choices, which syntax do we choose? It depends on your personal taste and the application at hand. If there are dozens of classes and functions inside the `products` module that I want to use, I generally import the module name using the `from ecommerce import products` syntax, and then access the individual classes using `products.Product`. If I only need one or two classes from the `products` module, I can import them directly using the `from ecommerce.proucts import Product` syntax. I don't personally use the first syntax very often unless I have some kind of name conflict (for example, I need to access two completely different modules called `products` and I need to separate them). Do whatever you think makes your code look more elegant.

Relative imports

When working with related modules in a package, it seems kind of silly to specify the full path; we know what our parent module is named. This is where **relative imports** come in. Relative imports are basically a way of saying find a class, function, or module as it is positioned relative to the current module. For example, if we are working in the `products` module and we want to import the `Database` class from the `database` module next to it, we could use a relative import:

```
from .database import Database
```

The period in front of `database` says "u*se the database module inside the current package*". In this case, the current package is the package containing the `products.py` file we are currently editing, that is, the `ecommerce` package.

If we were editing the `paypal` module inside the `ecommerce.payments` package, we would want to say "u*se the database package inside the parent package*" instead. This is easily done with two periods, as shown here:

```
from ..database import Database
```

We can use more periods to go further up the hierarchy. Of course, we can also go down one side and back up the other. We don't have a deep enough example hierarchy to illustrate this properly, but the following would be a valid import if we had an `ecommerce.contact` package containing an `email` module and wanted to import the `send_mail` function into our `paypal` module:

```
from ..contact.email import send_mail
```

This import uses two periods to say, *the parent of the payments package*, and then uses the normal `package.module` syntax to go back *up* into the contact package.

Finally, we can import code directly from packages, as opposed to just modules inside packages. In this example, we have an `ecommerce` package containing two modules named `database.py` and `products.py`. The database module contains a `db` variable that is accessed from a lot of places. Wouldn't it be convenient if this could be imported as `import ecommerce.db` instead of `import ecommerce.database.db`?

Remember the `__init__.py` file that defines a directory as a package? This file can contain any variable or class declarations we like, and they will be available as part of the package. In our example, if the `ecommerce/__init__.py` file contained this line:

```
from .database import db
```

We can then access the `db` attribute from `main.py` or any other file using this import:

```
from ecommerce import db
```

It might help to think of the `__init__.py` file as if it was an `ecommerce.py` file if that file were a module instead of a package. This can also be useful if you put all your code in a single module and later decide to break it up into a package of modules. The `__init__.py` file for the new package can still be the main point of contact for other modules talking to it, but the code can be internally organized into several different modules or subpackages.

I recommend not putting all your code in an `__init__.py` file, though. Programmers do not expect actual logic to happen in this file, and much like with `from x import *`, it can trip them up if they are looking for the declaration of a particular piece of code and can't find it until they check `__init__.py`.

Organizing module contents

Inside any one module, we can specify variables, classes, or functions. They can be a handy way to store the global state without namespace conflicts. For example, we have been importing the `Database` class into various modules and then instantiating it, but it might make more sense to have only one `database` object globally available from the `database` module. The `database` module might look like this:

```
class Database:
    # the database implementation
    pass

database = Database()
```

Then we can use any of the import methods we've discussed to access the `database` object, for example:

```
from ecommerce.database import database
```

A problem with the preceding class is that the `database` object is created immediately when the module is first imported, which is usually when the program starts up. This isn't always ideal since connecting to a database can take a while, slowing down startup, or the database connection information may not yet be available. We could delay creating the database until it is actually needed by calling an `initialize_database` function to create the module-level variable:

```
class Database:
    # the database implementation
    pass

database = None

def initialize_database():
    global database
    database = Database()
```

The `global` keyword tells Python that the database variable inside `initialize_database` is the module level one we just defined. If we had not specified the variable as global, Python would have created a new local variable that would be discarded when the method exits, leaving the module-level value unchanged.

As these two examples illustrate, all module-level code is executed immediately at the time it is imported. However, if it is inside a method or function, the function will be created, but its internal code will not be executed until the function is called. This can be a tricky thing for scripts (such as the main script in our e-commerce example) that perform execution. Often, we will write a program that does something useful, and then later find that we want to import a function or class from that module in a different program. However, as soon as we import it, any code at the module level is immediately executed. If we are not careful, we can end up running the first program when we really only meant to access a couple functions inside that module.

To solve this, we should always put our startup code in a function (conventionally, called `main`) and only execute that function when we know we are running the module as a script, but not when our code is being imported from a different script. But how do we know this?

```
class UsefulClass:
    '''This class might be useful to other modules.'''
    pass

def main():
    '''creates a useful class and does something with it for our
module.'''
    useful = UsefulClass()
    print(useful)

if __name__ == "__main__":
    main()
```

Every module has a `__name__` special variable (remember, Python uses double underscores for special variables, such as a class's `__init__` method) that specifies the name of the module when it was imported. When the module is executed directly with `python module.py`, it is never imported, so the `__name__` is arbitrarily set to the string `"__main__"`. Make it a policy to wrap all your scripts in an `if __name__ == "__main__":` test, just in case you write a function you will find useful to be imported by other code someday.

So, methods go in classes, which go in modules, which go in packages. Is that all there is to it?

Actually, no. This is the typical order of things in a Python program, but it's not the only possible layout. Classes can be defined anywhere. They are typically defined at the module level, but they can also be defined inside a function or method, like this:

```
def format_string(string, formatter=None):
    '''Format a string using the formatter object, which
    is expected to have a format() method that accepts
    a string.'''
    class DefaultFormatter:
        '''Format a string in title case.'''
        def format(self, string):
            return str(string).title()

    if not formatter:
        formatter = DefaultFormatter()

    return formatter.format(string)

hello_string = "hello world, how are you today?"
print(" input: " + hello_string)
print("output: " + format_string(hello_string))
```

The output will be as follows:

```
input: hello world, how are you today?
output: Hello World, How Are You Today?
```

The `format_string` function accepts a string and optional formatter object, and then applies the formatter to that string. If no formatter is supplied, it creates a formatter of its own as a local class and instantiates it. Since it is created inside the scope of the function, this class cannot be accessed from anywhere outside of that function. Similarly, functions can be defined inside other functions as well; in general, any Python statement can be executed at any time.

These inner classes and functions are occasionally useful for one-off items that don't require or deserve their own scope at the module level, or only make sense inside a single method. However, it is not common to see Python code that frequently uses this technique.

Who can access my data?

Most object-oriented programming languages have a concept of access control. This is related to abstraction. Some attributes and methods on an object are marked private, meaning only that object can access them. Others are marked protected, meaning only that class and any subclasses have access. The rest are public, meaning any other object is allowed to access them.

Python doesn't do this. Python doesn't really believe in enforcing laws that might someday get in your way. Instead, it provides unenforced guidelines and best practices. Technically, all methods and attributes on a class are publicly available. If we want to suggest that a method should not be used publicly, we can put a note in docstrings indicating that the method is meant for internal use only (preferably, with an explanation of how the public-facing API works!).

By convention, we should also prefix an attribute or method with an underscore character, _. Python programmers will interpret this as "*this is an internal variable, think three times before accessing it directly*". But there is nothing inside the interpreter to stop them from accessing it if they think it is in their best interest to do so. Because if they think so, why should we stop them? We may not have any idea what future uses our classes may be put to.

There's another thing you can do to strongly suggest that outside objects don't access a property or method: prefix it with a double underscore, __. This will perform **name mangling** on the attribute in question. This basically means that the method can still be called by outside objects if they really want to do it, but it requires extra work and is a strong indicator that you demand that your attribute remains private. For example:

```
class SecretString:
    '''A not-at-all secure way to store a secret string.'''

    def __init__(self, plain_string, pass_phrase):
        self.__plain_string = plain_string
        self.__pass_phrase = pass_phrase

    def decrypt(self, pass_phrase):
        '''Only show the string if the pass_phrase is correct.'''
        if pass_phrase == self.__pass_phrase:
            return self.__plain_string
        else:
            return ''
```

If we load this class and test it in the interactive interpreter, we can see that it hides the plain text string from the outside world:

```
>>> secret_string = SecretString("ACME: Top Secret", "antwerp")
>>> print(secret_string.decrypt("antwerp"))
ACME: Top Secret
>>> print(secret_string.__plain_text)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'SecretString' object has no attribute
'__plain_text'
```

It looks like it works; nobody can access our `plain_text` attribute without the passphrase, so it must be safe. Before we get too excited, though, let's see how easy it can be to hack our security:

```
>>> print(secret_string._SecretString__plain_string)
ACME: Top Secret
```

Oh no! Somebody has hacked our secret string. Good thing we checked! This is Python name mangling at work. When we use a double underscore, the property is prefixed with `_<classname>`. When methods in the class internally access the variable, they are automatically unmangled. When external classes wish to access it, they have to do the name mangling themselves. So, name mangling does not guarantee privacy, it only strongly recommends it. Most Python programmers will not touch a double underscore variable on another object unless they have an extremely compelling reason to do so.

However, most Python programmers will not touch a single underscore variable without a compelling reason either. Therefore, there are very few good reasons to use a name-mangled variable in Python, and doing so can cause grief. For example, a name-mangled variable may be useful to a subclass, and it would have to do the mangling itself. Let other objects access your hidden information if they want to, just let them know, using a single-underscore prefix or some clear docstrings, that you think this is not a good idea.

Third-party libraries

Python ships with a lovely standard library, which is a collection of packages and modules that are available on every machine that runs Python. However, you'll soon find that it doesn't contain everything you need. When this happens, you have two options:

- Write a supporting package yourself
- Use somebody else's code

We won't be covering the details about turning your packages into libraries, but if you have a problem you need to solve and you don't feel like coding it (the best programmers are extremely lazy and prefer to reuse existing, proven code, rather than write their own), you can probably find the library you want on the **Python Package Index** (**PyPI**) at `http://pypi.python.org/`. Once you've identified a package that you want to install, you can use a tool called `pip` to install it. However, `pip` does not come with Python, but Python 3.4 contains a useful tool called `ensurepip`, which will install it:

```
python -m ensurepip
```

This may fail for you on Linux, Mac OS, or other Unix systems, in which case, you'll need to become root to make it work. On most modern Unix systems, this can be done with `sudo python -m ensurepip`.

If you are using an older version of Python than Python 3.4, you'll need to download and install `pip` yourself, since `ensurepip` doesn't exist. You can do this by following the instructions at `http://pip.readthedocs.org/`.

Once `pip` is installed and you know the name of the package you want to install, you can install it using syntax such as:

```
pip install requests
```

However, if you do this, you'll either be installing the third-party library directly into your system Python directory, or more likely, get an error that you don't have permission to do so. You could force the installation as an administrator, but common consensus in the Python community is that you should only use system installers to install the third-party library to your system Python directory.

Instead, Python 3.4 supplies the `venv` tool. This utility basically gives you a mini Python installation called a *virtual environment* in your working directory. When you activate the mini Python, commands related to Python will work on that directory instead of the system directory. So when you run `pip` or `python`, it won't touch the system Python at all. Here's how to use it:

```
cd project_directory
python -m venv env
source env/bin/activate  # on Linux or MacOS
env/bin/activate.bat     # on Windows
```

Typically, you'll create a different virtual environment for each Python project you work on. You can store your virtual environments anywhere, but I keep mine in the same directory as the rest of my project files (but ignored in version control), so first we `cd` into that directory. Then we run the `venv` utility to create a virtual environment named `env`. Finally, we use one of the last two lines (depending on the operating system, as indicated in the comments) to activate the environment. We'll need to execute this line each time we want to use that particular virtualenv, and then use the command `deactivate` when we are done working on this project.

Virtual environments are a terrific way to keep your third-party dependencies separate. It is common to have different projects that depend on different versions of a particular library (for example, an older website might run on Django 1.5, while newer versions run on Django 1.8). Keeping each project in separate virtualenvs makes it easy to work in either version of Django. Further, it prevents conflicts between system-installed packages and `pip` installed packages if you try to install the same package using different tools.

Case study

To tie it all together, let's build a simple command-line notebook application. This is a fairly simple task, so we won't be experimenting with multiple packages. We will, however, see common usage of classes, functions, methods, and docstrings.

Let's start with a quick analysis: notes are short memos stored in a notebook. Each note should record the day it was written and can have tags added for easy querying. It should be possible to modify notes. We also need to be able to search for notes. All of these things should be done from the command line.

The obvious object is the `Note` object; less obvious one is a `Notebook` container object. Tags and dates also seem to be objects, but we can use dates from Python's standard library and a comma-separated string for tags. To avoid complexity, in the prototype, let's not define separate classes for these objects.

`Note` objects have attributes for `memo` itself, `tags`, and `creation_date`. Each note will also need a unique integer `id` so that users can select them in a menu interface. Notes could have a method to modify note content and another for tags, or we could just let the notebook access those attributes directly. To make searching easier, we should put a `match` method on the `Note` object. This method will accept a string and can tell us if a note matches the string without accessing the attributes directly. This way, if we want to modify the search parameters (to search tags instead of note contents, for example, or to make the search case-insensitive), we only have to do it in one place.

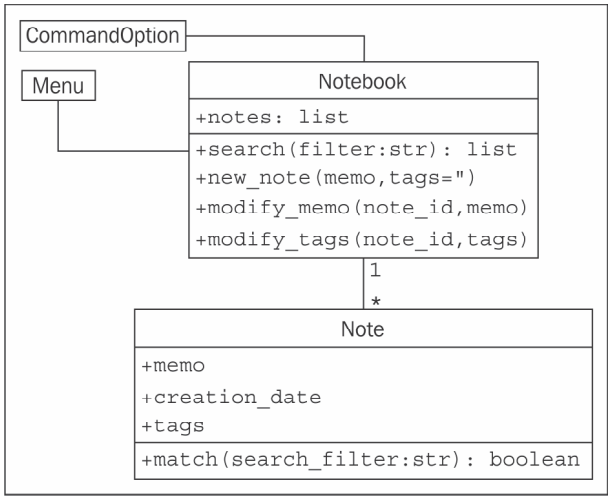
The `Notebook` object obviously has the list of notes as an attribute. It will also need a search method that returns a list of filtered notes.

But how do we interact with these objects? We've specified a command-line app, which can mean either that we run the program with different options to add or edit commands, or we have some kind of a menu that allows us to pick different things to do to the notebook. We should try to design it such that either interface is supported and future interfaces, such as a GUI toolkit or web-based interface, could be added in the future.

As a design decision, we'll implement the menu interface now, but will keep the command-line options version in mind to ensure we design our `Notebook` class with extensibility in mind.

If we have two command-line interfaces, each interacting with the `Notebook` object, then `Notebook` will need some methods for those interfaces to interact with. We need to be able to `add` a new note, and `modify` an existing note by `id`, in addition to the `search` method we've already discussed. The interfaces will also need to be able to list all notes, but they can do that by accessing the `notes` list attribute directly.

We may be missing a few details, but that gives us a really good overview of the code we need to write. We can summarize all this in a simple class diagram:



Before writing any code, let's define the folder structure for this project. The menu interface should clearly be in its own module, since it will be an executable script, and we may have other executable scripts accessing the notebook in the future. The `Notebook` and `Note` objects can live together in one module. These modules can both exist in the same top-level directory without having to put them in a package. An empty `command_option.py` module can help remind us in the future that we were planning to add new user interfaces.

```
parent_directory/
    notebook.py
    menu.py
    command_option.py
```

Now let's see some code. We start by defining the `Note` class as it seems simplest. The following example presents `Note` in its entirety. Docstrings within the example explain how it all fits together.

```
import datetime

# Store the next available id for all new notes
last_id = 0

class Note:
    '''Represent a note in the notebook. Match against a
```

```
        string in searches and store tags for each note.'''

        def __init__(self, memo, tags=''):
            '''initialize a note with memo and optional
            space-separated tags. Automatically set the note's
            creation date and a unique id.'''
            self.memo = memo
            self.tags = tags
            self.creation_date = datetime.date.today()
            global last_id
            last_id += 1
            self.id = last_id

        def match(self, filter):
            '''Determine if this note matches the filter
            text. Return True if it matches, False otherwise.

            Search is case sensitive and matches both text and
            tags.'''
            return filter in self.memo or filter in self.tags
```

Before continuing, we should quickly fire up the interactive interpreter and test our code so far. Test frequently and often because things never work the way you expect them to. Indeed, when I tested my first version of this example, I found out I had forgotten the `self` argument in the `match` function! We'll discuss automated testing in *Chapter 10, Python Design Patterns I*. For now, it suffices to check a few things using the interpreter:

```
>>> from notebook import Note
>>> n1 = Note("hello first")
>>> n2 = Note("hello again")
>>> n1.id
1
>>> n2.id
2
>>> n1.match('hello')
True
>>> n2.match('second')
False
```

It looks like everything is behaving as expected. Let's create our notebook next:

```
class Notebook:
    '''Represent a collection of notes that can be tagged,
    modified, and searched.'''

    def __init__(self):
        '''Initialize a notebook with an empty list.'''
        self.notes = []

    def new_note(self, memo, tags=''):
        '''Create a new note and add it to the list.'''
        self.notes.append(Note(memo, tags))

    def modify_memo(self, note_id, memo):
        '''Find the note with the given id and change its
        memo to the given value.'''
        for note in self.notes:
            if note.id == note_id:
                note.memo = memo
                break

    def modify_tags(self, note_id, tags):
        '''Find the note with the given id and change its
        tags to the given value.'''
        for note in self.notes:
            if note.id == note_id:
                note.tags = tags
                break

    def search(self, filter):
        '''Find all notes that match the given filter
        string.'''
        return [note for note in self.notes if
                note.match(filter)]
```

We'll clean this up in a minute. First, let's test it to make sure it works:

```
>>> from notebook import Note, Notebook
>>> n = Notebook()
>>> n.new_note("hello world")
>>> n.new_note("hello again")
>>> n.notes
[<notebook.Note object at 0xb730a78c>, <notebook.Note object at
  0xb73103ac>]
```

```
>>> n.notes[0].id
1
>>> n.notes[1].id
2
>>> n.notes[0].memo
'hello world'
>>> n.search("hello")
[<notebook.Note object at 0xb730a78c>, <notebook.Note object at
  0xb73103ac>]
>>> n.search("world")
[<notebook.Note object at 0xb730a78c>]
>>> n.modify_memo(1, "hi world")
>>> n.notes[0].memo
'hi world'
```

It does work. The code is a little messy though; our `modify_tags` and `modify_memo` methods are almost identical. That's not good coding practice. Let's see how we can improve it.

Both methods are trying to identify the note with a given ID before doing something to that note. So, let's add a method to locate the note with a specific ID. We'll prefix the method name with an underscore to suggest that the method is for internal use only, but of course, our menu interface can access the method if it wants to:

```
    def _find_note(self, note_id):
        '''Locate the note with the given id.'''
        for note in self.notes:
            if note.id == note_id:
                return note
        return None

    def modify_memo(self, note_id, memo):
        '''Find the note with the given id and change its
        memo to the given value.'''
        self._find_note(note_id).memo = memo
```

This should work for now. Let's have a look at the menu interface. The interface simply needs to present a menu and allow the user to input choices. Here's our first try:

```
import sys
```

```
from notebook import Notebook, Note

class Menu:
    '''Display a menu and respond to choices when run.'''
    def __init__(self):
        self.notebook = Notebook()
        self.choices = {
                "1": self.show_notes,
                "2": self.search_notes,
                "3": self.add_note,
                "4": self.modify_note,
                "5": self.quit
                }

    def display_menu(self):
        print("""
Notebook Menu

1. Show all Notes
2. Search Notes
3. Add Note
4. Modify Note
5. Quit
""")

    def run(self):
        '''Display the menu and respond to choices.'''
        while True:
            self.display_menu()
            choice = input("Enter an option: ")
            action = self.choices.get(choice)
            if action:
                action()
            else:
                print("{0} is not a valid choice".format(choice))

    def show_notes(self, notes=None):
        if not notes:
            notes = self.notebook.notes
        for note in notes:
            print("{0}: {1}\n{2}".format(
                note.id, note.tags, note.memo))

    def search_notes(self):
```

```
            filter = input("Search for: ")
            notes = self.notebook.search(filter)
            self.show_notes(notes)

        def add_note(self):
            memo = input("Enter a memo: ")
            self.notebook.new_note(memo)
            print("Your note has been added.")

        def modify_note(self):
            id = input("Enter a note id: ")
            memo = input("Enter a memo: ")
            tags = input("Enter tags: ")
            if memo:
                self.notebook.modify_memo(id, memo)
            if tags:
                self.notebook.modify_tags(id, tags)

        def quit(self):
            print("Thank you for using your notebook today.")
            sys.exit(0)

    if __name__ == "__main__":
        Menu().run()
```

This code first imports the notebook objects using an absolute import. Relative imports wouldn't work because we haven't placed our code inside a package. The `Menu` class's `run` method repeatedly displays a menu and responds to choices by calling functions on the notebook. This is done using an idiom that is rather peculiar to Python; it is a lightweight version of the command pattern that we will discuss in *Chapter 10*, *Python Design Patterns I*. The choices entered by the user are strings. In the menu's `__init__` method, we create a dictionary that maps strings to functions on the menu object itself. Then, when the user makes a choice, we retrieve the object from the dictionary. The `action` variable actually refers to a specific method, and is called by appending empty brackets (since none of the methods require parameters) to the variable. Of course, the user might have entered an inappropriate choice, so we check if the action really exists before calling it.

Each of the various methods request user input and call appropriate methods on the `Notebook` object associated with it. For the `search` implementation, we notice that after we've filtered the notes, we need to show them to the user, so we make the `show_notes` function serve double duty; it accepts an optional `notes` parameter. If it's supplied, it displays only the filtered notes, but if it's not, it displays all notes. Since the `notes` parameter is optional, `show_notes` can still be called with no parameters as an empty menu item.

If we test this code, we'll find that modifying notes doesn't work. There are two bugs, namely:

- The notebook crashes when we enter a note ID that does not exist. We should never trust our users to enter correct data!
- Even if we enter a correct ID, it will crash because the note IDs are integers, but our menu is passing a string.

The latter bug can be solved by modifying the `Notebook` class's `_find_note` method to compare the values using strings instead of the integers stored in the note, as follows:

```
def _find_note(self, note_id):
    '''Locate the note with the given id.'''
    for note in self.notes:
        if str(note.id) == str(note_id):
            return note
    return None
```

We simply convert both the input (`note_id`) and the note's ID to strings before comparing them. We could also convert the input to an integer, but then we'd have trouble if the user had entered the letter "a" instead of the number "1".

The problem with users entering note IDs that don't exist can be fixed by changing the two `modify` methods on the notebook to check whether `_find_note` returned a note or not, like this:

```
def modify_memo(self, note_id, memo):
    '''Find the note with the given id and change its
    memo to the given value.'''
    note = self._find_note(note_id)
    if note:
        note.memo = memo
        return True
    return False
```

This method has been updated to return `True` or `False`, depending on whether a note has been found. The menu could use this return value to display an error if the user entered an invalid note. This code is a bit unwieldy though; it would look a bit better if it raised an exception instead. We'll cover those in *Chapter 4*, *Expecting the Unexpected*.

Exercises

Write some object-oriented code. The goal is to use the principles and syntax you learned in this chapter to ensure you can use it, instead of just reading about it. If you've been working on a Python project, go back over it and see if there are some objects you can create and add properties or methods to. If it's large, try dividing it into a few modules or even packages and play with the syntax.

If you don't have such a project, try starting a new one. It doesn't have to be something you intend to finish, just stub out some basic design parts. You don't need to fully implement everything, often just a `print("this method will do something")` is all you need to get the overall design in place. This is called **top-down design**, in which you work out the different interactions and describe how they should work before actually implementing what they do. The converse, **bottom-up design**, implements details first and then ties them all together. Both patterns are useful at different times, but for understanding object-oriented principles, a top-down workflow is more suitable.

If you're having trouble coming up with ideas, try writing a to-do application. (Hint: It would be similar to the design of the notebook application, but with extra date management methods.) It can keep track of things you want to do each day, and allow you to mark them as completed.

Now, try designing a bigger project. It doesn't have to actually do anything, but make sure you experiment with the package and module importing syntax. Add some functions in various modules and try importing them from other modules and packages. Use relative and absolute imports. See the difference, and try to imagine scenarios where you would want to use each one.

Summary

In this chapter, we learned how simple it is to create classes and assign properties and methods in Python. Unlike many languages, Python differentiates between a constructor and an initializer. It has a relaxed attitude toward access control. There are many different levels of scope, including packages, modules, classes, and functions. We understood the difference between relative and absolute imports, and how to manage third-party packages that don't come with Python.

In the next chapter, we'll learn how to share implementation using inheritance.

3
When Objects Are Alike

In the programming world, duplicate code is considered evil. We should not have multiple copies of the same, or similar, code in different places.

There are many ways to merge pieces of code or objects that have a similar functionality. In this chapter, we'll be covering the most famous object-oriented principle: inheritance. As discussed in *Chapter 1*, *Object-oriented Design*, inheritance allows us to create *is a* relationships between two or more classes, abstracting common logic into superclasses and managing specific details in the subclass. In particular, we'll be covering the Python syntax and principles for:

- Basic inheritance
- Inheriting from built-ins
- Multiple inheritance
- Polymorphism and duck typing

Basic inheritance

Technically, every class we create uses inheritance. All Python classes are subclasses of the special class named `object`. This class provides very little in terms of data and behaviors (the behaviors it does provide are all double-underscore methods intended for internal use only), but it does allow Python to treat all objects in the same way.

If we don't explicitly inherit from a different class, our classes will automatically inherit from `object`. However, we can openly state that our class derives from `object` using the following syntax:

```
class MySubClass(object):
    pass
```

This is inheritance! This example is, technically, no different from our very first example in *Chapter 2*, *Objects in Python*, since Python 3 automatically inherits from `object` if we don't explicitly provide a different superclass. A superclass, or parent class, is a class that is being inherited from. A subclass is a class that is inheriting from a superclass. In this case, the superclass is `object`, and `MySubClass` is the subclass. A subclass is also said to be derived from its parent class or that the subclass extends the parent.

As you've probably figured out from the example, inheritance requires a minimal amount of extra syntax over a basic class definition. Simply include the name of the parent class inside parentheses after the class name but before the colon terminating the class definition. This is all we have to do to tell Python that the new class should be derived from the given superclass.

How do we apply inheritance in practice? The simplest and most obvious use of inheritance is to add functionality to an existing class. Let's start with a simple contact manager that tracks the name and e-mail address of several people. The contact class is responsible for maintaining a list of all contacts in a class variable, and for initializing the name and address for an individual contact:

```
class Contact:
    all_contacts = []

    def __init__(self, name, email):
        self.name = name
        self.email = email
        Contact.all_contacts.append(self)
```

This example introduces us to class variables. The `all_contacts` list, because it is part of the class definition, is shared by all instances of this class. This means that there is only one `Contact.all_contacts` list, which we can access as `Contact.all_contacts`. Less obviously, we can also access it as `self.all_contacts` on any object instantiated from `Contact`. If the field can't be found on the object, then it will be found on the class and thus refer to the same single list.

Be careful with this syntax, for if you ever *set* the variable using `self.all_contacts`, you will actually be creating a **new** instance variable associated only with that object. The class variable will still be unchanged and accessible as `Contact.all_contacts`.

This is a simple class that allows us to track a couple pieces of data about each contact. But what if some of our contacts are also suppliers that we need to order supplies from? We could add an `order` method to the `Contact` class, but that would allow people to accidentally order things from contacts who are customers or family friends. Instead, let's create a new `Supplier` class that acts like our `Contact` class, but has an additional `order` method:

```
class Supplier(Contact):
    def order(self, order):
        print("If this were a real system we would send "
                "'{}' order to '{}'".format(order, self.name))
```

Now, if we test this class in our trusty interpreter, we see that all contacts, including suppliers, accept a name and e-mail address in their `__init__`, but only suppliers have a functional order method:

```
>>> c = Contact("Some Body", "somebody@example.net")
>>> s = Supplier("Sup Plier", "supplier@example.net")
>>> print(c.name, c.email, s.name, s.email)
Some Body somebody@example.net Sup Plier supplier@example.net
>>> c.all_contacts
[<__main__.Contact object at 0xb7375ecc>,
 <__main__.Supplier object at 0xb7375f8c>]
>>> c.order("I need pliers")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Contact' object has no attribute 'order'
>>> s.order("I need pliers")
If this were a real system we would send 'I need pliers' order to
'Sup Plier '
```

So, now our `Supplier` class can do everything a contact can do (including adding itself to the list of `all_contacts`) and all the special things it needs to handle as a supplier. This is the beauty of inheritance.

Extending built-ins

One interesting use of this kind of inheritance is adding functionality to built-in classes. In the `Contact` class seen earlier, we are adding contacts to a list of all contacts. What if we also wanted to search that list by name? Well, we could add a method on the `Contact` class to search it, but it feels like this method actually belongs to the list itself. We can do this using inheritance:

```
class ContactList(list):
    def search(self, name):
        '''Return all contacts that contain the search value
        in their name.'''
        matching_contacts = []
        for contact in self:
            if name in contact.name:
                matching_contacts.append(contact)
        return matching_contacts

class Contact:
    all_contacts = ContactList()

    def __init__(self, name, email):
        self.name = name
        self.email = email
        self.all_contacts.append(self)
```

Instead of instantiating a normal list as our class variable, we create a new `ContactList` class that extends the built-in `list`. Then, we instantiate this subclass as our `all_contacts` list. We can test the new search functionality as follows:

```
>>> c1 = Contact("John A", "johna@example.net")
>>> c2 = Contact("John B", "johnb@example.net")
>>> c3 = Contact("Jenna C", "jennac@example.net")
>>> [c.name for c in Contact.all_contacts.search('John')]
['John A', 'John B']
```

Are you wondering how we changed the built-in syntax `[]` into something we can inherit from? Creating an empty list with `[]` is actually a shorthand for creating an empty list using `list()`; the two syntaxes behave identically:

```
>>> [] == list()
True
```

In reality, the `[]` syntax is actually so-called **syntax sugar** that calls the `list()` constructor under the hood. The `list` data type is a class that we can extend. In fact, the list itself extends the `object` class:

```
>>> isinstance([], object)
True
```

As a second example, we can extend the `dict` class, which is, similar to the list, the class that is constructed when using the `{}` syntax shorthand:

```
class LongNameDict(dict):
    def longest_key(self):
        longest = None
        for key in self:
            if not longest or len(key) > len(longest):
                longest = key
        return longest
```

This is easy to test in the interactive interpreter:

```
>>> longkeys = LongNameDict()
>>> longkeys['hello'] = 1
>>> longkeys['longest yet'] = 5
>>> longkeys['hello2'] = 'world'
>>> longkeys.longest_key()
'longest yet'
```

Most built-in types can be similarly extended. Commonly extended built-ins are `object`, `list`, `set`, `dict`, `file`, and `str`. Numerical types such as `int` and `float` are also occasionally inherited from.

Overriding and super

So, inheritance is great for *adding* new behavior to existing classes, but what about *changing* behavior? Our `contact` class allows only a name and an e-mail address. This may be sufficient for most contacts, but what if we want to add a phone number for our close friends?

As we saw in *Chapter 2, Objects in Python*, we can do this easily by just setting a `phone` attribute on the contact after it is constructed. But if we want to make this third variable available on initialization, we have to override `__init__`. Overriding means altering or replacing a method of the superclass with a new method (with the same name) in the subclass. No special syntax is needed to do this; the subclass's newly created method is automatically called instead of the superclass's method. For example:

```
class Friend(Contact):
    def __init__(self, name, email, phone):
        self.name = name
        self.email = email
        self.phone = phone
```

Any method can be overridden, not just `__init__`. Before we go on, however, we need to address some problems in this example. Our `Contact` and `Friend` classes have duplicate code to set up the `name` and `email` properties; this can make code maintenance complicated as we have to update the code in two or more places. More alarmingly, our `Friend` class is neglecting to add itself to the `all_contacts` list we have created on the `Contact` class.

What we really need is a way to execute the original `__init__` method on the `Contact` class. This is what the `super` function does; it returns the object as an instance of the parent class, allowing us to call the parent method directly:

```
class Friend(Contact):
    def __init__(self, name, email, phone):
        super().__init__(name, email)
        self.phone = phone
```

This example first gets the instance of the parent object using `super`, and calls `__init__` on that object, passing in the expected arguments. It then does its own initialization, namely, setting the `phone` attribute.

Note that the `super()` syntax does not work in older versions of Python. Like the [] and {} syntaxes for lists and dictionaries, it is a shorthand for a more complicated construct. We'll learn more about this shortly when we discuss multiple inheritance, but know for now that in Python 2, you would have to call `super(EmailContact, self).__init__()`. Specifically notice that the first argument is the name of the child class, not the name as the parent class you want to call, as some might expect. Also, remember the class comes before the object. I always forget the order, so the new syntax in Python 3 has saved me hours of having to look it up.

A `super()` call can be made inside any method, not just `__init__`. This means all methods can be modified via overriding and calls to `super`. The call to `super` can also be made at any point in the method; we don't have to make the call as the first line in the method. For example, we may need to manipulate or validate incoming parameters before forwarding them to the superclass.

Multiple inheritance

Multiple inheritance is a touchy subject. In principle, it's very simple: a subclass that inherits from more than one parent class is able to access functionality from both of them. In practice, this is less useful than it sounds and many expert programmers recommend against using it.

As a rule of thumb, if you think you need multiple inheritance, you're probably wrong, but if you know you need it, you're probably right.

The simplest and most useful form of multiple inheritance is called a **mixin**. A mixin is generally a superclass that is not meant to exist on its own, but is meant to be inherited by some other class to provide extra functionality. For example, let's say we wanted to add functionality to our `Contact` class that allows sending an e-mail to `self.email`. Sending e-mail is a common task that we might want to use on many other classes. So, we can write a simple mixin class to do the e-mailing for us:

```
class MailSender:
    def send_mail(self, message):
        print("Sending mail to " + self.email)
        # Add e-mail logic here
```

For brevity, we won't include the actual e-mail logic here; if you're interested in studying how it's done, see the `smtplib` module in the Python standard library.

This class doesn't do anything special (in fact, it can barely function as a standalone class), but it does allow us to define a new class that describes both a `Contact` and a `MailSender`, using multiple inheritance:

```
class EmailableContact(Contact, MailSender):
    pass
```

The syntax for multiple inheritance looks like a parameter list in the class definition. Instead of including one base class inside the parentheses, we include two (or more), separated by a comma. We can test this new hybrid to see the mixin at work:

```
>>> e = EmailableContact("John Smith", "jsmith@example.net")
```

```
>>> Contact.all_contacts
[<__main__.EmailableContact object at 0xb7205fac>]
>>> e.send_mail("Hello, test e-mail here")
Sending mail to jsmith@example.net
```

The `Contact` initializer is still adding the new contact to the `all_contacts` list, and the mixin is able to send mail to `self.email` so we know everything is working.

This wasn't so hard, and you're probably wondering what the dire warnings about multiple inheritance are. We'll get into the complexities in a minute, but let's consider some other options we had, rather than using a mixin here:

- We could have used single inheritance and added the `send_mail` function to the subclass. The disadvantage here is that the e-mail functionality then has to be duplicated for any other classes that need e-mail.
- We can create a standalone Python function for sending an e-mail, and just call that function with the correct e-mail address supplied as a parameter when the e-mail needs to be sent.
- We could have explored a few ways of using composition instead of inheritance. For example, `EmailableContact` could have a `MailSender` object instead of inheriting from it.
- We could monkey-patch (we'll briefly cover monkey-patching in *Chapter 7*, *Python Object-oriented Shortcuts*) the `Contact` class to have a `send_mail` method after the class has been created. This is done by defining a function that accepts the `self` argument, and setting it as an attribute on an existing class.

Multiple inheritance works all right when mixing methods from different classes, but it gets very messy when we have to call methods on the superclass. There are multiple superclasses. How do we know which one to call? How do we know what order to call them in?

Let's explore these questions by adding a home address to our `Friend` class. There are a few approaches we might take. An address is a collection of strings representing the street, city, country, and other related details of the contact. We could pass each of these strings as a parameter into the `Friend` class's `__init__` method. We could also store these strings in a tuple or dictionary and pass them into `__init__` as a single argument. This is probably the best course of action if there are no methods that need to be added to the address.

Another option would be to create a new `Address` class to hold those strings together, and then pass an instance of this class into the `__init__` method of our `Friend` class. The advantage of this solution is that we can add behavior (say, a method to give directions or to print a map) to the data instead of just storing it statically. This is an example of composition, as we discussed in *Chapter 1, Object-oriented Design*. The "has a" relationship of composition is a perfectly viable solution to this problem and allows us to reuse `Address` classes in other entities such as buildings, businesses, or organizations.

However, inheritance is also a viable solution, and that's what we want to explore. Let's add a new class that holds an address. We'll call this new class "AddressHolder" instead of "Address" because inheritance defines an *is a* relationship. It is not correct to say a "Friend" is an "Address" , but since a friend can have an "Address" , we can argue that a "Friend" is an "AddressHolder". Later, we could create other entities (companies, buildings) that also hold addresses. Here's our `AddressHolder` class:

```
class AddressHolder:
    def __init__(self, street, city, state, code):
        self.street = street
        self.city = city
        self.state = state
        self.code = code
```

Very simple; we just take all the data and toss it into instance variables upon initialization.

The diamond problem

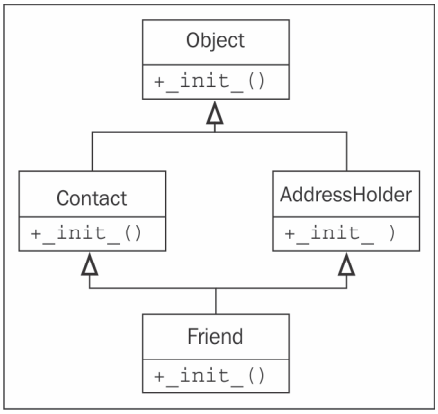
We can use multiple inheritance to add this new class as a parent of our existing `Friend` class. The tricky part is that we now have two parent `__init__` methods both of which need to be initialized. And they need to be initialized with different arguments. How do we do this? Well, we could start with a naive approach:

```
class Friend(Contact, AddressHolder):
    def __init__(
        self, name, email, phone,street, city, state, code):
        Contact.__init__(self, name, email)
        AddressHolder.__init__(self, street, city, state, code)
        self.phone = phone
```

In this example, we directly call the `__init__` function on each of the superclasses and explicitly pass the `self` argument. This example technically works; we can access the different variables directly on the class. But there are a few problems.

First, it is possible for a superclass to go uninitialized if we neglect to explicitly call the initializer. That wouldn't break this example, but it could cause hard-to-debug program crashes in common scenarios. Imagine trying to insert data into a database that has not been connected to, for example.

Second, and more sinister, is the possibility of a superclass being called multiple times because of the organization of the class hierarchy. Look at this inheritance diagram:

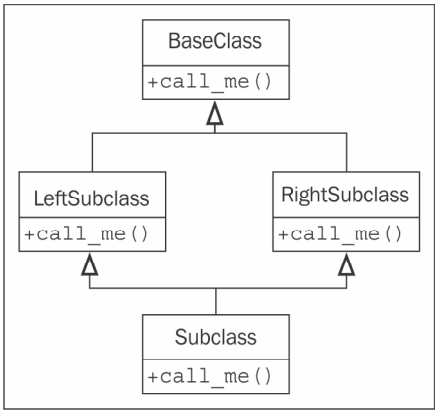


The `__init__` method from the `Friend` class first calls `__init__` on `Contact`, which implicitly initializes the `object` superclass (remember, all classes derive from `object`). `Friend` then calls `__init__` on `AddressHolder`, which implicitly initializes the `object` superclass *again*. This means the parent class has been set up twice. With the `object` class, that's relatively harmless, but in some situations, it could spell disaster. Imagine trying to connect to a database twice for every request!

The base class should only be called once. Once, yes, but when? Do we call `Friend`, then `Contact`, then `Object`, then `AddressHolder`? Or `Friend`, then `Contact`, then `AddressHolder`, then `Object`?

The order in which methods can be called can be adapted on the fly by modifying the `__mro__` (**Method Resolution Order**) attribute on the class. This is beyond the scope of this book. If you think you need to understand it, I recommend *Expert Python Programming*, *Tarek Ziadé*, *Packt Publishing*, or read the original documentation on the topic at `http://www.python.org/download/releases/2.3/mro/`.

Let's look at a second contrived example that illustrates this problem more clearly. Here we have a base class that has a method named `call_me`. Two subclasses override that method, and then another subclass extends both of these using multiple inheritance. This is called diamond inheritance because of the diamond shape of the class diagram:



Let's convert this diagram to code; this example shows when the methods are called:

```
class BaseClass:
    num_base_calls = 0
    def call_me(self):
        print("Calling method on Base Class")
        self.num_base_calls += 1

class LeftSubclass(BaseClass):
    num_left_calls = 0
    def call_me(self):
        BaseClass.call_me(self)
        print("Calling method on Left Subclass")
        self.num_left_calls += 1

class RightSubclass(BaseClass):
    num_right_calls = 0
    def call_me(self):
        BaseClass.call_me(self)
        print("Calling method on Right Subclass")
        self.num_right_calls += 1

class Subclass(LeftSubclass, RightSubclass):
```

```
        num_sub_calls = 0
        def call_me(self):
            LeftSubclass.call_me(self)
            RightSubclass.call_me(self)
            print("Calling method on Subclass")
            self.num_sub_calls += 1
```

This example simply ensures that each overridden `call_me` method directly calls the parent method with the same name. It lets us know each time a method is called by printing the information to the screen. It also updates a static variable on the class to show how many times it has been called. If we instantiate one `Subclass` object and call the method on it once, we get this output:

```
>>> s = Subclass()
>>> s.call_me()
Calling method on Base Class
Calling method on Left Subclass
Calling method on Base Class
Calling method on Right Subclass
Calling method on Subclass
>>> print(
... s.num_sub_calls,
... s.num_left_calls,
... s.num_right_calls,
... s.num_base_calls)
1 1 1 2
```

Thus we can clearly see the base class's `call_me` method being called twice. This could lead to some insidious bugs if that method is doing actual work—like depositing into a bank account—twice.

The thing to keep in mind with multiple inheritance is that we only want to call the "next" method in the class hierarchy, not the "parent" method. In fact, that next method may not be on a parent or ancestor of the current class. The `super` keyword comes to our rescue once again. Indeed, `super` was originally developed to make complicated forms of multiple inheritance possible. Here is the same code written using `super`:

```
class BaseClass:
    num_base_calls = 0
    def call_me(self):
```

```
            print("Calling method on Base Class")
            self.num_base_calls += 1

    class LeftSubclass(BaseClass):
        num_left_calls = 0
        def call_me(self):
            super().call_me()
            print("Calling method on Left Subclass")
            self.num_left_calls += 1

    class RightSubclass(BaseClass):
        num_right_calls = 0
        def call_me(self):
            super().call_me()
            print("Calling method on Right Subclass")
            self.num_right_calls += 1

    class Subclass(LeftSubclass, RightSubclass):
        num_sub_calls = 0
        def call_me(self):
            super().call_me()
            print("Calling method on Subclass")
            self.num_sub_calls += 1
```

The change is pretty minor; we simply replaced the naive direct calls with calls to `super()`, although the bottom subclass only calls `super` once rather than having to make the calls for both the left and right. The change is simple enough, but look at the difference when we execute it:

```
>>> s = Subclass()
>>> s.call_me()
Calling method on Base Class
Calling method on Right Subclass
Calling method on Left Subclass
Calling method on Subclass
>>> print(s.num_sub_calls, s.num_left_calls, s.num_right_calls,
s.num_base_calls)
1 1 1 1
```

Looks good, our base method is only being called once. But what is `super()` actually doing here? Since the `print` statements are executed after the `super` calls, the printed output is in the order each method is actually executed. Let's look at the output from back to front to see who is calling what.

First, `call_me` of `Subclass` calls `super().call_me()`, which happens to refer to `LeftSubclass.call_me()`. The `LeftSubclass.call_me()` method then calls `super().call_me()`, but in this case, `super()` is referring to `RightSubclass.call_me()`.

Pay particular attention to this: the `super` call is *not* calling the method on the superclass of `LeftSubclass` (which is `BaseClass`). Rather, it is calling `RightSubclass`, even though it is not a direct parent of `LeftSubclass`! This is the *next* method, not the parent method. `RightSubclass` then calls `BaseClass` and the `super` calls have ensured each method in the class hierarchy is executed once.

Different sets of arguments

This is going to make things complicated as we return to our `Friend` multiple inheritance example. In the `__init__` method for `Friend`, we were originally calling `__init__` for both parent classes, *with different sets of arguments*:

```
Contact.__init__(self, name, email)
AddressHolder.__init__(self, street, city, state, code)
```

How can we manage different sets of arguments when using `super`? We don't necessarily know which class `super` is going to try to initialize first. Even if we did, we need a way to pass the "extra" arguments so that subsequent calls to `super`, on other subclasses, receive the right arguments.

Specifically, if the first call to `super` passes the `name` and `email` arguments to `Contact.__init__`, and `Contact.__init__` then calls `super`, it needs to be able to pass the address-related arguments to the "next" method, which is `AddressHolder.__init__`.

This is a problem whenever we want to call superclass methods with the same name, but with different sets of arguments. Most often, the only time you would want to call a superclass with a completely different set of arguments is in `__init__`, as we're doing here. Even with regular methods, though, we may want to add optional parameters that only make sense to one subclass or set of subclasses.

Sadly, the only way to solve this problem is to plan for it from the beginning. We have to design our base class parameter lists to accept keyword arguments for any parameters that are not required by every subclass implementation. Finally, we must ensure the method freely accepts unexpected arguments and passes them on to its `super` call, in case they are necessary to later methods in the inheritance order.

Python's function parameter syntax provides all the tools we need to do this, but it makes the overall code look cumbersome. Have a look at the proper version of the `Friend` multiple inheritance code:

```
class Contact:
    all_contacts = []

    def __init__(self, name='', email='', **kwargs):
        super().__init__(**kwargs)
        self.name = name
        self.email = email
        self.all_contacts.append(self)

class AddressHolder:
    def __init__(self, street='', city='', state='', code='',
            **kwargs):
        super().__init__(**kwargs)
        self.street = street
        self.city = city
        self.state = state
        self.code = code

class Friend(Contact, AddressHolder):
    def __init__(self, phone='', **kwargs):
        super().__init__(**kwargs)
        self.phone = phone
```

We've changed all arguments to keyword arguments by giving them an empty string as a default value. We've also ensured that a `**kwargs` parameter is included to capture any additional parameters that our particular method doesn't know what to do with. It passes these parameters up to the next class with the `super` call.

If you aren't familiar with the `**kwargs` syntax, it basically collects any keyword arguments passed into the method that were not explicitly listed in the parameter list. These arguments are stored in a dictionary named `kwargs` (we can call the variable whatever we like, but convention suggests `kw`, or `kwargs`). When we call a different method (for example, `super().__init__`) with a `**kwargs` syntax, it unpacks the dictionary and passes the results to the method as normal keyword arguments. We'll cover this in detail in *Chapter 7, Python Object-oriented Shortcuts*.

The previous example does what it is supposed to do. But it's starting to look messy, and it has become difficult to answer the question, *What arguments do we need to pass into* `Friend.__init__`? This is the foremost question for anyone planning to use the class, so a docstring should be added to the method to explain what is happening.

Further, even this implementation is insufficient if we want to *reuse* variables in parent classes. When we pass the `**kwargs` variable to `super`, the dictionary does not include any of the variables that were included as explicit keyword arguments. For example, in `Friend.__init__`, the call to `super` does not have `phone` in the `kwargs` dictionary. If any of the other classes need the `phone` parameter, we need to ensure it is in the dictionary that is passed. Worse, if we forget to do this, it will be tough to debug because the superclass will not complain, but will simply assign the default value (in this case, an empty string) to the variable.

There are a few ways to ensure that the variable is passed upwards. Assume the `Contact` class does, for some reason, need to be initialized with a `phone` parameter, and the `Friend` class will also need access to it. We can do any of the following:

- Don't include `phone` as an explicit keyword argument. Instead, leave it in the `kwargs` dictionary. `Friend` can look it up using the syntax `kwargs['phone']`. When it passes `**kwargs` to the `super` call, `phone` will still be in the dictionary.
- Make `phone` an explicit keyword argument but update the `kwargs` dictionary before passing it to `super`, using the standard dictionary syntax `kwargs['phone'] = phone`.
- Make `phone` an explicit keyword argument, but update the `kwargs` dictionary using the `kwargs.update` method. This is useful if you have several arguments to update. You can create the dictionary passed into `update` using either the `dict(phone=phone)` constructor, or the dictionary syntax `{'phone': phone}`.
- Make `phone` an explicit keyword argument, but pass it to the super call explicitly with the syntax `super().__init__(phone=phone, **kwargs)`.

We have covered many of the caveats involved with multiple inheritance in Python. When we need to account for all the possible situations, we have to plan for them and our code will get messy. Basic multiple inheritance can be handy but, in many cases, we may want to choose a more transparent way of combining two disparate classes, usually using composition or one of the design patterns we'll be covering in *Chapter 10*, *Python Design Patterns I* and *Chapter 11*, *Python Design Patterns II*.

Polymorphism

We were introduced to polymorphism in *Chapter 1*, *Object-oriented Design*. It is a fancy name describing a simple concept: different behaviors happen depending on which subclass is being used, without having to explicitly know what the subclass actually is. As an example, imagine a program that plays audio files. A media player might need to load an `AudioFile` object and then `play` it. We'd put a `play()` method on the object, which is responsible for decompressing or extracting the audio and routing it to the sound card and speakers. The act of playing an `AudioFile` could feasibly be as simple as:

```
audio_file.play()
```

However, the process of decompressing and extracting an audio file is very different for different types of files. The `.wav` files are stored uncompressed, while `.mp3`, `.wma`, and `.ogg` files all have totally different compression algorithms.

We can use inheritance with polymorphism to simplify the design. Each type of file can be represented by a different subclass of `AudioFile`, for example, `WavFile`, `MP3File`. Each of these would have a `play()` method, but that method would be implemented differently for each file to ensure the correct extraction procedure is followed. The media player object would never need to know which subclass of `AudioFile` it is referring to; it just calls `play()` and polymorphically lets the object take care of the actual details of playing. Let's look at a quick skeleton showing how this might look:

```
class AudioFile:
    def __init__(self, filename):
        if not filename.endswith(self.ext):
            raise Exception("Invalid file format")

        self.filename = filename

class MP3File(AudioFile):
    ext = "mp3"
    def play(self):
        print("playing {} as mp3".format(self.filename))

class WavFile(AudioFile):
    ext = "wav"
    def play(self):
```

```
            print("playing {} as wav".format(self.filename))

    class OggFile(AudioFile):
        ext = "ogg"
        def play(self):
            print("playing {} as ogg".format(self.filename))
```

All audio files check to ensure that a valid extension was given upon initialization. But did you notice how the `__init__` method in the parent class is able to access the `ext` class variable from different subclasses? That's polymorphism at work. If the filename doesn't end with the correct name, it raises an exception (exceptions will be covered in detail in the next chapter). The fact that `AudioFile` doesn't actually store a reference to the `ext` variable doesn't stop it from being able to access it on the subclass.

In addition, each subclass of `AudioFile` implements `play()` in a different way (this example doesn't actually play the music; audio compression algorithms really deserve a separate book!). This is also polymorphism in action. The media player can use the exact same code to play a file, no matter what type it is; it doesn't care what subclass of `AudioFile` it is looking at. The details of decompressing the audio file are *encapsulated*. If we test this example, it works as we would hope:

```
>>> ogg = OggFile("myfile.ogg")
>>> ogg.play()
playing myfile.ogg as ogg
>>> mp3 = MP3File("myfile.mp3")
>>> mp3.play()
playing myfile.mp3 as mp3
>>> not_an_mp3 = MP3File("myfile.ogg")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "polymorphic_audio.py", line 4, in __init__
    raise Exception("Invalid file format")
Exception: Invalid file format
```

See how `AudioFile.__init__` is able to check the file type without actually knowing what subclass it is referring to?

Polymorphism is actually one of the coolest things about object-oriented programming, and it makes some programming designs obvious that weren't possible in earlier paradigms. However, Python makes polymorphism less cool because of duck typing. Duck typing in Python allows us to use *any* object that provides the required behavior without forcing it to be a subclass. The dynamic nature of Python makes this trivial. The following example does not extend `AudioFile`, but it can be interacted with in Python using the exact same interface:

```
class FlacFile:
    def __init__(self, filename):
        if not filename.endswith(".flac"):
            raise Exception("Invalid file format")

        self.filename = filename

    def play(self):
        print("playing {} as flac".format(self.filename))
```

Our media player can play this object just as easily as one that extends `AudioFile`.

Polymorphism is one of the most important reasons to use inheritance in many object-oriented contexts. Because any objects that supply the correct interface can be used interchangeably in Python, it reduces the need for polymorphic common superclasses. Inheritance can still be useful for sharing code but, if all that is being shared is the public interface, duck typing is all that is required. This reduced need for inheritance also reduces the need for multiple inheritance; often, when multiple inheritance appears to be a valid solution, we can just use duck typing to mimic one of the multiple superclasses.

Of course, just because an object satisfies a particular interface (by providing required methods or attributes) does not mean it will simply work in all situations. It has to fulfill that interface in a way that makes sense in the overall system. Just because an object provides a `play()` method does not mean it will automatically work with a media player. For example, our chess AI object from *Chapter 1*, *Object-oriented Design*, may have a `play()` method that moves a chess piece. Even though it satisfies the interface, this class would likely break in spectacular ways if we tried to plug it into a media player!

Another useful feature of duck typing is that the duck-typed object only needs to provide those methods and attributes that are actually being accessed. For example, if we needed to create a fake file object to read data from, we can create a new object that has a `read()` method; we don't have to override the `write` method if the code that is going to interact with the object will only be reading from the file. More succinctly, duck typing doesn't need to provide the entire interface of an object that is available, it only needs to fulfill the interface that is actually accessed.

Abstract base classes

While duck typing is useful, it is not always easy to tell in advance if a class is going to fulfill the protocol you require. Therefore, Python introduced the idea of abstract base classes. **Abstract base classes**, or **ABCs**, define a set of methods and properties that a class must implement in order to be considered a duck-type instance of that class. The class can extend the abstract base class itself in order to be used as an instance of that class, but it must supply all the appropriate methods.

In practice, it's rarely necessary to create new abstract base classes, but we may find occasions to implement instances of existing ABCs. We'll cover implementing ABCs first, and then briefly see how to create your own if you should ever need to.

Using an abstract base class

Most of the abstract base classes that exist in the Python Standard Library live in the `collections` module. One of the simplest ones is the `Container` class. Let's inspect it in the Python interpreter to see what methods this class requires:

```
>>> from collections import Container
>>> Container.__abstractmethods__
frozenset(['__contains__'])
```

So, the `Container` class has exactly one abstract method that needs to be implemented, `__contains__`. You can issue `help(Container.__contains__)` to see what the function signature should look like:

```
Help on method __contains__ in module _abcoll:
__contains__(self, x) unbound _abcoll.Container method
```

So, we see that `__contains__` needs to take a single argument. Unfortunately, the help file doesn't tell us much about what that argument should be, but it's pretty obvious from the name of the ABC and the single method it implements that this argument is the value the user is checking to see if the container holds.

This method is implemented by `list`, `str`, and `dict` to indicate whether or not a given value is in that data structure. However, we can also define a silly container that tells us whether a given value is in the set of odd integers:

```
class OddContainer:
    def __contains__(self, x):
        if not isinstance(x, int) or not x % 2:
            return False
        return True
```

Now, we can instantiate an `OddContainer` object and determine that, even though we did not extend `Container`, the class *is a* `Container` object:

```
>>> from collections import Container
>>> odd_container = OddContainer()
>>> isinstance(odd_container, Container)
True
>>> issubclass(OddContainer, Container)
True
```

And that is why duck typing is way more awesome than classical polymorphism. We can create *is a* relationships without the overhead of using inheritance (or worse, multiple inheritance).

The interesting thing about the `Container` ABC is that any class that implements it gets to use the `in` keyword for free. In fact, `in` is just syntax sugar that delegates to the `__contains__` method. Any class that has a `__contains__` method is a `Container` and can therefore be queried by the `in` keyword, for example:

```
>>> 1 in odd_container
True
>>> 2 in odd_container
False
>>> 3 in odd_container
True
>>> "a string" in odd_container
False
```

Creating an abstract base class

As we saw earlier, it's not necessary to have an abstract base class to enable duck typing. However, imagine we were creating a media player with third-party plugins. It is advisable to create an abstract base class in this case to document what API the third-party plugins should provide. The `abc` module provides the tools you need to do this, but I'll warn you in advance, this requires some of Python's most arcane concepts:

```
import abc

class MediaLoader(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def play(self):
        pass

    @abc.abstractproperty
    def ext(self):
```

```
            pass

        @classmethod
        def __subclasshook__(cls, C):
            if cls is MediaLoader:
                attrs = set(dir(C))
                if set(cls.__abstractmethods__) <= attrs:
                    return True

            return NotImplemented
```

This is a complicated example that includes several Python features that won't be explained until later in this book. It is included here for completeness, but you don't need to understand all of it to get the gist of how to create your own ABC.

The first weird thing is the `metaclass` keyword argument that is passed into the class where you would normally see the list of parent classes. This is a rarely used construct from the mystic art of metaclass programming. We won't be covering metaclasses in this book, so all you need to know is that by assigning the `ABCMeta` metaclass, you are giving your class superpower (or at least superclass) abilities.

Next, we see the `@abc.abstractmethod` and `@abc.abstractproperty` constructs. These are Python decorators. We'll discuss those in *Chapter 5*, *When to Use Object-oriented Programming*. For now, just know that by marking a method or property as being abstract, you are stating that any subclass of this class must implement that method or supply that property in order to be considered a proper member of the class.

See what happens if you implement subclasses that do or don't supply those properties:

```
>>> class Wav(MediaLoader):
...     pass
...
>>> x = Wav()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Wav with abstract methods
ext, play
>>> class Ogg(MediaLoader):
...     ext = '.ogg'
...     def play(self):
...         pass
...
>>> o = Ogg()
```

Since the `Wav` class fails to implement the abstract attributes, it is not possible to instantiate that class. The class is still a legal abstract class, but you'd have to subclass it to actually do anything. The `Ogg` class supplies both attributes, so it instantiates cleanly.

Going back to the `MediaLoader` ABC, let's dissect that `__subclasshook__` method. It is basically saying that any class that supplies concrete implementations of all the abstract attributes of this ABC should be considered a subclass of `MediaLoader`, even if it doesn't actually inherit from the `MediaLoader` class.

More common object-oriented languages have a clear separation between the interface and the implementation of a class. For example, some languages provide an explicit `interface` keyword that allows us to define the methods that a class must have without any implementation. In such an environment, an abstract class is one that provides both an interface and a concrete implementation of some but not all methods. Any class can explicitly state that it implements a given interface.

Python's ABCs help to supply the functionality of interfaces without compromising on the benefits of duck typing.

Demystifying the magic

You can copy and paste the subclass code without understanding it if you want to make abstract classes that fulfill this particular contract. We'll cover most of the unusual syntaxes throughout the book, but let's go over it line by line to get an overview.

```
@classmethod
```

This decorator marks the method as a class method. It essentially says that the method can be called on a class instead of an instantiated object:

```
def __subclasshook__(cls, C):
```

This defines the `__subclasshook__` class method. This special method is called by the Python interpreter to answer the question, *Is the class `C` a subclass of this class?*

```
    if cls is MediaLoader:
```

We check to see if the method was called specifically on this class, rather than, say a subclass of this class. This prevents, for example, the `Wav` class from being thought of as a parent class of the `Ogg` class:

```
        attrs = set(dir(C))
```

All this line does is get the set of methods and properties that the class has, including any parent classes in its class hierarchy:

```
            if set(cls.__abstractmethods__) <= attrs:
```

This line uses set notation to see whether the set of abstract methods in this class have been supplied in the candidate class. Note that it doesn't check to see whether the methods have been implemented, just if they are there. Thus, it's possible for a class to be a subclass and yet still be an abstract class itself.

```
                return True
```

If all the abstract methods have been supplied, then the candidate class is a subclass of this class and we return `True`. The method can legally return one of the three values: `True`, `False`, or `NotImplemented`. `True` and `False` indicate that the class is or is not definitively a subclass of this class:

```
        return NotImplemented
```

If any of the conditionals have not been met (that is, the class is not `MediaLoader` or not all abstract methods have been supplied), then return `NotImplemented`. This tells the Python machinery to use the default mechanism (does the candidate class explicitly extend this class?) for subclass detection.

In short, we can now define the `Ogg` class as a subclass of the `MediaLoader` class without actually extending the `MediaLoader` class:

```
>>> class Ogg():
...     ext = '.ogg'
...     def play(self):
...         print("this will play an ogg file")
...
>>> issubclass(Ogg, MediaLoader)
True
>>> isinstance(Ogg(), MediaLoader)
True
```

Case study

Let's try to tie everything we've learned together with a larger example. We'll be designing a simple real estate application that allows an agent to manage properties available for purchase or rent. There will be two types of properties: apartments and houses. The agent needs to be able to enter a few relevant details about new properties, list all currently available properties, and mark a property as sold or rented. For brevity, we won't worry about editing property details or reactivating a property after it is sold.

The project will allow the agent to interact with the objects using the Python interpreter prompt. In this world of graphical user interfaces and web applications, you might be wondering why we're creating such old-fashioned looking programs. Simply put, both windowed programs and web applications require a lot of overhead knowledge and boilerplate code to make them do what is required. If we were developing software using either of these paradigms, we'd get so lost in GUI programming or web programming that we'd lose sight of the object-oriented principles we're trying to master.

Luckily, most GUI and web frameworks utilize an object-oriented approach, and the principles we're studying now will help in understanding those systems in the future. We'll discuss them both briefly in *Chapter 13*, *Concurrency*, but complete details are far beyond the scope of a single book.

Looking at our requirements, it seems like there are quite a few nouns that might represent classes of objects in our system. Clearly, we'll need to represent a property. Houses and apartments may need separate classes. Rentals and purchases also seem to require separate representation. Since we're focusing on inheritance right now, we'll be looking at ways to share behavior using inheritance or multiple inheritance.

`House` and `Apartment` are both types of properties, so `Property` can be a superclass of those two classes. `Rental` and `Purchase` will need some extra thought; if we use inheritance, we'll need to have separate classes, for example, for `HouseRental` and `HousePurchase`, and use multiple inheritance to combine them. This feels a little clunky compared to a composition or association-based design, but let's run with it and see what we come up with.

Now then, what attributes might be associated with a `Property` class? Regardless of whether it is an apartment or a house, most people will want to know the square footage, number of bedrooms, and number of bathrooms. (There are numerous other attributes that might be modeled, but we'll keep it simple for our prototype.)

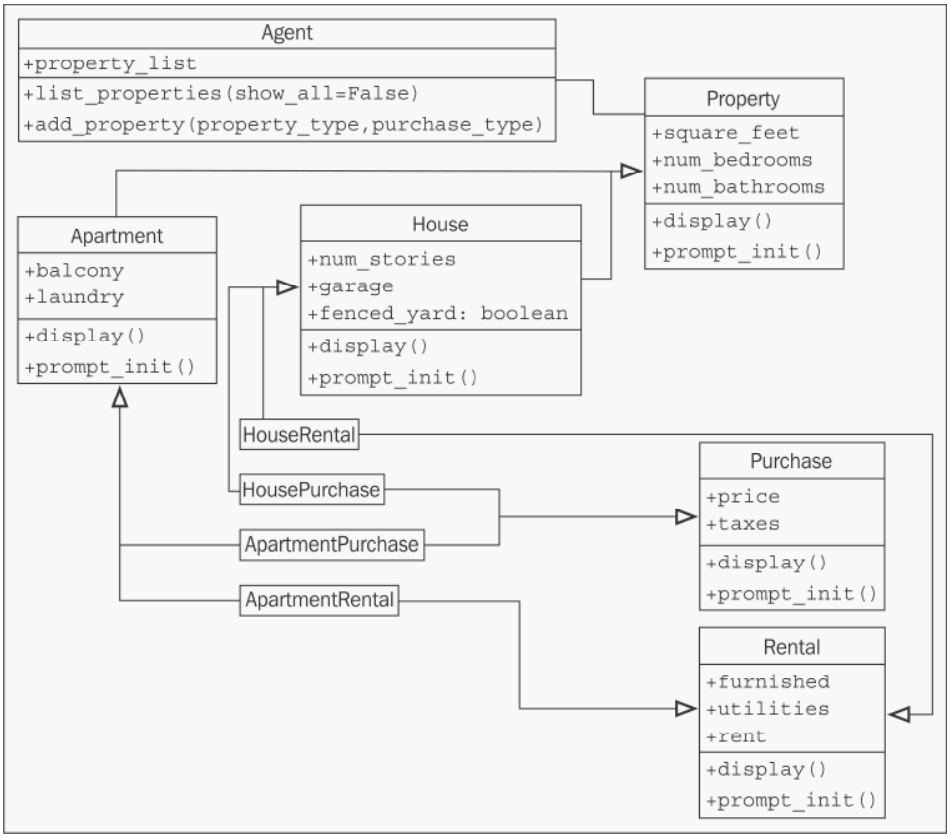
If the property is a house, it will want to advertise the number of stories, whether it has a garage (attached, detached, or none), and whether the yard is fenced. An apartment will want to indicate if it has a balcony, and if the laundry is ensuite, coin, or off-site.

Both property types will require a method to display the characteristics of that property. At the moment, no other behaviors are apparent.

Rental properties will need to store the rent per month, whether the property is furnished, and whether utilities are included, and if not, what they are estimated to be. Properties for purchase will need to store the purchase price and estimated annual property taxes. For our application, we'll only need to display this data, so we can get away with just adding a `display()` method similar to that used in the other classes.

Finally, we'll need an `Agent` object that holds a list of all properties, displays those properties, and allows us to create new ones. Creating properties will entail prompting the user for the relevant details for each property type. This could be done in the `Agent` object, but then `Agent` would need to know a lot of information about the types of properties. This is not taking advantage of polymorphism. Another alternative would be to put the prompts in the initializer or even a constructor for each class, but this would not allow the classes to be applied in a GUI or web application in the future. A better idea is to create a static method that does the prompting and returns a dictionary of the prompted parameters. Then, all the `Agent` has to do is prompt the user for the type of property and payment method, and ask the correct class to instantiate itself.

That's a lot of designing! The following class diagram may communicate our design decisions a little more clearly:



Wow, that's a lot of inheritance arrows! I don't think it would be possible to add another level of inheritance without crossing arrows. Multiple inheritance is a messy business, even at the design stage.

The trickiest aspects of these classes is going to be ensuring superclass methods get called in the inheritance hierarchy. Let's start with the `Property` implementation:

```
class Property:
    def __init__(self, square_feet='', beds='',
            baths='', **kwargs):
        super().__init__(**kwargs)
        self.square_feet = square_feet
        self.num_bedrooms = beds
        self.num_baths = baths

    def display(self):
        print("PROPERTY DETAILS")
        print("================")
        print("square footage: {}".format(self.square_feet))
        print("bedrooms: {}".format(self.num_bedrooms))
        print("bathrooms: {}".format(self.num_baths))
        print()

    def prompt_init():
        return dict(square_feet=input("Enter the square feet: "),
                beds=input("Enter number of bedrooms: "),
                baths=input("Enter number of baths: "))
    prompt_init = staticmethod(prompt_init)
```

This class is pretty straightforward. We've already added the extra `**kwargs` parameter to `__init__` because we know it's going to be used in a multiple inheritance situation. We've also included a call to `super().__init__` in case we are not the last call in the multiple inheritance chain. In this case, we're *consuming* the keyword arguments because we know they won't be needed at other levels of the inheritance hierarchy.

We see something new in the `prompt_init` method. This method is made into a static method immediately after it is initially created. Static methods are associated only with a class (something like class variables), rather than a specific object instance. Hence, they have no `self` argument. Because of this, the `super` keyword won't work (there is no parent object, only a parent class), so we simply call the static method on the parent class directly. This method uses the Python `dict` constructor to create a dictionary of values that can be passed into `__init__`. The value for each key is prompted with a call to `input`.

The `Apartment` class extends `Property`, and is similar in structure:

```
class Apartment(Property):
    valid_laundries = ("coin", "ensuite", "none")
    valid_balconies = ("yes", "no", "solarium")

    def __init__(self, balcony='', laundry='', **kwargs):
        super().__init__(**kwargs)
        self.balcony = balcony
        self.laundry = laundry

    def display(self):
        super().display()
        print("APARTMENT DETAILS")
        print("laundry: %s" % self.laundry)
        print("has balcony: %s" % self.balcony)

    def prompt_init():
        parent_init = Property.prompt_init()
        laundry = ''
        while laundry.lower() not in \
                Apartment.valid_laundries:
            laundry = input("What laundry facilities does "
                    "the property have? ({})".format(
                    ", ".join(Apartment.valid_laundries)))
        balcony = ''
        while balcony.lower() not in \
                Apartment.valid_balconies:
            balcony = input(
                "Does the property have a balcony? "
                "({})".format(
                ", ".join(Apartment.valid_balconies)))
        parent_init.update({
            "laundry": laundry,
            "balcony": balcony
        })
        return parent_init
    prompt_init = staticmethod(prompt_init)
```

The `display()` and `__init__()` methods call their respective parent class methods using `super()` to ensure the `Property` class is properly initialized.

The `prompt_init` static method is now getting dictionary values from the parent class, and then adding some additional values of its own. It calls the `dict.update` method to merge the new dictionary values into the first one. However, that `prompt_init` method is looking pretty ugly; it loops twice until the user enters a valid input using structurally similar code but different variables. It would be nice to extract this validation logic so we can maintain it in only one location; it will likely also be useful to later classes.

With all the talk on inheritance, we might think this is a good place to use a mixin. Instead, we have a chance to study a situation where inheritance is not the best solution. The method we want to create will be used in a static method. If we were to inherit from a class that provided validation functionality, the functionality would also have to be provided as a static method that did not access any instance variables on the class. If it doesn't access any instance variables, what's the point of making it a class at all? Why don't we just make this validation functionality a module-level function that accepts an input string and a list of valid answers, and leave it at that?

Let's explore what this validation function would look like:

```
def get_valid_input(input_string, valid_options):
    input_string += " ({}) ".format(", ".join(valid_options))
    response = input(input_string)
    while response.lower() not in valid_options:
        response = input(input_string)
    return response
```

We can test this function in the interpreter, independent of all the other classes we've been working on. This is a good sign, it means different pieces of our design are not tightly coupled to each other and can later be improved independently, without affecting other pieces of code.

```
>>> get_valid_input("what laundry?", ("coin", "ensuite", "none"))
what laundry? (coin, ensuite, none) hi
what laundry? (coin, ensuite, none) COIN
'COIN'
```

Now, let's quickly update our `Apartment.prompt_init` method to use this new function for validation:

```
        def prompt_init():
            parent_init = Property.prompt_init()
            laundry = get_valid_input(
```

```
                "What laundry facilities does "
                "the property have? ",
                Apartment.valid_laundries)
        balcony = get_valid_input(
            "Does the property have a balcony? ",
            Apartment.valid_balconies)
        parent_init.update({
            "laundry": laundry,
            "balcony": balcony
        })
        return parent_init
    prompt_init = staticmethod(prompt_init)
```

That's much easier to read (and maintain!) than our original version. Now we're ready to build the `House` class. This class has a parallel structure to `Apartment`, but refers to different prompts and variables:

```
class House(Property):
    valid_garage = ("attached", "detached", "none")
    valid_fenced = ("yes", "no")

    def __init__(self, num_stories='',
            garage='', fenced='', **kwargs):
        super().__init__(**kwargs)
        self.garage = garage
        self.fenced = fenced
        self.num_stories = num_stories

    def display(self):
        super().display()
        print("HOUSE DETAILS")
        print("# of stories: {}".format(self.num_stories))
        print("garage: {}".format(self.garage))
        print("fenced yard: {}".format(self.fenced))


    def prompt_init():
        parent_init = Property.prompt_init()
        fenced = get_valid_input("Is the yard fenced? ",
                    House.valid_fenced)
        garage = get_valid_input("Is there a garage? ",
                House.valid_garage)
```

```
        num_stories = input("How many stories? ")

        parent_init.update({
            "fenced": fenced,
            "garage": garage,
            "num_stories": num_stories
        })
        return parent_init
    prompt_init = staticmethod(prompt_init)
```

There's nothing new to explore here, so let's move on to the `Purchase` and `Rental` classes. In spite of having apparently different purposes, they are also similar in design to the ones we just discussed:

```
class Purchase:
    def __init__(self, price='', taxes='', **kwargs):
        super().__init__(**kwargs)
        self.price = price
        self.taxes = taxes

    def display(self):
        super().display()
        print("PURCHASE DETAILS")
        print("selling price: {}".format(self.price))
        print("estimated taxes: {}".format(self.taxes))

    def prompt_init():
        return dict(
            price=input("What is the selling price? "),
            taxes=input("What are the estimated taxes? "))
    prompt_init = staticmethod(prompt_init)

class Rental:
    def __init__(self, furnished='', utilities='',
            rent='', **kwargs):
        super().__init__(**kwargs)
        self.furnished = furnished
        self.rent = rent
        self.utilities = utilities

    def display(self):
        super().display()
        print("RENTAL DETAILS")
```

```
            print("rent: {}".format(self.rent))
            print("estimated utilities: {}".format(
                self.utilities))
            print("furnished: {}".format(self.furnished))

        def prompt_init():
            return dict(
                rent=input("What is the monthly rent? "),
                utilities=input(
                    "What are the estimated utilities? "),
                furnished = get_valid_input(
                    "Is the property furnished? ",
                        ("yes", "no")))
        prompt_init = staticmethod(prompt_init)
```

These two classes don't have a superclass (other than `object`), but we still call `super().__init__` because they are going to be combined with the other classes, and we don't know what order the `super` calls will be made in. The interface is similar to that used for `House` and `Apartment`, which is very useful when we combine the functionality of these four classes in separate subclasses. For example:

```
class HouseRental(Rental, House):
    def prompt_init():
        init = House.prompt_init()
        init.update(Rental.prompt_init())
        return init
    prompt_init = staticmethod(prompt_init)
```

This is slightly surprising, as the class on its own has neither an `__init__` nor `display` method! Because both parent classes appropriately call `super` in these methods, we only have to extend those classes and the classes will behave in the correct order. This is not the case with `prompt_init`, of course, since it is a static method that does not call `super`, so we implement this one explicitly. We should test this class to make sure it is behaving properly before we write the other three combinations:

```
>>> init = HouseRental.prompt_init()
Enter the square feet: 1
Enter number of bedrooms: 2
Enter number of baths: 3
Is the yard fenced?  (yes, no) no
Is there a garage?  (attached, detached, none) none
How many stories? 4
What is the monthly rent? 5
What are the estimated utilities? 6
```

```
Is the property furnished?  (yes, no) no
>>> house = HouseRental(**init)
>>> house.display()
PROPERTY DETAILS
================
square footage: 1
bedrooms: 2
bathrooms: 3

HOUSE DETAILS
# of stories: 4
garage: none
fenced yard: no

RENTAL DETAILS
rent: 5
estimated utilities: 6
furnished: no
```

It looks like it is working fine. The `prompt_init` method is prompting for initializers to all the super classes, and `display()` is also cooperatively calling all three superclasses.

The order of the inherited classes in the preceding example is important. If we had written `class HouseRental(House, Rental)` instead of `class HouseRental(Rental, House)`, `display()` would not have called `Rental.display()`! When `display` is called on our version of `HouseRental`, it refers to the `Rental` version of the method, which calls `super.display()` to get the `House` version, which again calls `super.display()` to get the property version. If we reversed it, `display` would refer to the `House` class's `display()`. When super is called, it calls the method on the `Property` parent class. But `Property` does not have a call to `super` in its `display` method. This means `Rental` class's `display` method would not be called! By placing the inheritance list in the order we did, we ensure that `Rental` calls `super`, which then takes care of the `House` side of the hierarchy. You might think we could have added a `super` call to `Property.display()`, but that will fail because the next superclass of `Property` is `object`, and `object` does not have a `display` method. Another way to fix this is to allow `Rental` and `Purchase` to extend the `Property` class instead of deriving directly from `object`. (Or we could modify the method resolution order dynamically, but that is beyond the scope of this book.)

Now that we have tested it, we are prepared to create the rest of our combined subclasses:

```
class ApartmentRental(Rental, Apartment):
    def prompt_init():
        init = Apartment.prompt_init()
        init.update(Rental.prompt_init())
        return init
    prompt_init = staticmethod(prompt_init)

class ApartmentPurchase(Purchase, Apartment):
    def prompt_init():
        init = Apartment.prompt_init()
        init.update(Purchase.prompt_init())
        return init
    prompt_init = staticmethod(prompt_init)

class HousePurchase(Purchase, House):
    def prompt_init():
        init = House.prompt_init()
        init.update(Purchase.prompt_init())
        return init
    prompt_init = staticmethod(prompt_init)
```

That should be the most intense designing out of our way! Now all we have to do is create the `Agent` class, which is responsible for creating new listings and displaying existing ones. Let's start with the simpler storing and listing of properties:

```
class Agent:
    def __init__(self):
        self.property_list = []

    def display_properties(self):
        for property in self.property_list:
            property.display()
```

Adding a property will require first querying the type of property and whether property is for purchase or rental. We can do this by displaying a simple menu. Once this has been determined, we can extract the correct subclass and prompt for all the details using the `prompt_init` hierarchy we've already developed. Sounds simple? It is. Let's start by adding a dictionary class variable to the `Agent` class:

```
    type_map = {
        ("house", "rental"): HouseRental,
        ("house", "purchase"): HousePurchase,
```

```
        ("apartment", "rental"): ApartmentRental,
        ("apartment", "purchase"): ApartmentPurchase
        }
```

That's some pretty funny looking code. This is a dictionary, where the keys are tuples of two distinct strings, and the values are class objects. Class objects? Yes, classes can be passed around, renamed, and stored in containers just like *normal* objects or primitive data types. With this simple dictionary, we can simply hijack our earlier `get_valid_input` method to ensure we get the correct dictionary keys and look up the appropriate class, like this:

```
    def add_property(self):
        property_type = get_valid_input(
                "What type of property? ",
                ("house", "apartment")).lower()
        payment_type = get_valid_input(
                "What payment type? ",
                ("purchase", "rental")).lower()

        PropertyClass = self.type_map[
            (property_type, payment_type)]
        init_args = PropertyClass.prompt_init()
        self.property_list.append(PropertyClass(**init_args))
```

This may look a bit funny too! We look up the class in the dictionary and store it in a variable named `PropertyClass`. We don't know exactly which class is available, but the class knows itself, so we can polymorphically call `prompt_init` to get a dictionary of values appropriate to pass into the constructor. Then we use the keyword argument syntax to convert the dictionary into arguments and construct the new object to load the correct data.

Now our user can use this `Agent` class to add and view lists of properties. It wouldn't take much work to add features to mark a property as available or unavailable or to edit and remove properties. Our prototype is now in a good enough state to take to a real estate `agent` and demonstrate its functionality. Here's how a demo session might work:

```
>>> agent = Agent()
>>> agent.add_property()
What type of property?  (house, apartment) house
What payment type?  (purchase, rental) rental
Enter the square feet: 900
Enter number of bedrooms: 2
Enter number of baths: one and a half
```

```
Is the yard fenced?  (yes, no) yes
Is there a garage?  (attached, detached, none) detached
How many stories? 1
What is the monthly rent? 1200
What are the estimated utilities? included
Is the property furnished?  (yes, no) no
>>> agent.add_property()
What type of property?  (house, apartment) apartment
What payment type?  (purchase, rental) purchase
Enter the square feet: 800
Enter number of bedrooms: 3
Enter number of baths: 2
What laundry facilities does the property have?  (coin, ensuite,
one) ensuite
Does the property have a balcony? (yes, no, solarium) yes
What is the selling price? $200,000
What are the estimated taxes? 1500
>>> agent.display_properties()
PROPERTY DETAILS
================
square footage: 900
bedrooms: 2
bathrooms: one and a half

HOUSE DETAILS
# of stories: 1
garage: detached
fenced yard: yes
RENTAL DETAILS
rent: 1200
estimated utilities: included
furnished: no
PROPERTY DETAILS
================
square footage: 800
bedrooms: 3
```

```
bathrooms: 2

APARTMENT DETAILS
laundry: ensuite
has balcony: yes
PURCHASE DETAILS
selling price: $200,000
estimated taxes: 1500
```

Exercises

Look around you at some of the physical objects in your workspace and see if you can describe them in an inheritance hierarchy. Humans have been dividing the world into taxonomies like this for centuries, so it shouldn't be difficult. Are there any non-obvious inheritance relationships between classes of objects? If you were to model these objects in a computer application, what properties and methods would they share? Which ones would have to be polymorphically overridden?
What properties would be completely different between them?

Now, write some code. No, not for the physical hierarchy; that's boring. Physical items have more properties than methods. Just think about a pet programming project you've wanted to tackle in the past year, but never got around to. For whatever problem you want to solve, try to think of some basic inheritance relationships. Then implement them. Make sure that you also pay attention to the sorts of relationships that you actually don't need to use inheritance for. Are there any places where you might want to use multiple inheritance? Are you sure? Can you see any place you would want to use a mixin? Try to knock together a quick prototype. It doesn't have to be useful or even partially working. You've seen how you can test code using `python -i` already; just write some code and test it in the interactive interpreter. If it works, write some more. If it doesn't, fix it!

Now, take a look at the real estate example. This turned out to be quite an effective use of multiple inheritance. I have to admit though, I had my doubts when I started the design. Have a look at the original problem and see if you can come up with another design to solve it that uses only single inheritance. How would you do it with abstract base classes? What about a design that doesn't use inheritance at all? Which do you think is the most elegant solution? Elegance is a primary goal in Python development, but each programmer has a different opinion as to what is the most elegant solution. Some people tend to think and understand problems most clearly using composition, while others find multiple inheritance to be the most useful model.

Finally, try adding some new features to the three designs. Whatever features strike your fancy are fine. I'd like to see a way to differentiate between available and unavailable properties, for starters. It's not of much use to me if it's already rented!

Which design is easiest to extend? Which is hardest? If somebody asked you why you thought this, would you be able to explain yourself?

Summary

We've gone from simple inheritance, one of the most useful tools in the object-oriented programmer's toolbox, all the way through to multiple inheritance, one of the most complicated. Inheritance can be used to add functionality to existing classes and built-ins using inheritance. Abstracting similar code into a parent class can help increase maintainability. Methods on parent classes can be called using `super` and argument lists must be formatted safely for these calls to work when using multiple inheritance.

In the next chapter, we'll cover the subtle art of handling exceptional circumstances.

4
Expecting the Unexpected

Programs are very fragile. It would be ideal if code always returned a valid result, but sometimes a valid result can't be calculated. For example, it's not possible to divide by zero, or to access the eighth item in a five-item list.

In the old days, the only way around this was to rigorously check the inputs for every function to make sure they made sense. Typically, functions had special return values to indicate an error condition; for example, they could return a negative number to indicate that a positive value couldn't be calculated. Different numbers might mean different errors occurred. Any code that called this function would have to explicitly check for an error condition and act accordingly. A lot of code didn't bother to do this, and programs simply crashed. However, in the object-oriented world, this is not the case.

In this chapter, we will study **exceptions**, special error objects that only need to be handled when it makes sense to handle them. In particular, we will cover:

- How to cause an exception to occur
- How to recover when an exception has occurred
- How to handle different exception types in different ways
- Cleaning up when an exception has occurred
- Creating new types of exception
- Using the exception syntax for flow control

Raising exceptions

In principle, an exception is just an object. There are many different exception classes available, and we can easily define more of our own. The one thing they all have in common is that they inherit from a built-in class called `BaseException`. These exception objects become special when they are handled inside the program's flow of control. When an exception occurs, everything that was supposed to happen doesn't happen, unless it was supposed to happen when an exception occurred. Make sense? Don't worry, it will!

The easiest way to cause an exception to occur is to do something silly! Chances are you've done this already and seen the exception output. For example, any time Python encounters a line in your program that it can't understand, it bails with `SyntaxError`, which is a type of exception. Here's a common one:

```
>>> print "hello world"
  File "<stdin>", line 1
    print "hello world"
                      ^
SyntaxError: invalid syntax
```

This `print` statement was a valid command in Python 2 and previous versions, but in Python 3, because `print` is now a function, we have to enclose the arguments in parenthesis. So, if we type the preceding command into a Python 3 interpreter, we get the `SyntaxError`.

In addition to `SyntaxError`, some other common exceptions, which we can handle, are shown in the following example:

```
>>> x = 5 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero

>>> lst = [1,2,3]
>>> print(lst[3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

>>> lst + 2
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list

>>> lst.add
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'add'

>>> d = {'a': 'hello'}
>>> d['b']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'b'

>>> print(this_is_not_a_var)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'this_is_not_a_var' is not defined
```

Sometimes these exceptions are indicators of something wrong in our program (in which case we would go to the indicated line number and fix it), but they also occur in legitimate situations. A `ZeroDivisionError` doesn't always mean we received an invalid input. It could also mean we have received a different input. The user may have entered a zero by mistake, or on purpose, or it may represent a legitimate value, such as an empty bank account or the age of a newborn child.

You may have noticed all the preceding built-in exceptions end with the name `Error`. In Python, the words `error` and `exception` are used almost interchangeably. Errors are sometimes considered more dire than exceptions, but they are dealt with in exactly the same way. Indeed, all the error classes in the preceding example have `Exception` (which extends `BaseException`) as their superclass.

Raising an exception

We'll get to handling exceptions in a minute, but first, let's discover what we should do if we're writing a program that needs to inform the user or a calling function that the inputs are somehow invalid. Wouldn't it be great if we could use the same mechanism that Python uses? Well, we can! Here's a simple class that adds items to a list only if they are even numbered integers:

```
    class EvenOnly(list):
        def append(self, integer):
            if not isinstance(integer, int):
                raise TypeError("Only integers can be added")
            if integer % 2:
                raise ValueError("Only even numbers can be added")
            super().append(integer)
```

This class extends the `list` built-in, as we discussed in *Chapter 2, Objects in Python*, and overrides the `append` method to check two conditions that ensure the item is an even integer. We first check if the input is an instance of the `int` type, and then use the modulus operator to ensure it is divisible by two. If either of the two conditions is not met, the `raise` keyword causes an exception to occur. The `raise` keyword is simply followed by the object being raised as an exception. In the preceding example, two objects are newly constructed from the built-in classes `TypeError` and `ValueError`. The raised object could just as easily be an instance of a new exception class we create ourselves (we'll see how shortly), an exception that was defined elsewhere, or even an exception object that has been previously raised and handled. If we test this class in the Python interpreter, we can see that it is outputting useful error information when exceptions occur, just as before:

```
>>> e = EvenOnly()
>>> e.append("a string")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "even_integers.py", line 7, in add
    raise TypeError("Only integers can be added")
TypeError: Only integers can be added

>>> e.append(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "even_integers.py", line 9, in add
    raise ValueError("Only even numbers can be added")
ValueError: Only even numbers can be added
>>> e.append(2)
```

While this class is effective for demonstrating exceptions in action, it isn't very good at its job. It is still possible to get other values into the list using index notation or slice notation. This can all be avoided by overriding other appropriate methods, some of which are double-underscore methods.

The effects of an exception

When an exception is raised, it appears to stop program execution immediately. Any lines that were supposed to run after the exception is raised are not executed, and unless the exception is dealt with, the program will exit with an error message. Take a look at this simple function:

```
def no_return():
    print("I am about to raise an exception")
    raise Exception("This is always raised")
    print("This line will never execute")
    return "I won't be returned"
```

If we execute this function, we see that the first `print` call is executed and then the exception is raised. The second `print` statement is never executed, and the `return` statement never executes either:

```
>>> no_return()
I am about to raise an exception
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "exception_quits.py", line 3, in no_return
    raise Exception("This is always raised")
Exception: This is always raised
```

Furthermore, if we have a function that calls another function that raises an exception, nothing will be executed in the first function after the point where the second function was called. Raising an exception stops all execution right up through the function call stack until it is either handled or forces the interpreter to exit. To demonstrate, let's add a second function that calls the earlier one:

```
def call_exceptor():
    print("call_exceptor starts here...")
    no_return()
    print("an exception was raised...")
    print("...so these lines don't run")
```

When we call this function, we see that the first `print` statement executes, as well as the first line in the `no_return` function. But once the exception is raised, nothing else executes:

```
>>> call_exceptor()
call_exceptor starts here...
I am about to raise an exception
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "method_calls_excepting.py", line 9, in call_exceptor
    no_return()
  File "method_calls_excepting.py", line 3, in no_return
    raise Exception("This is always raised")
Exception: This is always raised
```

We'll soon see that when the interpreter is not actually taking a shortcut and exiting immediately, we can react to and deal with the exception inside either method. Indeed, exceptions can be handled at any level after they are initially raised.

Look at the exception's output (called a traceback) from bottom to top, and notice how both methods are listed. Inside `no_return`, the exception is initially raised. Then, just above that, we see that inside `call_exceptor`, that pesky `no_return` function was called and the exception bubbled up to the calling method. From there, it went up one more level to the main interpreter, which, not knowing what else to do with it, gave up and printed a traceback.

Handling exceptions

Now let's look at the tail side of the exception coin. If we encounter an exception situation, how should our code react to or recover from it? We handle exceptions by wrapping any code that might throw one (whether it is exception code itself, or a call to any function or method that may have an exception raised inside it) inside a `try...except` clause. The most basic syntax looks like this:

```
try:
    no_return()
except:
    print("I caught an exception")
print("executed after the exception")
```

If we run this simple script using our existing `no_return` function, which as we know very well, always throws an exception, we get this output:

```
I am about to raise an exception
I caught an exception
executed after the exception
```

The `no_return` function happily informs us that it is about to raise an exception, but we fooled it and caught the exception. Once caught, we were able to clean up after ourselves (in this case, by outputting that we were handling the situation), and continue on our way, with no interference from that offensive function. The remainder of the code in the `no_return` function still went unexecuted, but the code that called the function was able to recover and continue.

Note the indentation around `try` and `except`. The `try` clause wraps any code that might throw an exception. The `except` clause is then back on the same indentation level as the `try` line. Any code to handle the exception is indented after the `except` clause. Then normal code resumes at the original indentation level.

The problem with the preceding code is that it will catch any type of exception. What if we were writing some code that could raise both a `TypeError` and a `ZeroDivisionError`? We might want to catch the `ZeroDivisionError`, but let the `TypeError` propagate to the console. Can you guess the syntax?

Here's a rather silly function that does just that:

```
def funny_division(divider):
    try:
        return 100 / divider
    except ZeroDivisionError:
        return "Zero is not a good idea!"

print(funny_division(0))
print(funny_division(50.0))
print(funny_division("hello"))
```

The function is tested with `print` statements that show it behaving as expected:

```
Zero is not a good idea!
2.0
Traceback (most recent call last):
  File "catch_specific_exception.py", line 9, in <module>
    print(funny_division("hello"))
  File "catch_specific_exception.py", line 3, in funny_division
    return 100 / anumber
TypeError: unsupported operand type(s) for /: 'int' and 'str'.
```

The first line of output shows that if we enter `0`, we get properly mocked. If we call with a valid number (note that it's not an integer, but it's still a valid divisor), it operates correctly. Yet if we enter a string (you were wondering how to get a `TypeError`, weren't you?), it fails with an exception. If we had used an empty `except` clause that didn't specify a `ZeroDivisionError`, it would have accused us of dividing by zero when we sent it a string, which is not a proper behavior at all.

We can even catch two or more different exceptions and handle them with the same code. Here's an example that raises three different types of exception. It handles `TypeError` and `ZeroDivisionError` with the same exception handler, but it may also raise a `ValueError` if you supply the number `13`:

```
    def funny_division2(anumber):
        try:
            if anumber == 13:
                raise ValueError("13 is an unlucky number")
            return 100 / anumber
        except (ZeroDivisionError, TypeError):
            return "Enter a number other than zero"

    for val in (0, "hello", 50.0, 13):

        print("Testing {}:".format(val), end=" ")
        print(funny_division2(val))
```

The `for` loop at the bottom loops over several test inputs and prints the results. If you're wondering about that `end` argument in the `print` statement, it just turns the default trailing newline into a space so that it's joined with the output from the next line. Here's a run of the program:

```
Testing 0: Enter a number other than zero
Testing hello: Enter a number other than zero
Testing 50.0: 2.0
Testing 13: Traceback (most recent call last):
  File "catch_multiple_exceptions.py", line 11, in <module>
    print(funny_division2(val))
  File "catch_multiple_exceptions.py", line 4, in funny_division2
    raise ValueError("13 is an unlucky number")
ValueError: 13 is an unlucky number
```

The number `0` and the string are both caught by the `except` clause, and a suitable error message is printed. The exception from the number `13` is not caught because it is a `ValueError`, which was not included in the types of exceptions being handled. This is all well and good, but what if we want to catch different exceptions and do different things with them? Or maybe we want to do something with an exception and then allow it to continue to bubble up to the parent function, as if it had never been caught? We don't need any new syntax to deal with these cases. It's possible to stack `except` clauses, and only the first match will be executed. For the second question, the `raise` keyword, with no arguments, will reraise the last exception if we're already inside an exception handler. Observe in the following code:

```
def funny_division3(anumber):
    try:
        if anumber == 13:
            raise ValueError("13 is an unlucky number")
        return 100 / anumber
    except ZeroDivisionError:
        return "Enter a number other than zero"
    except TypeError:
        return "Enter a numerical value"
    except ValueError:
        print("No, No, not 13!")
        raise
```

The last line reraises the `ValueError`, so after outputting `No, No, not 13!`, it will raise the exception again; we'll still get the original stack trace on the console.

If we stack exception clauses like we did in the preceding example, only the first matching clause will be run, even if more than one of them fits. How can more than one clause match? Remember that exceptions are objects, and can therefore be subclassed. As we'll see in the next section, most exceptions extend the `Exception` class (which is itself derived from `BaseException`). If we catch `Exception` before we catch `TypeError`, then only the `Exception` handler will be executed, because `TypeError` is an `Exception` by inheritance.

This can come in handy in cases where we want to handle some exceptions specifically, and then handle all remaining exceptions as a more general case. We can simply catch `Exception` after catching all the specific exceptions and handle the general case there.

Sometimes, when we catch an exception, we need a reference to the `Exception` object itself. This most often happens when we define our own exceptions with custom arguments, but can also be relevant with standard exceptions. Most exception classes accept a set of arguments in their constructor, and we might want to access those attributes in the exception handler. If we define our own exception class, we can even call custom methods on it when we catch it. The syntax for capturing an exception as a variable uses the `as` keyword:

```
try:
    raise ValueError("This is an argument")
except ValueError as e:
    print("The exception arguments were", e.args)
```

If we run this simple snippet, it prints out the string argument that we passed into `ValueError` upon initialization.

We've seen several variations on the syntax for handling exceptions, but we still don't know how to execute code regardless of whether or not an exception has occurred. We also can't specify code that should be executed only if an exception does not occur. Two more keywords, `finally` and `else`, can provide the missing pieces. Neither one takes any extra arguments. The following example randomly picks an exception to throw and raises it. Then some not-so-complicated exception handling code is run that illustrates the newly introduced syntax:

```
import random
some_exceptions = [ValueError, TypeError, IndexError, None]

try:
    choice = random.choice(some_exceptions)
    print("raising {}".format(choice))
    if choice:
        raise choice("An error")
except ValueError:
    print("Caught a ValueError")
except TypeError:
    print("Caught a TypeError")
except Exception as e:
    print("Caught some other error: %s" %
        ( e.__class__.__name__))
else:
    print("This code called if there is no exception")
finally:
    print("This cleanup code is always called")
```

If we run this example—which illustrates almost every conceivable exception handling scenario—a few times, we'll get different output each time, depending on which exception `random` chooses. Here are some example runs:

```
$ python finally_and_else.py
raising None
This code called if there is no exception
This cleanup code is always called

$ python finally_and_else.py
raising <class 'TypeError'>
Caught a TypeError
This cleanup code is always called

$ python finally_and_else.py
raising <class 'IndexError'>
Caught some other error: IndexError
This cleanup code is always called

$ python finally_and_else.py
raising <class 'ValueError'>
Caught a ValueError
This cleanup code is always called
```

Note how the `print` statement in the `finally` clause is executed no matter what happens. This is extremely useful when we need to perform certain tasks after our code has finished running (even if an exception has occurred). Some common examples include:

- Cleaning up an open database connection
- Closing an open file
- Sending a closing handshake over the network

The `finally` clause is also very important when we execute a `return` statement from inside a `try` clause. The `finally` handle will still be executed before the value is returned.

Also, pay attention to the output when no exception is raised: both the `else` and the `finally` clauses are executed. The `else` clause may seem redundant, as the code that should be executed only when no exception is raised could just be placed after the entire `try...except` block. The difference is that the `else` block will still be executed if an exception is caught and handled. We'll see more on this when we discuss using exceptions as flow control later.

Any of the `except`, `else`, and `finally` clauses can be omitted after a `try` block (although `else` by itself is invalid). If you include more than one, the `except` clauses must come first, then the `else` clause, with the `finally` clause at the end. The order of the `except` clauses normally goes from most specific to most generic.

The exception hierarchy

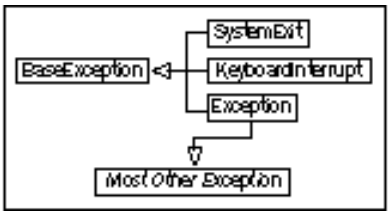
We've already seen several of the most common built-in exceptions, and you'll probably encounter the rest over the course of your regular Python development. As we noticed earlier, most exceptions are subclasses of the `Exception` class. But this is not true of all exceptions. `Exception` itself actually inherits from a class called `BaseException`. In fact, all exceptions must extend the `BaseException` class or one of its subclasses.

There are two key exceptions, `SystemExit` and `KeyboardInterrupt`, that derive directly from `BaseException` instead of `Exception`. The `SystemExit` exception is raised whenever the program exits naturally, typically because we called the `sys.exit` function somewhere in our code (for example, when the user selected an exit menu item, clicked the "close" button on a window, or entered a command to shut down a server). The exception is designed to allow us to clean up code before the program ultimately exits, so we generally don't need to handle it explicitly (because cleanup code happens inside a `finally` clause).

If we do handle it, we would normally reraise the exception, since catching it would stop the program from exiting. There are, of course, situations where we might want to stop the program exiting, for example, if there are unsaved changes and we want to prompt the user if they really want to exit. Usually, if we handle `SystemExit` at all, it's because we want to do something special with it, or are anticipating it directly. We especially don't want it to be accidentally caught in generic clauses that catch all normal exceptions. This is why it derives directly from `BaseException`.

The `KeyboardInterrupt` exception is common in command-line programs. It is thrown when the user explicitly interrupts program execution with an OS-dependent key combination (normally, *Ctrl* + *C*). This is a standard way for the user to deliberately interrupt a running program, and like `SystemExit`, it should almost always respond by terminating the program. Also, like `SystemExit`, it should handle any cleanup tasks inside `finally` blocks.

Here is a class diagram that fully illustrates the exception hierarchy:



When we use the `except:` clause without specifying any type of exception, it will catch all subclasses of `BaseException`; which is to say, it will catch all exceptions, including the two special ones. Since we almost always want these to get special treatment, it is unwise to use the `except:` statement without arguments. If you want to catch all exceptions other than `SystemExit` and `KeyboardInterrupt`, explicitly catch `Exception`.

Furthermore, if you do want to catch all exceptions, I suggest using the syntax `except BaseException:` instead of a raw `except:`. This helps explicitly tell future readers of your code that you are intentionally handling the special case exceptions.

Defining our own exceptions

Often, when we want to raise an exception, we find that none of the built-in exceptions are suitable. Luckily, it's trivial to define new exceptions of our own. The name of the class is usually designed to communicate what went wrong, and we can provide arbitrary arguments in the initializer to include additional information.

All we have to do is inherit from the `Exception` class. We don't even have to add any content to the class! We can, of course, extend `BaseException` directly, but then it will not be caught by generic `except Exception` clauses.

Here's a simple exception we might use in a banking application:

```
class InvalidWithdrawal(Exception):
    pass

raise InvalidWithdrawal("You don't have $50 in your account")
```

The last line illustrates how to raise the newly defined exception. We are able to pass an arbitrary number of arguments into the exception. Often a string message is used, but any object that might be useful in a later exception handler can be stored. The `Exception.__init__` method is designed to accept any arguments and store them as a tuple in an attribute named `args`. This makes exceptions easier to define without needing to override `__init__`.

Of course, if we do want to customize the initializer, we are free to do so. Here's an exception whose initializer accepts the current balance and the amount the user wanted to withdraw. In addition, it adds a method to calculate how overdrawn the request was:

```
class InvalidWithdrawal(Exception):
    def __init__(self, balance, amount):
        super().__init__("account doesn't have ${}".format(
            amount))
        self.amount = amount
        self.balance = balance

    def overage(self):
        return self.amount - self.balance

raise InvalidWithdrawal(25, 50)
```

The `raise` statement at the end illustrates how to construct this exception. As you can see, we can do anything with an exception that we would do with other objects. We could catch an exception and pass it around as a working object, although it is more common to include a reference to the working object as an attribute on an exception and pass that around instead.

Here's how we would handle an `InvalidWithdrawal` exception if one was raised:

```
try:
    raise InvalidWithdrawal(25, 50)
except InvalidWithdrawal as e:
    print("I'm sorry, but your withdrawal is "
            "more than your balance by "
            "${}".format(e.overage()))
```

Here we see a valid use of the `as` keyword. By convention, most Python coders name the exception variable `e`, although, as usual, you are free to call it `ex`, `exception`, or `aunt_sally` if you prefer.

There are many reasons for defining our own exceptions. It is often useful to add information to the exception or log it in some way. But the utility of custom exceptions truly comes to light when creating a framework, library, or API that is intended for access by other programmers. In that case, be careful to ensure your code is raising exceptions that make sense to the client programmer. They should be easy to handle and clearly describe what went on. The client programmer should easily see how to fix the error (if it reflects a bug in their code) or handle the exception (if it's a situation they need to be made aware of).

Exceptions aren't exceptional. Novice programmers tend to think of exceptions as only useful for exceptional circumstances. However, the definition of exceptional circumstances can be vague and subject to interpretation. Consider the following two functions:

```
def divide_with_exception(number, divisor):
    try:
        print("{} / {} = {}".format(
            number, divisor, number / divisor * 1.0))
    except ZeroDivisionError:
        print("You can't divide by zero")

def divide_with_if(number, divisor):
    if divisor == 0:
        print("You can't divide by zero")
    else:
        print("{} / {} = {}".format(
            number, divisor, number / divisor * 1.0))
```

These two functions behave identically. If `divisor` is zero, an error message is printed; otherwise, a message printing the result of division is displayed. We could avoid a `ZeroDivisionError` ever being thrown by testing for it with an `if` statement. Similarly, we can avoid an `IndexError` by explicitly checking whether or not the parameter is within the confines of the list, and a `KeyError` by checking if the key is in a dictionary.

But we shouldn't do this. For one thing, we might write an `if` statement that checks whether or not the index is lower than the parameters of the list, but forget to check negative values.

Remember, Python lists support negative indexing; `-1` refers to the last element in the list.

Eventually, we would discover this and have to find all the places where we were checking code. But if we had simply caught the `IndexError` and handled it, our code would just work.

Python programmers tend to follow a model of *Ask forgiveness rather than permission*, which is to say, they execute code and then deal with anything that goes wrong. The alternative, to *look before you leap*, is generally frowned upon. There are a few reasons for this, but the main one is that it shouldn't be necessary to burn CPU cycles looking for an unusual situation that is not going to arise in the normal path through the code. Therefore, it is wise to use exceptions for exceptional circumstances, even if those circumstances are only a little bit exceptional. Taking this argument further, we can actually see that the exception syntax is also effective for flow control. Like an `if` statement, exceptions can be used for decision making, branching, and message passing.

Imagine an inventory application for a company that sells widgets and gadgets. When a customer makes a purchase, the item can either be available, in which case the item is removed from inventory and the number of items left is returned, or it might be out of stock. Now, being out of stock is a perfectly normal thing to happen in an inventory application. It is certainly not an exceptional circumstance. But what do we return if it's out of stock? A string saying out of stock? A negative number? In both cases, the calling method would have to check whether the return value is a positive integer or something else, to determine if it is out of stock. That seems a bit messy. Instead, we can raise `OutOfStockException` and use the `try` statement to direct program flow control. Make sense? In addition, we want to make sure we don't sell the same item to two different customers, or sell an item that isn't in stock yet. One way to facilitate this is to lock each type of item to ensure only one person can update it at a time. The user must lock the item, manipulate the item (purchase, add stock, count items left…), and then unlock the item. Here's an incomplete `Inventory` example with docstrings that describes what some of the methods should do:

```
class Inventory:
    def lock(self, item_type):
        '''Select the type of item that is going to
        be manipulated. This method will lock the
        item so nobody else can manipulate the
        inventory until it's returned. This prevents
        selling the same item to two different
        customers.'''
        pass

    def unlock(self, item_type):
        '''Release the given type so that other
        customers can access it.'''
        pass

    def purchase(self, item_type):
```

```
        '''If the item is not locked, raise an
        exception. If the item_type  does not exist,
        raise an exception. If the item is currently
        out of stock, raise an exception. If the item
        is available, subtract one item and return
        the number of items left.'''
        pass
```

We could hand this object prototype to a developer and have them implement the methods to do exactly as they say while we work on the code that needs to make a purchase. We'll use Python's robust exception handling to consider different branches, depending on how the purchase was made:

```
item_type = 'widget'
inv = Inventory()
inv.lock(item_type)
try:
    num_left = inv.purchase(item_type)
except InvalidItemType:
    print("Sorry, we don't sell {}".format(item_type))
except OutOfStock:
    print("Sorry, that item is out of stock.")
else:
    print("Purchase complete. There are "
            "{} {}s left".format(num_left, item_type))
finally:
    inv.unlock(item_type)
```

Pay attention to how all the possible exception handling clauses are used to ensure the correct actions happen at the correct time. Even though `OutOfStock` is not a terribly exceptional circumstance, we are able to use an exception to handle it suitably. This same code could be written with an `if...elif...else` structure, but it wouldn't be as easy to read or maintain.

We can also use exceptions to pass messages between different methods. For example, if we wanted to inform the customer as to what date the item is expected to be in stock again, we could ensure our `OutOfStock` object requires a `back_in_stock` parameter when it is constructed. Then, when we handle the exception, we can check that value and provide additional information to the customer. The information attached to the object can be easily passed between two different parts of the program. The exception could even provide a method that instructs the inventory object to reorder or backorder an item.

Using exceptions for flow control can make for some handy program designs. The important thing to take from this discussion is that exceptions are not a bad thing that we should try to avoid. Having an exception occur does not mean that you should have prevented this exceptional circumstance from happening. Rather, it is just a powerful way to communicate information between two sections of code that may not be directly calling each other.

Case study

We've been looking at the use and handling of exceptions at a fairly low level of detail—syntax and definitions. This case study will help tie it all in with our previous chapters so we can see how exceptions are used in the larger context of objects, inheritance, and modules.

Today, we'll be designing a simple central authentication and authorization system. The entire system will be placed in one module, and other code will be able to query that module object for authentication and authorization purposes. We should admit, from the start, that we aren't security experts, and that the system we are designing may be full of security holes. Our purpose is to study exceptions, not to secure a system. It will be sufficient, however, for a basic login and permission system that other code can interact with. Later, if that other code needs to be made more secure, we can have a security or cryptography expert review or rewrite our module, preferably without changing the API.

Authentication is the process of ensuring a user is really the person they say they are. We'll follow the lead of common web systems today, which use a username and private password combination. Other methods of authentication include voice recognition, fingerprint or retinal scanners, and identification cards.

Authorization, on the other hand, is all about determining whether a given (authenticated) user is permitted to perform a specific action. We'll create a basic permission list system that stores a list of the specific people allowed to perform each action.

In addition, we'll add some administrative features to allow new users to be added to the system. For brevity, we'll leave out editing of passwords or changing of permissions once they've been added, but these (highly necessary) features can certainly be added in the future.

There's a simple analysis; now let's proceed with design. We're obviously going to need a `User` class that stores the username and an encrypted password. This class will also allow a user to log in by checking whether a supplied password is valid. We probably won't need a `Permission` class, as those can just be strings mapped to a list of users using a dictionary. We should have a central `Authenticator` class that handles user management and logging in or out. The last piece of the puzzle is an `Authorizor` class that deals with permissions and checking whether a user can perform an activity. We'll provide a single instance of each of these classes in the `auth` module so that other modules can use this central mechanism for all their authentication and authorization needs. Of course, if they want to instantiate private instances of these classes, for non-central authorization activities, they are free to do so.

We'll also be defining several exceptions as we go along. We'll start with a special `AuthException` base class that accepts a `username` and optional `user` object as parameters; most of our self-defined exceptions will inherit from this one.

Let's build the `User` class first; it seems simple enough. A new user can be initialized with a username and password. The password will be stored encrypted to reduce the chances of its being stolen. We'll also need a `check_password` method to test whether a supplied password is the correct one. Here is the class in full:

```
import hashlib

class User:
    def __init__(self, username, password):
        '''Create a new user object. The password
        will be encrypted before storing.'''
        self.username = username
        self.password = self._encrypt_pw(password)
        self.is_logged_in = False

    def _encrypt_pw(self, password):
        '''Encrypt the password with the username and return
        the sha digest.'''
        hash_string = (self.username + password)
        hash_string = hash_string.encode("utf8")
        return hashlib.sha256(hash_string).hexdigest()

    def check_password(self, password):
        '''Return True if the password is valid for this
        user, false otherwise.'''
        encrypted = self._encrypt_pw(password)
        return encrypted == self.password
```

Since the code for encrypting a password is required in both `__init__` and `check_password`, we pull it out to its own method. This way, it only needs to be changed in one place if someone realizes it is insecure and needs improvement. This class could easily be extended to include mandatory or optional personal details, such as names, contact information, and birth dates.

Before we write code to add users (which will happen in the as-yet undefined `Authenticator` class), we should examine some use cases. If all goes well, we can add a user with a username and password; the `User` object is created and inserted into a dictionary. But in what ways can all not go well? Well, clearly we don't want to add a user with a username that already exists in the dictionary. If we did so, we'd overwrite an existing user's data and the new user might have access to that user's privileges. So, we'll need a `UsernameAlreadyExists` exception. Also, for security's sake, we should probably raise an exception if the password is too short. Both of these exceptions will extend `AuthException`, which we mentioned earlier. So, before writing the `Authenticator` class, let's define these three exception classes:

```
class AuthException(Exception):
    def __init__(self, username, user=None):
        super().__init__(username, user)
        self.username = username
        self.user = user

class UsernameAlreadyExists(AuthException):
    pass

class PasswordTooShort(AuthException):
    pass
```

The `AuthException` requires a username and has an optional user parameter. This second parameter should be an instance of the `User` class associated with that username. The two specific exceptions we're defining simply need to inform the calling class of an exceptional circumstance, so we don't need to add any extra methods to them.

Now let's start on the `Authenticator` class. It can simply be a mapping of usernames to user objects, so we'll start with a dictionary in the initialization function. The method for adding a user needs to check the two conditions (password length and previously existing users) before creating a new `User` instance and adding it to the dictionary:

```
class Authenticator:
    def __init__(self):
        '''Construct an authenticator to manage
```

```
            users logging in and out.'''
            self.users = {}

        def add_user(self, username, password):
            if username in self.users:
                raise UsernameAlreadyExists(username)
            if len(password) < 6:
                raise PasswordTooShort(username)
            self.users[username] = User(username, password)
```

We could, of course, extend the password validation to raise exceptions for passwords that are too easy to crack in other ways, if we desired. Now let's prepare the `login` method. If we weren't thinking about exceptions just now, we might just want the method to return `True` or `False`, depending on whether the login was successful or not. But we are thinking about exceptions, and this could be a good place to use them for a not-so-exceptional circumstance. We could raise different exceptions, for example, if the username does not exist or the password does not match. This will allow anyone trying to log a user in to elegantly handle the situation using a `try/except/else` clause. So, first we add these new exceptions:

```
class InvalidUsername(AuthException):
    pass

class InvalidPassword(AuthException):
    pass
```

Then we can define a simple `login` method to our `Authenticator` class that raises these exceptions if necessary. If not, it flags the `user` as logged in and returns:

```
        def login(self, username, password):
            try:
                user = self.users[username]
            except KeyError:
                raise InvalidUsername(username)

            if not user.check_password(password):
                raise InvalidPassword(username, user)

            user.is_logged_in = True
            return True
```

Notice how the `KeyError` is handled. This could have been handled using `if username not in self.users:` instead, but we chose to handle the exception directly. We end up eating up this first exception and raising a brand new one of our own that better suits the user-facing API.

We can also add a method to check whether a particular username is logged in. Deciding whether to use an exception here is trickier. Should we raise an exception if the username does not exist? Should we raise an exception if the user is not logged in?

To answer these questions, we need to think about how the method would be accessed. Most often, this method will be used to answer the yes/no question, "Should I allow them access to <*something*>?" The answer will either be, "Yes, the username is valid and they are logged in", or "No, the username is not valid or they are not logged in". Therefore, a Boolean return value is sufficient. There is no need to use exceptions here, just for the sake of using an exception.

```
        def is_logged_in(self, username):
            if username in self.users:
                return self.users[username].is_logged_in
            return False
```

Finally, we can add a default authenticator instance to our module so that the client code can access it easily using `auth.authenticator`:

```
    authenticator = Authenticator()
```

This line goes at the module level, outside any class definition, so the authenticator variable can be accessed as `auth.authenticator`. Now we can start on the `Authorizor` class, which maps permissions to users. The `Authorizor` class should not permit user access to a permission if they are not logged in, so they'll need a reference to a specific authenticator. We'll also need to set up the permission dictionary upon initialization:

```
    class Authorizor:
        def __init__(self, authenticator):
            self.authenticator = authenticator
            self.permissions = {}
```

Now we can write methods to add new permissions and to set up which users are associated with each permission:

```
        def add_permission(self, perm_name):
            '''Create a new permission that users
            can be added to'''
            try:
                perm_set = self.permissions[perm_name]
            except KeyError:
                self.permissions[perm_name] = set()
            else:
```

```
            raise PermissionError("Permission Exists")

    def permit_user(self, perm_name, username):
        '''Grant the given permission to the user'''
        try:
            perm_set = self.permissions[perm_name]
        except KeyError:
            raise PermissionError("Permission does not exist")
        else:
            if username not in self.authenticator.users:
                raise InvalidUsername(username)
            perm_set.add(username)
```

The first method allows us to create a new permission, unless it already exists, in which case an exception is raised. The second allows us to add a username to a permission, unless either the permission or the username doesn't yet exist.

We use `set` instead of `list` for usernames, so that even if you grant a user permission more than once, the nature of sets means the user is only in the set once. We'll discuss sets further in a later chapter.

A `PermissionError` is raised in both methods. This new error doesn't require a username, so we'll make it extend `Exception` directly, instead of our custom `AuthException`:

```
class PermissionError(Exception):
    pass
```

Finally, we can add a method to check whether a user has a specific `permission` or not. In order for them to be granted access, they have to be both logged into the authenticator and in the set of people who have been granted access to that privilege. If either of these conditions is unsatisfied, an exception is raised:

```
    def check_permission(self, perm_name, username):
        if not self.authenticator.is_logged_in(username):
            raise NotLoggedInError(username)
        try:
            perm_set = self.permissions[perm_name]
        except KeyError:
            raise PermissionError("Permission does not exist")
        else:
            if username not in perm_set:
                raise NotPermittedError(username)
            else:
                return True
```

There are two new exceptions in here; they both take usernames, so we'll define them as subclasses of `AuthException`:

```
class NotLoggedInError(AuthException):
    pass

class NotPermittedError(AuthException):
    pass
```

Finally, we can add a default `authorizor` to go with our default authenticator:

```
authorizor = Authorizor(authenticator)
```

That completes a basic authentication/authorization system. We can test the system at the Python prompt, checking to see whether a user, `joe`, is permitted to do tasks in the paint department:

```
>>> import auth
>>> auth.authenticator.add_user("joe", "joepassword")
>>> auth.authorizor.add_permission("paint")
>>> auth.authorizor.check_permission("paint", "joe")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "auth.py", line 109, in check_permission
    raise NotLoggedInError(username)
auth.NotLoggedInError: joe
>>> auth.authenticator.is_logged_in("joe")
False
>>> auth.authenticator.login("joe", "joepassword")
True
>>> auth.authorizor.check_permission("paint", "joe")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "auth.py", line 116, in check_permission
    raise NotPermittedError(username)
auth.NotPermittedError: joe
>>> auth.authorizor.check_permission("mix", "joe")
Traceback (most recent call last):
  File "auth.py", line 111, in check_permission
    perm_set = self.permissions[perm_name]
```

```
KeyError: 'mix'

During handling of the above exception, another exception occurred:
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "auth.py", line 113, in check_permission
    raise PermissionError("Permission does not exist")
auth.PermissionError: Permission does not exist
>>> auth.authorizor.permit_user("mix", "joe")
Traceback (most recent call last):
  File "auth.py", line 99, in permit_user
    perm_set = self.permissions[perm_name]
KeyError: 'mix'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "auth.py", line 101, in permit_user
    raise PermissionError("Permission does not exist")
auth.PermissionError: Permission does not exist
>>> auth.authorizor.permit_user("paint", "joe")
>>> auth.authorizor.check_permission("paint", "joe")
True
```

While verbose, the preceding output shows all of our code and most of our exceptions in action, but to really understand the API we've defined, we should write some exception handling code that actually uses it. Here's a basic menu interface that allows certain users to change or test a program:

```
import auth

# Set up a test user and permission
auth.authenticator.add_user("joe", "joepassword")
auth.authorizor.add_permission("test program")
auth.authorizor.add_permission("change program")
auth.authorizor.permit_user("test program", "joe")

class Editor:
```

```
    def __init__(self):
        self.username = None
        self.menu_map = {
                "login": self.login,
                "test": self.test,
                "change": self.change,
                "quit": self.quit
           }

    def login(self):
        logged_in = False
        while not logged_in:
            username = input("username: ")
            password = input("password: ")
            try:
                logged_in = auth.authenticator.login(
                        username, password)
            except auth.InvalidUsername:
                print("Sorry, that username does not exist")
            except auth.InvalidPassword:
                print("Sorry, incorrect password")
            else:
                self.username = username
    def is_permitted(self, permission):
        try:
            auth.authorizor.check_permission(
                permission, self.username)
        except auth.NotLoggedInError as e:
            print("{} is not logged in".format(e.username))
            return False
        except auth.NotPermittedError as e:
            print("{} cannot {}".format(
                e.username, permission))
            return False
        else:
            return True

    def test(self):
        if self.is_permitted("test program"):
            print("Testing program now...")

    def change(self):
        if self.is_permitted("change program"):
            print("Changing program now...")

    def quit(self):
```

```
            raise SystemExit()

        def menu(self):
            try:
                answer = ""
                while True:
                    print("""
Please enter a command:
\tlogin\tLogin
\ttest\tTest the program
\tchange\tChange the program
\tquit\tQuit
""")
                    answer = input("enter a command: ").lower()
                    try:
                        func = self.menu_map[answer]
                    except KeyError:
                        print("{} is not a valid option".format(
                            answer))
                    else:
                        func()
            finally:
                print("Thank you for testing the auth module")

Editor().menu()
```

This rather long example is conceptually very simple. The `is_permitted` method is probably the most interesting; this is a mostly internal method that is called by both `test` and `change` to ensure the user is permitted access before continuing. Of course, those two methods are stubs, but we aren't writing an editor here; we're illustrating the use of exceptions and exception handlers by testing an authentication and authorization framework!

Exercises

If you've never dealt with exceptions before, the first thing you need to do is look at any old Python code you've written and notice if there are places you should have been handling exceptions. How would you handle them? Do you need to handle them at all? Sometimes, letting the exception propagate to the console is the best way to communicate to the user, especially if the user is also the script's coder. Sometimes, you can recover from the error and allow the program to continue. Sometimes, you can only reformat the error into something the user can understand and display it to them.

Some common places to look are file I/O (is it possible your code will try to read a file that doesn't exist?), mathematical expressions (is it possible that a value you are dividing by is zero?), list indices (is the list empty?), and dictionaries (does the key exist?). Ask yourself if you should ignore the problem, handle it by checking values first, or handle it with an exception. Pay special attention to areas where you might have used `finally` and `else` to ensure the correct code is executed under all conditions.

Now write some new code. Think of a program that requires authentication and authorization, and try writing some code that uses the `auth` module we built in the case study. Feel free to modify the module if it's not flexible enough. Try to handle all the exceptions in a sensible way. If you're having trouble coming up with something that requires authentication, try adding authorization to the notepad example from *Chapter 2*, *Objects in Python*, or add authorization to the `auth` module itself—it's not a terribly useful module if just anybody can start adding permissions! Maybe require an administrator username and password before allowing privileges to be added or changed.

Finally, try to think of places in your code where you can raise exceptions. It can be in code you've written or are working on; or you can write a new project as an exercise. You'll probably have the best luck for designing a small framework or API that is meant to be used by other people; exceptions are a terrific communication tool between your code and someone else's. Remember to design and document any self-raised exceptions as part of the API, or they won't know whether or how to handle them!

Summary

In this chapter, we went into the gritty details of raising, handling, defining, and manipulating exceptions. Exceptions are a powerful way to communicate unusual circumstances or error conditions without requiring a calling function to explicitly check return values. There are many built-in exceptions and raising them is trivially easy. There are several different syntaxes for handling different exception events.

In the next chapter, everything we've studied so far will come together as we discuss how object-oriented programming principles and structures should best be applied in Python applications.

5
When to Use Object-oriented Programming

In previous chapters, we've covered many of the defining features of object-oriented programming. We now know the principles and paradigms of object-oriented design, and we've covered the syntax of object-oriented programming in Python.

Yet, we don't know exactly how and when to utilize these principles and syntax in practice. In this chapter, we'll discuss some useful applications of the knowledge we've gained, picking up some new topics along the way:

- How to recognize objects
- Data and behaviors, once again
- Wrapping data in behavior using properties
- Restricting data using behavior
- The Don't Repeat Yourself principle
- Recognizing repeated code

Treat objects as objects

This may seem obvious; you should generally give separate objects in your problem domain a special class in your code. We've seen examples of this in the case studies in previous chapters; first, we identify objects in the problem and then model their data and behaviors.

Identifying objects is a very important task in object-oriented analysis and programming. But it isn't always as easy as counting the nouns in a short paragraph, as we've been doing. Remember, objects are things that have both data and behavior. If we are working only with data, we are often better off storing it in a list, set, dictionary, or some other Python data structure (which we'll be covering thoroughly in *Chapter 6*, *Python Data Structures*). On the other hand, if we are working only with behavior, but no stored data, a simple function is more suitable.

An object, however, has both data and behavior. Proficient Python programmers use built-in data structures unless (or until) there is an obvious need to define a class. There is no reason to add an extra level of abstraction if it doesn't help organize our code. On the other hand, the "obvious" need is not always self-evident.

We can often start our Python programs by storing data in a few variables. As the program expands, we will later find that we are passing the same set of related variables to a set of functions. This is the time to think about grouping both variables and functions into a class. If we are designing a program to model polygons in two-dimensional space, we might start with each polygon being represented as a list of points. The points would be modeled as two-tuples (*x*, *y*) describing where that point is located. This is all data, stored in a set of nested data structures (specifically, a list of tuples):

```
square = [(1,1), (1,2), (2,2), (2,1)]
```

Now, if we want to calculate the distance around the perimeter of the polygon, we simply need to sum the distances between the two points. To do this, we also need a function to calculate the distance between two points. Here are two such functions:

```
import math

def distance(p1, p2):
    return math.sqrt((p1[0]-p2[0])**2 + (p1[1]-p2[1])**2)

def perimeter(polygon):
    perimeter = 0
    points = polygon + [polygon[0]]
    for i in range(len(polygon)):
        perimeter += distance(points[i], points[i+1])
    return perimeter
```

Now, as object-oriented programmers, we clearly recognize that a `polygon` class could encapsulate the list of points (data) and the `perimeter` function (behavior). Further, a `point` class, such as we defined in *Chapter 2*, *Objects in Python*, might encapsulate the `x` and `y` coordinates and the `distance` method. The question is: is it valuable to do this?

For the previous code, maybe yes, maybe no. With our recent experience in object-oriented principles, we can write an object-oriented version in record time. Let's compare them

```
import math

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, p2):
        return math.sqrt((self.x-p2.x)**2 + (self.y-p2.y)**2)

class Polygon:
    def __init__(self):
        self.vertices = []

    def add_point(self, point):
        self.vertices.append((point))

    def perimeter(self):
        perimeter = 0
        points = self.vertices + [self.vertices[0]]
        for i in range(len(self.vertices)):
            perimeter += points[i].distance(points[i+1])
        return perimeter
```

As we can see from the highlighted sections, there is twice as much code here as there was in our earlier version, although we could argue that the `add_point` method is not strictly necessary.

Now, to understand the differences a little better, let's compare the two APIs in use. Here's how to calculate the perimeter of a square using the object-oriented code:

```
>>> square = Polygon()
>>> square.add_point(Point(1,1))
>>> square.add_point(Point(1,2))
>>> square.add_point(Point(2,2))
>>> square.add_point(Point(2,1))
>>> square.perimeter()
4.0
```

That's fairly succinct and easy to read, you might think, but let's compare it to the function-based code:

```
>>> square = [(1,1), (1,2), (2,2), (2,1)]
>>> perimeter(square)
4.0
```

Hmm, maybe the object-oriented API isn't so compact! That said, I'd argue that it was easier to *read* than the functional example: How do we know what the list of tuples is supposed to represent in the second version? How do we remember what kind of object (a list of two-tuples? That's not intuitive!) we're supposed to pass into the `perimeter` function? We would need a lot of documentation to explain how these functions should be used.

In contrast, the object-oriented code is relatively self-documenting, we just have to look at the list of methods and their parameters to know what the object does and how to use it. By the time we wrote all the documentation for the functional version, it would probably be longer than the object-oriented code.

Finally, code length is not a good indicator of code complexity. Some programmers get hung up on complicated "one liners" that do an incredible amount of work in one line of code. This can be a fun exercise, but the result is often unreadable, even to the original author the following day. Minimizing the amount of code can often make a program easier to read, but do not blindly assume this is the case.

Luckily, this trade-off isn't necessary. We can make the object-oriented `Polygon` API as easy to use as the functional implementation. All we have to do is alter our `Polygon` class so that it can be constructed with multiple points. Let's give it an initializer that accepts a list of `Point` objects. In fact, let's allow it to accept tuples too, and we can construct the `Point` objects ourselves, if needed:

```
def __init__(self, points=None):
    points = points if points else []
    self.vertices = []
    for point in points:
        if isinstance(point, tuple):
            point = Point(*point)
        self.vertices.append(point)
```

This initializer goes through the list and ensures that any tuples are converted to points. If the object is not a tuple, we leave it as is, assuming that it is either a `Point` object already, or an unknown duck-typed object that can act like a `Point` object.

Still, there's no clear winner between the object-oriented and more data-oriented versions of this code. They both do the same thing. If we have new functions that accept a polygon argument, such as `area(polygon)` or `point_in_polygon(polygon, x, y)`, the benefits of the object-oriented code become increasingly obvious. Likewise, if we add other attributes to the polygon, such as `color` or `texture`, it makes more and more sense to encapsulate that data into a single class.

The distinction is a design decision, but in general, the more complicated a set of data is, the more likely it is to have multiple functions specific to that data, and the more useful it is to use a class with attributes and methods instead.

When making this decision, it also pays to consider how the class will be used. If we're only trying to calculate the perimeter of one polygon in the context of a much greater problem, using a function will probably be quickest to code and easier to use "one time only". On the other hand, if our program needs to manipulate numerous polygons in a wide variety of ways (calculate perimeter, area, intersection with other polygons, move or scale them, and so on), we have most certainly identified an object; one that needs to be extremely versatile.

Additionally, pay attention to the interaction between objects. Look for inheritance relationships; inheritance is impossible to model elegantly without classes, so make sure to use them. Look for the other types of relationships we discussed in *Chapter 1*, *Object-oriented Design*, association and composition. Composition can, technically, be modeled using only data structures; for example, we can have a list of dictionaries holding tuple values, but it is often less complicated to create a few classes of objects, especially if there is behavior associated with the data.

Don't rush to use an object just because you can use an object, but *never* neglect to create a class when you need to use a class.

Adding behavior to class data with properties

Throughout this book, we've been focusing on the separation of behavior and data. This is very important in object-oriented programming, but we're about to see that, in Python, the distinction can be uncannily blurry. Python is very good at blurring distinctions; it doesn't exactly help us to "think outside the box". Rather, it teaches us to stop thinking about the box.

Before we get into the details, let's discuss some bad object-oriented theory. Many object-oriented languages (Java is the most notorious) teach us to never access attributes directly. They insist that we write attribute access like this:

```
class Color:
    def __init__(self, rgb_value, name):
        self._rgb_value = rgb_value
        self._name = name

    def set_name(self, name):
        self._name = name

    def get_name(self):
        return self._name
```

The variables are prefixed with an underscore to suggest that they are private (other languages would actually force them to be private). Then the get and set methods provide access to each variable. This class would be used in practice as follows:

```
>>> c = Color("#ff0000", "bright red")
>>> c.get_name()
'bright red'
>>> c.set_name("red")
>>> c.get_name()
'red'
```

This is not nearly as readable as the direct access version that Python favors:

```
class Color:
    def __init__(self, rgb_value, name):
        self.rgb_value = rgb_value
        self.name = name

c = Color("#ff0000", "bright red")
print(c.name)
c.name = "red"
```

So why would anyone insist upon the method-based syntax? Their reasoning is that someday we may want to add extra code when a value is set or retrieved. For example, we could decide to cache a value and return the cached value, or we might want to validate that the value is a suitable input.

In code, we could decide to change the `set_name()` method as follows:

```
def set_name(self, name):
    if not name:
        raise Exception("Invalid Name")
    self._name = name
```

Now, in Java and similar languages, if we had written our original code to do direct attribute access, and then later changed it to a method like the preceding one, we'd have a problem: anyone who had written code that accessed the attribute directly would now have to access the method. If they don't change the access style from attribute access to a function call, their code will be broken. The mantra in these languages is that we should never make public members private. This doesn't make much sense in Python since there isn't any real concept of private members!

Python gives us the `property` keyword to make methods look like attributes. We can therefore write our code to use direct member access, and if we unexpectedly need to alter the implementation to do some calculation when getting or setting that attribute's value, we can do so without changing the interface. Let's see how it looks:

```
class Color:
    def __init__(self, rgb_value, name):
        self.rgb_value = rgb_value
        self._name = name

    def _set_name(self, name):
        if not name:
            raise Exception("Invalid Name")
        self._name = name

    def _get_name(self):
        return self._name

    name = property(_get_name, _set_name)
```

If we had started with the earlier non-method-based class, which set the `name` attribute directly, we could later change the code to look like the preceding one. We first change the `name` attribute into a (semi-) private `_name` attribute. Then we add two more (semi-) private methods to get and set that variable, doing our validation when we set it.

Finally, we have the `property` declaration at the bottom. This is the magic. It creates a new attribute on the `Color` class called `name`, which now replaces the previous `name` attribute. It sets this attribute to be a property, which calls the two methods we just created whenever the property is accessed or changed. This new version of the `Color` class can be used exactly the same way as the previous version, yet it now does validation when we set the `name` attribute:

```
>>> c = Color("#0000ff", "bright red")
>>> print(c.name)
bright red
>>> c.name = "red"
>>> print(c.name)
red
>>> c.name = ""
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "setting_name_property.py", line 8, in _set_name
    raise Exception("Invalid Name")
Exception: Invalid Name
```

So, if we'd previously written code to access the `name` attribute, and then changed it to use our `property` object, the previous code would still work, unless it was sending an empty `property` value, which is the behavior we wanted to forbid in the first place. Success!

Bear in mind that even with the `name` property, the previous code is not 100 percent safe. People can still access the `_name` attribute directly and set it to an empty string if they want to. But if they access a variable we've explicitly marked with an underscore to suggest it is private, they're the ones that have to deal with the consequences, not us.

Properties in detail

Think of the `property` function as returning an object that proxies any requests to set or access the attribute value through the methods we have specified. The `property` keyword is like a constructor for such an object, and that object is set as the public facing member for the given attribute.

This `property` constructor can actually accept two additional arguments, a deletion function and a docstring for the property. The `delete` function is rarely supplied in practice, but it can be useful for logging that a value has been deleted, or possibly to veto deleting if we have reason to do so. The docstring is just a string describing what the property does, no different from the docstrings we discussed in *Chapter 2*, *Objects in Python*. If we do not supply this parameter, the docstring will instead be copied from the docstring for the first argument: the getter method. Here is a silly example that simply states whenever any of the methods are called:

```
class Silly:
    def _get_silly(self):
        print("You are getting silly")
        return self._silly
    def _set_silly(self, value):
        print("You are making silly {}".format(value))
        self._silly = value
    def _del_silly(self):
        print("Whoah, you killed silly!")
        del self._silly

    silly = property(_get_silly, _set_silly,
            _del_silly, "This is a silly property")
```

If we actually use this class, it does indeed print out the correct strings when we ask it to:

```
>>> s = Silly()
>>> s.silly = "funny"
You are making silly funny
>>> s.silly
You are getting silly
'funny'
>>> del s.silly
Whoah, you killed silly!
```

Further, if we look at the help file for the `Silly` class (by issuing `help(silly)` at the interpreter prompt), it shows us the custom docstring for our `silly` attribute:

```
Help on class Silly in module __main__:

class Silly(builtins.object)
```

```
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  silly
 |      This is a silly property
```

Once again, everything is working as we planned. In practice, properties are normally only defined with the first two parameters: the getter and setter functions. If we want to supply a docstring for a property, we can define it on the getter function; the property proxy will copy it into its own docstring. The deletion function is often left empty because object attributes are rarely deleted. If a coder does try to delete a property that doesn't have a deletion function specified, it will raise an exception. Therefore, if there is a legitimate reason to delete our property, we should supply that function.

Decorators – another way to create properties

If you've never used Python decorators before, you might want to skip this section and come back to it after we've discussed the decorator pattern in *Chapter 10*, *Python Design Patterns I*. However, you don't need to understand what's going on to use the decorator syntax to make property methods more readable.

The property function can be used with the decorator syntax to turn a get function into a property:

```
class Foo:
    @property
    def foo(self):
        return "bar"
```

This applies the `property` function as a decorator, and is equivalent to the previous `foo = property(foo)` syntax. The main difference, from a readability perspective, is that we get to mark the `foo` function as a property at the top of the method, instead of after it is defined, where it can be easily overlooked. It also means we don't have to create private methods with underscore prefixes just to define a property.

Going one step further, we can specify a setter function for the new property as follows:

```
class Foo:
    @property
    def foo(self):
        return self._foo

    @foo.setter
    def foo(self, value):
        self._foo = value
```

This syntax looks pretty odd, although the intent is obvious. First, we decorate the `foo` method as a getter. Then, we decorate a second method with exactly the same name by applying the `setter` attribute of the originally decorated `foo` method! The `property` function returns an object; this object always comes with its own `setter` attribute, which can then be applied as a decorator to other functions. Using the same name for the get and set methods is not required, but it does help group the multiple methods that access one property together.

We can also specify a deletion function with `@foo.deleter`. We cannot specify a docstring using `property` decorators, so we need to rely on the property copying the docstring from the initial getter method.

Here's our previous `Silly` class rewritten to use `property` as a decorator:

```
class Silly:
    @property
    def silly(self):
        "This is a silly property"
        print("You are getting silly")
        return self._silly

    @silly.setter
    def silly(self, value):
        print("You are making silly {}".format(value))
        self._silly = value

    @silly.deleter
    def silly(self):
        print("Whoah, you killed silly!")
        del self._silly
```

This class operates *exactly* the same as our earlier version, including the help text. You can use whichever syntax you feel is more readable and elegant.

Deciding when to use properties

With the property built-in clouding the division between behavior and data, it can be confusing to know which one to choose. The example use case we saw earlier is one of the most common uses of properties; we have some data on a class that we later want to add behavior to. There are also other factors to take into account when deciding to use a property.

Technically, in Python, data, properties, and methods are all attributes on a class. The fact that a method is callable does not distinguish it from other types of attributes; indeed, we'll see in *Chapter 7*, *Python Object-oriented Shortcuts*, that it is possible to create normal objects that can be called like functions. We'll also discover that functions and methods are themselves normal objects.

The fact that methods are just callable attributes, and properties are just customizable attributes can help us make this decision. Methods should typically represent actions; things that can be done to, or performed by, the object. When you call a method, even with only one argument, it should *do* something. Method names are generally verbs.

Once confirming that an attribute is not an action, we need to decide between standard data attributes and properties. In general, always use a standard attribute until you need to control access to that property in some way. In either case, your attribute is usually a noun. The only difference between an attribute and a property is that we can invoke custom actions automatically when a property is retrieved, set, or deleted.

Let's look at a more realistic example. A common need for custom behavior is caching a value that is difficult to calculate or expensive to look up (requiring, for example, a network request or database query). The goal is to store the value locally to avoid repeated calls to the expensive calculation.

We can do this with a custom getter on the property. The first time the value is retrieved, we perform the lookup or calculation. Then we could locally cache the value as a private attribute on our object (or in dedicated caching software), and the next time the value is requested, we return the stored data. Here's how we might cache a web page:

```
from urllib.request import urlopen

class WebPage:
    def __init__(self, url):
        self.url = url
        self._content = None

    @property
```

```
        def content(self):
            if not self._content:
                print("Retrieving New Page...")
                self._content = urlopen(self.url).read()
            return self._content
```

We can test this code to see that the page is only retrieved once:

```
>>> import time
>>> webpage = WebPage("http://ccphillips.net/")
>>> now = time.time()
>>> content1 = webpage.content
Retrieving New Page...
>>> time.time() - now
22.43316888809204
>>> now = time.time()
>>> content2 = webpage.content
>>> time.time() - now
1.9266459941864014
>>> content2 == content1
True
```

I was on an awful satellite connection when I originally tested this code and it took 20 seconds the first time I loaded the content. The second time, I got the result in 2 seconds (which is really just the amount of time it took to type the lines into the interpreter).

Custom getters are also useful for attributes that need to be calculated on the fly, based on other object attributes. For example, we might want to calculate the average for a list of integers:

```
    class AverageList(list):
        @property
        def average(self):
            return sum(self) / len(self)
```

This very simple class inherits from `list`, so we get list-like behavior for free. We just add a property to the class, and presto, our list can have an average:

```
>>> a = AverageList([1,2,3,4])
>>> a.average
2.5
```

Of course, we could have made this a method instead, but then we should call it `calculate_average()`, since methods represent actions. But a property called `average` is more suitable, both easier to type, and easier to read.

Custom setters are useful for validation, as we've already seen, but they can also be used to proxy a value to another location. For example, we could add a content setter to the `WebPage` class that automatically logs into our web server and uploads a new page whenever the value is set.

Manager objects

We've been focused on objects and their attributes and methods. Now, we'll take a look at designing higher-level objects: the kinds of objects that manage other objects. The objects that tie everything together.

The difference between these objects and most of the examples we've seen so far is that our examples tend to represent concrete ideas. Management objects are more like office managers; they don't do the actual "visible" work out on the floor, but without them, there would be no communication between departments and nobody would know what they are supposed to do (although, this can be true anyway if the organization is badly managed!). Analogously, the attributes on a management class tend to refer to other objects that do the "visible" work; the behaviors on such a class delegate to those other classes at the right time, and pass messages between them.

As an example, we'll write a program that does a find and replace action for text files stored in a compressed ZIP file. We'll need objects to represent the ZIP file and each individual text file (luckily, we don't have to write these classes, they're available in the Python standard library). The manager object will be responsible for ensuring three steps occur in order:

1. Unzipping the compressed file.
2. Performing the find and replace action.
3. Zipping up the new files.

The class is initialized with the `.zip` filename and search and replace strings. We create a temporary directory to store the unzipped files in, so that the folder stays clean. The Python 3.4 `pathlib` library helps out with file and directory manipulation. We'll learn more about that in *Chapter 8*, *Strings and Serialization*, but the interface should be pretty clear in the following example:

```
import sys
import shutil
import zipfile
```

```
from pathlib import Path

class ZipReplace:
    def __init__(self, filename, search_string, replace_string):
        self.filename = filename
        self.search_string = search_string
        self.replace_string = replace_string
        self.temp_directory = Path("unzipped-{}".format(
                filename))
```

Then, we create an overall "manager" method for each of the three steps. This method delegates responsibility to other methods. Obviously, we could do all three steps in one method, or indeed, in one script without ever creating an object. There are several advantages to separating the three steps:

- **Readability**: The code for each step is in a self-contained unit that is easy to read and understand. The method names describe what the method does, and less additional documentation is required to understand what is going on.
- **Extensibility**: If a subclass wanted to use compressed TAR files instead of ZIP files, it could override the `zip` and `unzip` methods without having to duplicate the `find_replace` method.
- **Partitioning**: An external class could create an instance of this class and call the `find_replace` method directly on some folder without having to `zip` the content.

The delegation method is the first in the following code; the rest of the methods are included for completeness:

```
    def zip_find_replace(self):
        self.unzip_files()
        self.find_replace()
        self.zip_files()

    def unzip_files(self):
        self.temp_directory.mkdir()
        with zipfile.ZipFile(self.filename) as zip:
            zip.extractall(str(self.temp_directory))

    def find_replace(self):
        for filename in self.temp_directory.iterdir():
            with filename.open() as file:
                contents = file.read()
            contents = contents.replace(
                    self.search_string, self.replace_string)
```

```
                with filename.open("w") as file:
                    file.write(contents)

        def zip_files(self):
            with zipfile.ZipFile(self.filename, 'w') as file:
                for filename in self.temp_directory.iterdir():
                    file.write(str(filename), filename.name)
            shutil.rmtree(str(self.temp_directory))

    if __name__ == "__main__":
        ZipReplace(*sys.argv[1:4]).zip_find_replace()
```

For brevity, the code for zipping and unzipping files is sparsely documented. Our current focus is on object-oriented design; if you are interested in the inner details of the `zipfile` module, refer to the documentation in the standard library, either online or by typing `import zipfile ; help(zipfile)` into your interactive interpreter. Note that this example only searches the top-level files in a ZIP file; if there are any folders in the unzipped content, they will not be scanned, nor will any files inside those folders.

The last two lines in the example allow us to run the program from the command line by passing the `zip` filename, search string, and replace string as arguments:

```
python zipsearch.py hello.zip hello hi
```

Of course, this object does not have to be created from the command line; it could be imported from another module (to perform batch ZIP file processing) or accessed as part of a GUI interface or even a higher-level management object that knows where to get ZIP files (for example, to retrieve them from an FTP server or back them up to an external disk).

As programs become more and more complex, the objects being modeled become less and less like physical objects. Properties are other abstract objects and methods are actions that change the state of those abstract objects. But at the heart of every object, no matter how complex, is a set of concrete properties and well-defined behaviors.

Removing duplicate code

Often the code in management style classes such as `ZipReplace` is quite generic and can be applied in a variety of ways. It is possible to use either composition or inheritance to help keep this code in one place, thus eliminating duplicate code. Before we look at any examples of this, let's discuss a tiny bit of theory. Specifically, why is duplicate code a bad thing?

There are several reasons, but they all boil down to readability and maintainability. When we're writing a new piece of code that is similar to an earlier piece, the easiest thing to do is copy the old code and change whatever needs to be changed (variable names, logic, comments) to make it work in the new location. Alternatively, if we're writing new code that seems similar, but not identical to code elsewhere in the project, it is often easier to write fresh code with similar behavior, rather than figure out how to extract the overlapping functionality.

But as soon as someone has to read and understand the code and they come across duplicate blocks, they are faced with a dilemma. Code that might have made sense suddenly has to be understood. How is one section different from the other? How are they the same? Under what conditions is one section called? When do we call the other? You might argue that you're the only one reading your code, but if you don't touch that code for eight months it will be as incomprehensible to you as it is to a fresh coder. When we're trying to read two similar pieces of code, we have to understand why they're different, as well as how they're different. This wastes the reader's time; code should always be written to be readable first.

I once had to try to understand someone's code that had three identical copies of the same 300 lines of very poorly written code. I had been working with the code for a month before I finally comprehended that the three "identical" versions were actually performing slightly different tax calculations. Some of the subtle differences were intentional, but there were also obvious areas where someone had updated a calculation in one function without updating the other two. The number of subtle, incomprehensible bugs in the code could not be counted. I eventually replaced all 900 lines with an easy-to-read function of 20 lines or so.

Reading such duplicate code can be tiresome, but code maintenance is even more tormenting. As the preceding story suggests, keeping two similar pieces of code up to date can be a nightmare. We have to remember to update both sections whenever we update one of them, and we have to remember how the multiple sections differ so we can modify our changes when we are editing each of them. If we forget to update both sections, we will end up with extremely annoying bugs that usually manifest themselves as, "but I fixed that already, why is it still happening?"

The result is that people who are reading or maintaining our code have to spend astronomical amounts of time understanding and testing it compared to if we had written the code in a nonrepetitive manner in the first place. It's even more frustrating when we are the ones doing the maintenance; we find ourselves saying, "why didn't I do this right the first time?" The time we save by copy-pasting existing code is lost the very first time we have to maintain it. Code is both read and modified many more times and much more often than it is written. Comprehensible code should always be paramount.

This is why programmers, especially Python programmers (who tend to value elegant code more than average), follow what is known as the **Don't Repeat Yourself** (**DRY**) principle. DRY code is maintainable code. My advice to beginning programmers is to never use the copy and paste feature of their editor. To intermediate programmers, I suggest they think thrice before they hit *Ctrl* + *C*.

But what should we do instead of code duplication? The simplest solution is often to move the code into a function that accepts parameters to account for whatever parts are different. This isn't a terribly object-oriented solution, but it is frequently optimal.

For example, if we have two pieces of code that unzip a ZIP file into two different directories, we can easily write a function that accepts a parameter for the directory to which it should be unzipped instead. This may make the function itself slightly more difficult to read, but a good function name and docstring can easily make up for that, and any code that invokes the function will be easier to read.

That's certainly enough theory! The moral of the story is: always make the effort to refactor your code to be easier to read instead of writing bad code that is only easier to write.

In practice

Let's explore two ways we can reuse existing code. After writing our code to replace strings in a ZIP file full of text files, we are later contracted to scale all the images in a ZIP file to 640 x 480. Looks like we could use a very similar paradigm to what we used in `ZipReplace`. The first impulse might be to save a copy of that file and change the `find_replace` method to `scale_image` or something similar.

But, that's uncool. What if someday we want to change the `unzip` and `zip` methods to also open TAR files? Or maybe we want to use a guaranteed unique directory name for temporary files. In either case, we'd have to change it in two different places!

We'll start by demonstrating an inheritance-based solution to this problem. First we'll modify our original `ZipReplace` class into a superclass for processing generic ZIP files:

```
import os
import shutil
import zipfile
from pathlib import Path


class ZipProcessor:
    def __init__(self, zipname):
        self.zipname = zipname
```

```
            self.temp_directory = Path("unzipped-{}".format(
                    zipname[:-4]))

        def process_zip(self):
            self.unzip_files()
            self.process_files()
            self.zip_files()

        def unzip_files(self):
            self.temp_directory.mkdir()
            with zipfile.ZipFile(self.zipname) as zip:
                zip.extractall(str(self.temp_directory))

        def zip_files(self):
            with zipfile.ZipFile(self.zipname, 'w') as file:
                for filename in self.temp_directory.iterdir():
                    file.write(str(filename), filename.name)
            shutil.rmtree(str(self.temp_directory))
```

We changed the `filename` property to `zipname` to avoid confusion with the `filename` local variables inside the various methods. This helps make the code more readable even though it isn't actually a change in design.

We also dropped the two parameters to `__init__` (`search_string` and `replace_string`) that were specific to `ZipReplace`. Then we renamed the `zip_find_replace` method to `process_zip` and made it call an (as yet undefined) `process_files` method instead of `find_replace`; these name changes help demonstrate the more generalized nature of our new class. Notice that we have removed the `find_replace` method altogether; that code is specific to `ZipReplace` and has no business here.

This new `ZipProcessor` class doesn't actually define a `process_files` method; so if we ran it directly, it would raise an exception. Because it isn't meant to run directly, we removed the main call at the bottom of the original script.

Now, before we move on to our image processing app, let's fix up our original `zipsearch` class to make use of this parent class:

```
from zip_processor import ZipProcessor
import sys
import os

class ZipReplace(ZipProcessor):
    def __init__(self, filename, search_string,
            replace_string):
        super().__init__(filename)
```

```
            self.search_string = search_string
            self.replace_string = replace_string

        def process_files(self):
            '''perform a search and replace on all files in the
            temporary directory'''
            for filename in self.temp_directory.iterdir():
                with filename.open() as file:
                    contents = file.read()
                contents = contents.replace(
                        self.search_string, self.replace_string)
                with filename.open("w") as file:
                    file.write(contents)

    if __name__ == "__main__":
        ZipReplace(*sys.argv[1:4]).process_zip()
```

This code is a bit shorter than the original version, since it inherits its ZIP processing abilities from the parent class. We first import the base class we just wrote and make `ZipReplace` extend that class. Then we use `super()` to initialize the parent class. The `find_replace` method is still here, but we renamed it to `process_files` so the parent class can call it from its management interface. Because this name isn't as descriptive as the old one, we added a docstring to describe what it is doing.

Now, that was quite a bit of work, considering that all we have now is a program that is functionally not different from the one we started with! But having done that work, it is now much easier for us to write other classes that operate on files in a ZIP archive, such as the (hypothetically requested) photo scaler. Further, if we ever want to improve or bug fix the zip functionality, we can do it for all classes by changing only the one `ZipProcessor` base class. Maintenance will be much more effective.

See how simple it is now to create a photo scaling class that takes advantage of the `ZipProcessor` functionality. (Note: this class requires the third-party `pillow` library to get the `PIL` module. You can install it with `pip install pillow`.)

```
from zip_processor import ZipProcessor
import sys
from PIL import Image

class ScaleZip(ZipProcessor):

    def process_files(self):
        '''Scale each image in the directory to 640x480'''
        for filename in self.temp_directory.iterdir():
            im = Image.open(str(filename))
```

```
            scaled = im.resize((640, 480))
            scaled.save(str(filename))

if __name__ == "__main__":
    ScaleZip(*sys.argv[1:4]).process_zip()
```

Look how simple this class is! All that work we did earlier paid off. All we do is open each file (assuming that it is an image; it will unceremoniously crash if a file cannot be opened), scale it, and save it back. The `ZipProcessor` class takes care of the zipping and unzipping without any extra work on our part.

Case study

For this case study, we'll try to delve further into the question, "when should I choose an object versus a built-in type?" We'll be modeling a `Document` class that might be used in a text editor or word processor. What objects, functions, or properties should it have?

We might start with a `str` for the `Document` contents, but in Python, strings aren't mutable (able to be changed). Once a `str` is defined, it is forever. We can't insert a character into it or remove one without creating a brand new string object. That would be leaving a lot of `str` objects taking up memory until Python's garbage collector sees fit to clean up behind us.

So, instead of a string, we'll use a list of characters, which we can modify at will. In addition, a `Document` class would need to know the current cursor position within the list, and should probably also store a filename for the document.

Real text editors use a binary-tree based data structure called a `rope` to model their document contents. This book's title isn't "advanced data structures", so if you're interested in learning more about this fascinating topic, you may want to search the web for the rope data structure.

Now, what methods should it have? There are a lot of things we might want to do to a text document, including inserting, deleting, and selecting characters, cut, copy, paste, the selection, and saving or closing the document. It looks like there are copious amounts of both data and behavior, so it makes sense to put all this stuff into its own `Document` class.

A pertinent question is: should this class be composed of a bunch of basic Python objects such as `str` filenames, `int` cursor positions, and a `list` of characters? Or should some or all of those things be specially defined objects in their own right? What about individual lines and characters, do they need to have classes of their own?

We'll answer these questions as we go, but let's start with the simplest possible `Document` class first and see what it can do:

```
class Document:
    def __init__(self):
        self.characters = []
        self.cursor = 0
        self.filename = ''

    def insert(self, character):
        self.characters.insert(self.cursor, character)
        self.cursor += 1

    def delete(self):
        del self.characters[self.cursor]

    def save(self):
        with open(self.filename, 'w') as f:
            f.write(''.join(self.characters))

    def forward(self):
        self.cursor += 1

    def back(self):
        self.cursor -= 1
```

This simple class allows us full control over editing a basic document. Have a look at it in action:

```
>>> doc = Document()
>>> doc.filename = "test_document"
>>> doc.insert('h')
>>> doc.insert('e')
>>> doc.insert('l')
>>> doc.insert('l')
>>> doc.insert('o')
>>> "".join(doc.characters)
'hello'
>>> doc.back()
>>> doc.delete()
>>> doc.insert('p')
>>> "".join(doc.characters)
'hellp'
```

Looks like it's working. We could connect a keyboard's letter and arrow keys to these methods and the document would track everything just fine.

But what if we want to connect more than just arrow keys. What if we want to connect the *Home* and *End* keys as well? We could add more methods to the `Document` class that search forward or backwards for newline characters (in Python, a newline character, or `\n` represents the end of one line and the beginning of a new one) in the string and jump to them, but if we did that for every possible movement action (move by words, move by sentences, *Page Up*, *Page Down*, end of line, beginning of whitespace, and more), the class would be huge. Maybe it would be better to put those methods on a separate object. So, let us turn the cursor attribute into an object that is aware of its position and can manipulate that position. We can move the forward and back methods to that class, and add a couple more for the *Home* and *End* keys:

```
class Cursor:
    def __init__(self, document):
        self.document = document
        self.position = 0

    def forward(self):
        self.position += 1

    def back(self):
        self.position -= 1

    def home(self):
        while self.document.characters[
                self.position-1] != '\n':
            self.position -= 1
            if self.position == 0:
                # Got to beginning of file before newline
                break

    def end(self):
        while self.position < len(self.document.characters
                ) and self.document.characters[
                    self.position] != '\n':
            self.position += 1
```

This class takes the document as an initialization parameter so the methods have access to the content of the document's character list. It then provides simple methods for moving backwards and forwards, as before, and for moving to the `home` and `end` positions.

This code is not very safe. You can very easily move past the ending position, and if you try to go home on an empty file, it will crash. These examples are kept short to make them readable, but that doesn't mean they are defensive! You can improve the error checking of this code as an exercise; it might be a great opportunity to expand your exception handling skills.

The `Document` class itself is hardly changed, except for removing the two methods that were moved to the `Cursor` class:

```
class Document:
    def __init__(self):
        self.characters = []
        self.cursor = Cursor(self)
        self.filename = ''

       def insert(self, character):
        self.characters.insert(self.cursor.position,
                character)
        self.cursor.forward()

    def delete(self):
        del self.characters[self.cursor.position]

    def save(self):
        f = open(self.filename, 'w')
        f.write(''.join(self.characters))
        f.close()
```

We simply updated anything that accessed the old cursor integer to use the new object instead. We can test that the `home` method is really moving to the newline character:

```
>>> d = Document()
>>> d.insert('h')
>>> d.insert('e')
>>> d.insert('l')
>>> d.insert('l')
>>> d.insert('o')
>>> d.insert('\n')
>>> d.insert('w')
```

```
>>> d.insert('o')
>>> d.insert('r')
>>> d.insert('l')
>>> d.insert('d')
>>> d.cursor.home()
>>> d.insert("*")
>>> print("".join(d.characters))
hello
*world
```

Now, since we've been using that string `join` function a lot (to concatenate the characters so we can see the actual document contents), we can add a property to the `Document` class to give us the complete string:

```
    @property
    def string(self):
        return "".join(self.characters)
```

This makes our testing a little simpler:

```
>>> print(d.string)
hello
world
```

This framework is simple (though it might be a bit time consuming!) to extend to create and edit a complete plaintext document. Now, let's extend it to work for rich text; text that can have **bold**, underlined, or *italic* characters.

There are two ways we could process this; the first is to insert "fake" characters into our character list that act like instructions, such as "bold characters until you find a stop bold character". The second is to add information to each character indicating what formatting it should have. While the former method is probably more common, we'll implement the latter solution. To do that, we're obviously going to need a class for characters. This class will have an attribute representing the character, as well as three Boolean attributes representing whether it is bold, italic, or underlined.

Hmm, wait! Is this `Character` class going to have any methods? If not, maybe we should use one of the many Python data structures instead; a tuple or named tuple would probably be sufficient. Are there any actions that we would want to do to, or invoke on a character?

Well, clearly, we might want to do things with characters, such as delete or copy them, but those are things that need to be handled at the `Document` level, since they are really modifying the list of characters. Are there things that need to be done to individual characters?

Actually, now that we're thinking about what a `Character` class actually is... what is it? Would it be safe to say that a `Character` class is a string? Maybe we should use an inheritance relationship here? Then we can take advantage of the numerous methods that `str` instances come with.

What sorts of methods are we talking about? There's `startswith`, `strip`, `find`, `lower`, and many more. Most of these methods expect to be working on strings that contain more than one character. In contrast, if `Character` were to subclass `str`, we'd probably be wise to override `__init__` to raise an exception if a multi-character string were supplied. Since all those methods we'd get for free wouldn't really apply to our `Character` class, it seems we needn't use inheritance, after all.

This brings us back to our original question; should `Character` even be a class? There is a very important special method on the `object` class that we can take advantage of to represent our characters. This method, called `__str__` (two underscores, like `__init__`), is used in string manipulation functions like `print` and the `str` constructor to convert any class to a string. The default implementation does some boring stuff like printing the name of the module and class and its address in memory. But if we override it, we can make it print whatever we like. For our implementation, we could make it prefix characters with special characters to represent whether they are bold, italic, or underlined. So, we will create a class to represent a character, and here it is:

```
class Character:
    def __init__(self, character,
                 bold=False, italic=False, underline=False):
        assert len(character) == 1
        self.character = character
        self.bold = bold
        self.italic = italic
        self.underline = underline

    def __str__(self):
        bold = "*" if self.bold else ''
        italic = "/" if self.italic else ''
        underline = "_" if self.underline else ''
        return bold + italic + underline + self.character
```

This class allows us to create characters and prefix them with a special character when the `str()` function is applied to them. Nothing too exciting there. We only have to make a few minor modifications to the `Document` and `Cursor` classes to work with this class. In the `Document` class, we add these two lines at the beginning of the `insert` method:

```
def insert(self, character):
    if not hasattr(character, 'character'):
        character = Character(character)
```

This is a rather strange bit of code. Its basic purpose is to check whether the character being passed in is a `Character` or a `str`. If it is a string, it is wrapped in a `Character` class so all objects in the list are `Character` objects. However, it is entirely possible that someone using our code would want to use a class that is neither `Character` nor string, using duck typing. If the object has a character attribute, we assume it is a "`Character`-like" object. But if it does not, we assume it is a "`str`-like" object and wrap it in `Character`. This helps the program take advantage of duck typing as well as polymorphism; as long as an object has a character attribute, it can be used in the `Document` class.

This generic check could be very useful, for example, if we wanted to make a programmer's editor with syntax highlighting: we'd need extra data on the character, such as what type of syntax token the character belongs to. Note that if we are doing a lot of this kind of comparison, it's probably better to implement `Character` as an abstract base class with an appropriate `__subclasshook__`, as discussed in *Chapter 3*, *When Objects Are Alike*.

In addition, we need to modify the string property on `Document` to accept the new `Character` values. All we need to do is call `str()` on each character before we join it:

```
@property
def string(self):
    return "".join((str(c) for c in self.characters))
```

This code uses a generator expression, which we'll discuss in *Chapter 9*, *The Iterator Pattern*. It's a shortcut to perform a specific action on all the objects in a sequence.

Finally, we also need to check `Character.character`, instead of just the string character we were storing before, in the `home` and `end` functions when we're looking to see whether it matches a newline character:

```
def home(self):
    while self.document.characters[
            self.position-1].character != '\n':
        self.position -= 1
        if self.position == 0:
```

```
                # Got to beginning of file before newline
                break

    def end(self):
        while self.position < len(
                self.document.characters) and \
                self.document.characters[
                        self.position
                        ].character != '\n':
            self.position += 1
```

This completes the formatting of characters. We can test it to see that it works:

```
>>> d = Document()
>>> d.insert('h')
>>> d.insert('e')
>>> d.insert(Character('l', bold=True))
>>> d.insert(Character('l', bold=True))
>>> d.insert('o')
>>> d.insert('\n')
>>> d.insert(Character('w', italic=True))
>>> d.insert(Character('o', italic=True))
>>> d.insert(Character('r', underline=True))
>>> d.insert('l')
>>> d.insert('d')
>>> print(d.string)
he*l*lo
/w/o_rld
>>> d.cursor.home()
>>> d.delete()
>>> d.insert('W')
>>> print(d.string)
he*l*lo
W/o_rld
>>> d.characters[0].underline = True
>>> print(d.string)
_he*l*lo
W/o_rld
```

As expected, whenever we print the string, each bold character is preceded by a `*` character, each italic character by a `/` character, and each underlined character by a `_` character. All our functions seem to work, and we can modify characters in the list after the fact. We have a working rich text document object that could be plugged into a proper user interface and hooked up with a keyboard for input and a screen for output. Naturally, we'd want to display real bold, italic, and underlined characters on the screen, instead of using our `__str__` method, but it was sufficient for the basic testing we demanded of it.

Exercises

We've looked at various ways that objects, data, and methods can interact with each other in an object-oriented Python program. As usual, your first thoughts should be how you can apply these principles to your own work. Do you have any messy scripts lying around that could be rewritten using an object-oriented manager? Look through some of your old code and look for methods that are not actions. If the name isn't a verb, try rewriting it as a property.

Think about code you've written in any language. Does it break the DRY principle? Is there any duplicate code? Did you copy and paste code? Did you write two versions of similar pieces of code because you didn't feel like understanding the original code? Go back over some of your recent code now and see whether you can refactor the duplicate code using inheritance or composition. Try to pick a project you're still interested in maintaining; not code so old that you never want to touch it again. It helps keep your interest up when you do the improvements!

Now, look back over some of the examples we saw in this chapter. Start with the cached web page example that uses a property to cache the retrieved data. An obvious problem with this example is that the cache is never refreshed. Add a timeout to the property's getter, and only return the cached page if the page has been requested before the timeout has expired. You can use the `time` module (`time.time() - an_old_time` returns the number of seconds that have elapsed since `an_old_time`) to determine whether the cache has expired.

Now look at the inheritance-based `ZipProcessor`. It might be reasonable to use composition instead of inheritance here. Instead of extending the class in the `ZipReplace` and `ScaleZip` classes, you could pass instances of those classes into the `ZipProcessor` constructor and call them to do the processing part. Implement this.

Which version do you find easier to use? Which is more elegant? What is easier to read? These are subjective questions; the answer varies for each of us. Knowing the answer, however, is important; if you find you prefer inheritance over composition, you have to pay attention that you don't overuse inheritance in your daily coding. If you prefer composition, make sure you don't miss opportunities to create an elegant inheritance-based solution.

Finally, add some error handlers to the various classes we created in the case study. They should ensure single characters are entered, that you don't try to move the cursor past the end or beginning of the file, that you don't delete a character that doesn't exist, and that you don't save a file without a filename. Try to think of as many edge cases as you can, and account for them (thinking about edge cases is about 90 percent of a professional programmer's job!) Consider different ways to handle them; should you raise an exception when the user tries to move past the end of the file, or just stay on the last character?

Pay attention, in your daily coding, to the copy and paste commands. Every time you use them in your editor, consider whether it would be a good idea to improve your program's organization so that you only have one version of the code you are about to copy.

Summary

In this chapter, we focused on identifying objects, especially objects that are not immediately apparent; objects that manage and control. Objects should have both data and behavior, but properties can be used to blur the distinction between the two. The DRY principle is an important indicator of code quality and inheritance and composition can be applied to reduce code duplication.

In the next chapter, we'll cover several of the built-in Python data structures and objects, focusing on their object-oriented properties and how they can be extended or adapted.

6
Python Data Structures

In our examples so far, we've already seen many of the built-in Python data structures in action. You've probably also covered many of them in introductory books or tutorials. In this chapter, we'll be discussing the object-oriented features of these data structures, when they should be used instead of a regular class, and when they should not be used. In particular, we'll be covering:

- Tuples and named tuples
- Dictionaries
- Lists and sets
- How and why to extend built-in objects
- Three types of queues

Empty objects

Let's start with the most basic Python built-in, one that we've seen many times already, the one that we've extended in every class we have created: the `object`. Technically, we can instantiate an `object` without writing a subclass:

```
>>> o = object()
>>> o.x = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'object' object has no attribute 'x'
```

Unfortunately, as you can see, it's not possible to set any attributes on an `object` that was instantiated directly. This isn't because the Python developers wanted to force us to write our own classes, or anything so sinister. They did this to save memory; a lot of memory. When Python allows an object to have arbitrary attributes, it takes a certain amount of system memory to keep track of what attributes each object has, for storing both the attribute name and its value. Even if no attributes are stored, memory is allocated for *potential* new attributes. Given the dozens, hundreds, or thousands of objects (every class extends object) in a typical Python program; this small amount of memory would quickly become a large amount of memory. So, Python disables arbitrary properties on `object`, and several other built-ins, by default.

It is possible to restrict arbitrary properties on our own classes using **slots**. Slots are beyond the scope of this book, but you now have a search term if you are looking for more information. In normal use, there isn't much benefit to using slots, but if you're writing an object that will be duplicated thousands of times throughout the system, they can help save memory, just as they do for `object`.

It is, however, trivial to create an empty object class of our own; we saw it in our earliest example:

```
class MyObject:
    pass
```

And, as we've already seen, it's possible to set attributes on such classes:

```
>>> m = MyObject()
>>> m.x = "hello"
>>> m.x
'hello'
```

If we wanted to group properties together, we could store them in an empty object like this. But we are usually better off using other built-ins designed for storing data. It has been stressed throughout this book that classes and objects should only be used when you want to specify *both* data and behaviors. The main reason to write an empty class is to quickly block something out, knowing we'll come back later to add behavior. It is much easier to adapt behaviors to a class than it is to replace a data structure with an object and change all references to it. Therefore, it is important to decide from the outset if the data is just data, or if it is an object in disguise. Once that design decision is made, the rest of the design naturally falls into place.

Tuples and named tuples

Tuples are objects that can store a specific number of other objects in order. They are immutable, so we can't add, remove, or replace objects on the fly. This may seem like a massive restriction, but the truth is, if you need to modify a tuple, you're using the wrong data type (usually a list would be more suitable). The primary benefit of tuples' immutability is that we can use them as keys in dictionaries, and in other locations where an object requires a hash value.

Tuples are used to store data; behavior cannot be stored in a tuple. If we require behavior to manipulate a tuple, we have to pass the tuple into a function (or method on another object) that performs the action.

Tuples should generally store values that are somehow different from each other. For example, we would not put three stock symbols in a tuple, but we might create a tuple of stock symbol, current price, high, and low for the day. The primary purpose of a tuple is to aggregate different pieces of data together into one container. Thus, a tuple can be the easiest tool to replace the "object with no data" idiom.

We can create a tuple by separating the values with a comma. Usually, tuples are wrapped in parentheses to make them easy to read and to separate them from other parts of an expression, but this is not always mandatory. The following two assignments are identical (they record a stock, the current price, the high, and the low for a rather profitable company):

```
>>> stock = "FB", 75.00, 75.03, 74.90
>>> stock2 = ("FB", 75.00, 75.03, 74.90)
```

If we're grouping a tuple inside of some other object, such as a function call, list comprehension, or generator, the parentheses are required. Otherwise, it would be impossible for the interpreter to know whether it is a tuple or the next function parameter. For example, the following function accepts a tuple and a date, and returns a tuple of the date and the middle value between the stock's high and low value:

```
import datetime
def middle(stock, date):
    symbol, current, high, low = stock
    return (((high + low) / 2), date)

mid_value, date = middle(("FB", 75.00, 75.03, 74.90),
        datetime.date(2014, 10, 31))
```

The tuple is created directly inside the function call by separating the values with commas and enclosing the entire tuple in parenthesis. This tuple is then followed by a comma to separate it from the second argument.

This example also illustrates tuple unpacking. The first line inside the function unpacks the `stock` parameter into four different variables. The tuple has to be exactly the same length as the number of variables, or it will raise an exception. We can also see an example of tuple unpacking on the last line, where the tuple returned inside the function is unpacked into two values, `mid_value` and `date`. Granted, this is a strange thing to do, since we supplied the date to the function in the first place, but it gave us a chance to see unpacking at work.

Unpacking is a very useful feature in Python. We can group variables together to make storing and passing them around simpler, but the moment we need to access all of them, we can unpack them into separate variables. Of course, sometimes we only need access to one of the variables in the tuple. We can use the same syntax that we use for other sequence types (lists and strings, for example) to access an individual value:

```
>>> stock = "FB", 75.00, 75.03, 74.90
>>> high = stock[2]
>>> high
75.03
```

We can even use slice notation to extract larger pieces of tuples:

```
>>> stock[1:3]
(75.00, 75.03)
```

These examples, while illustrating how flexible tuples can be, also demonstrate one of their major disadvantages: readability. How does someone reading this code know what is in the second position of a specific tuple? They can guess, from the name of the variable we assigned it to, that it is `high` of some sort, but if we had just accessed the tuple value in a calculation without assigning it, there would be no such indication. They would have to paw through the code to find where the tuple was declared before they could discover what it does.

Accessing tuple members directly is fine in some circumstances, but don't make a habit of it. Such so-called "magic numbers" (numbers that seem to come out of thin air with no apparent meaning within the code) are the source of many coding errors and lead to hours of frustrated debugging. Try to use tuples only when you know that all the values are going to be useful at once and it's normally going to be unpacked when it is accessed. If you have to access a member directly or using a slice and the purpose of that value is not immediately obvious, at least include a comment explaining where it came from.

Named tuples

So, what do we do when we want to group values together, but know we're frequently going to need to access them individually? Well, we could use an empty object, as discussed in the previous section (but that is rarely useful unless we anticipate adding behavior later), or we could use a dictionary (most useful if we don't know exactly how many or which specific data will be stored), as we'll cover in the next section.

If, however, we do not need to add behavior to the object, and we know in advance what attributes we need to store, we can use a named tuple. Named tuples are tuples with attitude. They are a great way to group read-only data together.

Constructing a named tuple takes a bit more work than a normal tuple. First, we have to import `namedtuple`, as it is not in the namespace by default. Then, we describe the named tuple by giving it a name and outlining its attributes. This returns a class-like object that we can instantiate with the required values as many times as we want:

```
from collections import namedtuple
Stock = namedtuple("Stock", "symbol current high low")
stock = Stock("FB", 75.00, high=75.03, low=74.90)
```

The `namedtuple` constructor accepts two arguments. The first is an identifier for the named tuple. The second is a string of space-separated attributes that the named tuple can have. The first attribute should be listed, followed by a space (or comma if you prefer), then the second attribute, then another space, and so on. The result is an object that can be called just like a normal class to instantiate other objects. The constructor must have exactly the right number of arguments that can be passed in as arguments or keyword arguments. As with normal objects, we can create as many instances of this "class" as we like, with different values for each.

The resulting `namedtuple` can then be packed, unpacked, and otherwise treated like a normal tuple, but we can also access individual attributes on it as if it were an object:

```
>>> stock.high
75.03
>>> symbol, current, high, low = stock
>>> current
75.00
```

Remember that creating named tuples is a two-step process. First, use `collections.namedtuple` to create a class, and then construct instances of that class.

Named tuples are perfect for many "data only" representations, but they are not ideal for all situations. Like tuples and strings, named tuples are immutable, so we cannot modify an attribute once it has been set. For example, the current value of my company's stock has gone down since we started this discussion, but we can't set the new value:

```
>>> stock.current = 74.98
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

If we need to be able to change stored data, a dictionary may be what we need instead.

Dictionaries

Dictionaries are incredibly useful containers that allow us to map objects directly to other objects. An empty object with attributes to it is a sort of dictionary; the names of the properties map to the property values. This is actually closer to the truth than it sounds; internally, objects normally represent attributes as a dictionary, where the values are properties or methods on the objects (see the `__dict__` attribute if you don't believe me). Even the attributes on a module are stored, internally, in a dictionary.

Dictionaries are extremely efficient at looking up a value, given a specific key object that maps to that value. They should always be used when you want to find one object based on some other object. The object that is being stored is called the **value**; the object that is being used as an index is called the **key**. We've already seen dictionary syntax in some of our previous examples.

Dictionaries can be created either using the `dict()` constructor or using the `{}` syntax shortcut. In practice, the latter format is almost always used. We can prepopulate a dictionary by separating the keys from the values using a colon, and separating the key value pairs using a comma.

For example, in a stock application, we would most often want to look up prices by the stock symbol. We can create a dictionary that uses stock symbols as keys, and tuples of current, high, and low as values like this:

```
stocks = {"GOOG": (613.30, 625.86, 610.50),
          "MSFT": (30.25, 30.70, 30.19)}
```

As we've seen in previous examples, we can then look up values in the dictionary by requesting a key inside square brackets. If the key is not in the dictionary, it will raise an exception:

```
>>> stocks["GOOG"]
(613.3, 625.86, 610.5)
>>> stocks["RIM"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'RIM'
```

We can, of course, catch the `KeyError` and handle it. But we have other options. Remember, dictionaries are objects, even if their primary purpose is to hold other objects. As such, they have several behaviors associated with them. One of the most useful of these methods is the `get` method; it accepts a key as the first parameter and an optional default value if the key doesn't exist:

```
>>> print(stocks.get("RIM"))
None
>>> stocks.get("RIM", "NOT FOUND")
'NOT FOUND'
```

For even more control, we can use the `setdefault` method. If the key is in the dictionary, this method behaves just like `get`; it returns the value for that key. Otherwise, if the key is not in the dictionary, it will not only return the default value we supply in the method call (just like `get` does), it will also set the key to that same value. Another way to think of it is that `setdefault` sets a value in the dictionary only if that value has not previously been set. Then it returns the value in the dictionary, either the one that was already there, or the newly provided default value.

```
>>> stocks.setdefault("GOOG", "INVALID")
(613.3, 625.86, 610.5)
>>> stocks.setdefault("BBRY", (10.50, 10.62, 10.39))
(10.50, 10.62, 10.39)
>>> stocks["BBRY"]
(10.50, 10.62, 10.39)
```

The `GOOG` stock was already in the dictionary, so when we tried to `setdefault` it to an invalid value, it just returned the value already in the dictionary. `BBRY` was not in the dictionary, so `setdefault` returned the default value and set the new value in the dictionary for us. We then check that the new stock is, indeed, in the dictionary.

Three other very useful dictionary methods are `keys()`, `values()`, and `items()`. The first two return an iterator over all the keys and all the values in the dictionary. We can use these like lists or in `for` loops if we want to process all the keys or values. The `items()` method is probably the most useful; it returns an iterator over tuples of `(key, value)` pairs for every item in the dictionary. This works great with tuple unpacking in a `for` loop to loop over associated keys and values. This example does just that to print each stock in the dictionary with its current value:

```
>>> for stock, values in stocks.items():
...     print("{} last value is {}".format(stock, values[0]))
...
GOOG last value is 613.3
BBRY last value is 10.50
MSFT last value is 30.25
```

Each key/value tuple is unpacked into two variables named `stock` and `values` (we could use any variable names we wanted, but these both seem appropriate) and then printed in a formatted string.

Notice that the stocks do not show up in the same order in which they were inserted. Dictionaries, due to the efficient algorithm (known as hashing) that is used to make key lookup so fast, are inherently unsorted.

So, there are numerous ways to retrieve data from a dictionary once it has been instantiated; we can use square brackets as index syntax, the `get` method, the `setdefault` method, or iterate over the `items` method, among others.

Finally, as you likely already know, we can set a value in a dictionary using the same indexing syntax we use to retrieve a value:

```
>>> stocks["GOOG"] = (597.63, 610.00, 596.28)
>>> stocks['GOOG']
(597.63, 610.0, 596.28)
```

Google's price is lower today, so I've updated the tuple value in the dictionary. We can use this index syntax to set a value for any key, regardless of whether the key is in the dictionary. If it is in the dictionary, the old value will be replaced with the new one; otherwise, a new key/value pair will be created.

We've been using strings as dictionary keys, so far, but we aren't limited to string keys. It is common to use strings as keys, especially when we're storing data in a dictionary to gather it together (instead of using an object with named properties). But we can also use tuples, numbers, or even objects we've defined ourselves as dictionary keys. We can even use different types of keys in a single dictionary:

```
random_keys = {}
random_keys["astring"] = "somestring"
random_keys[5] = "aninteger"
random_keys[25.2] = "floats work too"
random_keys[("abc", 123)] = "so do tuples"

class AnObject:
    def __init__(self, avalue):
        self.avalue = avalue

my_object = AnObject(14)
random_keys[my_object] = "We can even store objects"
my_object.avalue = 12
try:
    random_keys[[1,2,3]] = "we can't store lists though"
except:
    print("unable to store list\n")

for key, value in random_keys.items():
    print("{} has value {}".format(key, value))
```

This code shows several different types of keys we can supply to a dictionary. It also shows one type of object that cannot be used. We've already used lists extensively, and we'll be seeing many more details of them in the next section. Because lists can change at any time (by adding or removing items, for example), they cannot hash to a specific value.

Objects that are **hashable** basically have a defined algorithm that converts the object into a unique integer value for rapid lookup. This hash is what is actually used to look up values in a dictionary. For example, strings map to integers based on the characters in the string, while tuples combine hashes of the items inside the tuple. Any two objects that are somehow considered equal (like strings with the same characters or tuples with the same values) should have the same hash value, and the hash value for an object should never ever change. Lists, however, can have their contents changed, which would change their hash value (two lists should only be equal if their contents are the same). Because of this, they can't be used as dictionary keys. For the same reason, dictionaries cannot be used as keys into other dictionaries.

In contrast, there are no limits on the types of objects that can be used as dictionary values. We can use a string key that maps to a list value, for example, or we can have a nested dictionary as a value in another dictionary.

Dictionary use cases

Dictionaries are extremely versatile and have numerous uses. There are two major ways that dictionaries can be used. The first is dictionaries where all the keys represent different instances of similar objects; for example, our stock dictionary. This is an indexing system. We use the stock symbol as an index to the values. The values could even have been complicated self-defined objects that made buy and sell decisions or set a stop-loss, rather than our simple tuples.

The second design is dictionaries where each key represents some aspect of a single structure; in this case, we'd probably use a separate dictionary for each object, and they'd all have similar (though often not identical) sets of keys. This latter situation can often also be solved with named tuples. These should typically be used when we know exactly what attributes the data must store, and we know that all pieces of the data must be supplied at once (when the item is constructed). But if we need to create or change dictionary keys over time or we don't know exactly what the keys might be, a dictionary is more suitable.

Using defaultdict

We've seen how to use `setdefault` to set a default value if a key doesn't exist, but this can get a bit monotonous if we need to set a default value every time we look up a value. For example, if we're writing code that counts the number of times a letter occurs in a given sentence, we could do this:

```
def letter_frequency(sentence):
    frequencies = {}
    for letter in sentence:
        frequency = frequencies.setdefault(letter, 0)
        frequencies[letter] = frequency + 1
    return frequencies
```

Every time we access the dictionary, we need to check that it has a value already, and if not, set it to zero. When something like this needs to be done every time an empty key is requested, we can use a different version of the dictionary, called `defaultdict`:

```
from collections import defaultdict
def letter_frequency(sentence):
    frequencies = defaultdict(int)
```

```
    for letter in sentence:
        frequencies[letter] += 1
    return frequencies
```

This code looks like it couldn't possibly work. The `defaultdict` accepts a function in its constructor. Whenever a key is accessed that is not already in the dictionary, it calls that function, with no parameters, to create a default value.

In this case, the function it calls is `int`, which is the constructor for an integer object. Normally, integers are created simply by typing an integer number into our code, and if we do create one using the `int` constructor, we pass it the item we want to create (for example, to convert a string of digits into an integer). But if we call `int` without any arguments, it returns, conveniently, the number zero. In this code, if the letter doesn't exist in the `defaultdict`, the number zero is returned when we access it. Then we add one to this number to indicate we've found an instance of that letter, and the next time we find one, that number will be returned and we can increment the value again.

The `defaultdict` is useful for creating dictionaries of containers. If we want to create a dictionary of stock prices for the past 30 days, we could use a stock symbol as the key and store the prices in `list`; the first time we access the stock price, we would want it to create an empty list. Simply pass `list` into the `defaultdict`, and it will be called every time an empty key is accessed. We can do similar things with sets or even empty dictionaries if we want to associate one with a key.

Of course, we can also write our own functions and pass them into the `defaultdict`. Suppose we want to create a `defaultdict` where each new element contains a tuple of the number of items inserted into the dictionary at that time and an empty list to hold other things. Nobody knows why we would want to create such an object, but let's have a look:

```
from collections import defaultdict
num_items = 0
def tuple_counter():
    global num_items
    num_items += 1
    return (num_items, [])

d = defaultdict(tuple_counter)
```

When we run this code, we can access empty keys and insert into the list all in one statement:

```
>>> d = defaultdict(tuple_counter)
>>> d['a'][1].append("hello")
>>> d['b'][1].append('world')
```

```
>>> d
defaultdict(<function tuple_counter at 0x82f2c6c>,
{'a': (1, ['hello']), 'b': (2, ['world'])})
```

When we print `dict` at the end, we see that the counter really was working.

This example, while succinctly demonstrating how to create our own function for `defaultdict`, is not actually very good code; using a global variable means that if we created four different `defaultdict` segments that each used `tuple_counter`, it would count the number of entries in all dictionaries, rather than having a different count for each one. It would be better to create a class and pass a method on that class to `defaultdict`.

Counter

You'd think that you couldn't get much simpler than `defaultdict(int)`, but the "I want to count specific instances in an iterable" use case is common enough that the Python developers created a specific class for it. The previous code that counts characters in a string can easily be calculated in a single line:

```
from collections import Counter
def letter_frequency(sentence):
    return Counter(sentence)
```

The `Counter` object behaves like a beefed up dictionary where the keys are the items being counted and the values are the number of such items. One of the most useful functions is the `most_common()` method. It returns a list of (key, count) tuples ordered by the count. You can optionally pass an integer argument into `most_common()` to request only the top most common elements. For example, you could write a simple polling application as follows:

```
from collections import Counter

responses = [
    "vanilla",
    "chocolate",
    "vanilla",
    "vanilla",
    "caramel",
    "strawberry",
    "vanilla"
]

print(
```

```
        "The children voted for {} ice cream".format(
            Counter(responses).most_common(1)[0][0]
        )
    )
```

Presumably, you'd get the responses from a database or by using a complicated vision algorithm to count the kids who raised their hands. Here, we hardcode it so that we can test the `most_common` method. It returns a list that has only one element (because we requested one element in the parameter). This element stores the name of the top choice at position zero, hence the double `[0][0]` at the end of the call. I think they look like a surprised face, don't you? Your computer is probably amazed it can count data so easily. It's ancestor, Hollerith's Tabulating Machine for the 1890 US census, must be so jealous!

Lists

Lists are the least object-oriented of Python's data structures. While lists are, themselves, objects, there is a lot of syntax in Python to make using them as painless as possible. Unlike many other object-oriented languages, lists in Python are simply available. We don't need to import them and rarely need to call methods on them. We can loop over a list without explicitly requesting an iterator object, and we can construct a list (as with a dictionary) with custom syntax. Further, list comprehensions and generator expressions turn them into a veritable Swiss-army knife of computing functionality.

We won't go into too much detail of the syntax; you've seen it in introductory tutorials across the Web and in previous examples in this book. You can't code Python very long without learning how to use lists! Instead, we'll be covering when lists should be used, and their nature as objects. If you don't know how to create or append to a list, how to retrieve items from a list, or what "slice notation" is, I direct you to the official Python tutorial, post-haste. It can be found online at `http://docs.python.org/3/tutorial/`.

In Python, lists should normally be used when we want to store several instances of the "same" type of object; lists of strings or lists of numbers; most often, lists of objects we've defined ourselves. Lists should always be used when we want to store items in some kind of order. Often, this is the order in which they were inserted, but they can also be sorted by some criteria.

As we saw in the case study from the previous chapter, lists are also very useful when we need to modify the contents: insert to or delete from an arbitrary location of the list, or update a value within the list.

Like dictionaries, Python lists use an extremely efficient and well-tuned internal data structure so we can worry about what we're storing, rather than how we're storing it. Many object-oriented languages provide different data structures for queues, stacks, linked lists, and array-based lists. Python does provide special instances of some of these classes, if optimizing access to huge sets of data is required. Normally, however, the list data structure can serve all these purposes at once, and the coder has complete control over how they access it.

Don't use lists for collecting different attributes of individual items. We do not want, for example, a list of the properties a particular shape has. Tuples, named tuples, dictionaries, and objects would all be more suitable for this purpose. In some languages, they might create a list in which each alternate item is a different type; for example, they might write `['a', 1, 'b', 3]` for our letter frequency list. They'd have to use a strange loop that accesses two elements in the list at once or a modulus operator to determine which position was being accessed.

Don't do this in Python. We can group related items together using a dictionary, as we did in the previous section (if sort order doesn't matter), or using a list of tuples. Here's a rather convoluted example that demonstrates how we could do the frequency example using a list. It is much more complicated than the dictionary examples, and illustrates the effect choosing the right (or wrong) data structure can have on the readability of our code:

```
import string
CHARACTERS  = list(string.ascii_letters) + [" "]

def letter_frequency(sentence):
    frequencies = [(c, 0) for c in CHARACTERS]
    for letter in sentence:
        index = CHARACTERS.index(letter)
        frequencies[index] = (letter,frequencies[index][1]+1)
    return frequencies
```

This code starts with a list of possible characters. The `string.ascii_letters` attribute provides a string of all the letters, lowercase and uppercase, in order. We convert this to a list, and then use list concatenation (the plus operator causes two lists to be merged into one) to add one more character, the space. These are the available characters in our frequency list (the code would break if we tried to add a letter that wasn't in the list, but an exception handler could solve this).

The first line inside the function uses a list comprehension to turn the `CHARACTERS` list into a list of tuples. List comprehensions are an important, non-object-oriented tool in Python; we'll be covering them in detail in the next chapter.

Then we loop over each of the characters in the sentence. We first look up the index of the character in the `CHARACTERS` list, which we know has the same index in our frequencies list, since we just created the second list from the first. We then update that index in the frequencies list by creating a new tuple, discarding the original one. Aside from the garbage collection and memory waste concerns, this is rather difficult to read!

Like dictionaries, lists are objects too, and they have several methods that can be invoked upon them. Here are some common ones:

- The `append(element)` method adds an element to the end of the list
- The `insert(index, element)` method inserts an item at a specific position
- The `count(element)` method tells us how many times an element appears in the list
- The `index()` method tells us the index of an item in the list, raising an exception if it can't find it
- The `find()` method does the same thing, but returns `-1` instead of raising an exception for missing items
- The `reverse()` method does exactly what it says—turns the list around
- The `sort()` method has some rather intricate object-oriented behaviors, which we'll cover now

Sorting lists

Without any parameters, `sort` will generally do the expected thing. If it's a list of strings, it will place them in alphabetical order. This operation is case sensitive, so all capital letters will be sorted before lowercase letters, that is `Z` comes before `a`. If it is a list of numbers, they will be sorted in numerical order. If a list of tuples is provided, the list is sorted by the first element in each tuple. If a mixture containing unsortable items is supplied, the sort will raise a `TypeError` exception.

If we want to place objects we define ourselves into a list and make those objects sortable, we have to do a bit more work. The special method `__lt__`, which stands for "less than", should be defined on the class to make instances of that class comparable. The `sort` method on list will access this method on each object to determine where it goes in the list. This method should return `True` if our class is somehow less than the passed parameter, and `False` otherwise. Here's a rather silly class that can be sorted based on either a string or a number:

```
class WeirdSortee:
    def __init__(self, string, number, sort_num):
        self.string = string
        self.number = number
        self.sort_num = sort_num

    def __lt__(self, object):
        if self.sort_num:
            return self.number < object.number
        return self.string < object.string

    def __repr__(self):
        return"{}:{}".format(self.string, self.number)
```

The `__repr__` method makes it easy to see the two values when we print a list. The `__lt__` method's implementation compares the object to another instance of the same class (or any duck typed object that has `string`, `number`, and `sort_num` attributes; it will fail if those attributes are missing). The following output illustrates this class in action, when it comes to sorting:

```
>>> a = WeirdSortee('a', 4, True)
>>> b = WeirdSortee('b', 3, True)
>>> c = WeirdSortee('c', 2, True)
>>> d = WeirdSortee('d', 1, True)
>>> l = [a,b,c,d]
>>> l
[a:4, b:3, c:2, d:1]
>>> l.sort()
>>> l
[d:1, c:2, b:3, a:4]
```

```
>>> for i in l:
...     i.sort_num = False
...
>>> l.sort()
>>> l
[a:4, b:3, c:2, d:1]
```

The first time we call `sort`, it sorts by numbers because `sort_num` is `True` on all the objects being compared. The second time, it sorts by letters. The `__lt__` method is the only one we need to implement to enable sorting. Technically, however, if it is implemented, the class should normally also implement the similar `__gt__`, `__eq__`, `__ne__`, `__ge__`, and `__le__` methods so that all of the `<`, `>`, `==`, `!=`, `>=`, and `<=` operators also work properly. You can get this for free by implementing `__lt__` and `__eq__`, and then applying the `@total_ordering` class decorator to supply the rest:

```
from functools import total_ordering

@total_ordering
class WeirdSortee:
    def __init__(self, string, number, sort_num):
        self.string = string
        self.number = number
        self.sort_num = sort_num

    def __lt__(self, object):
        if self.sort_num:
            return self.number < object.number
        return self.string < object.string

    def __repr__(self):
        return"{}:{}".format(self.string, self.number)

    def __eq__(self, object):
        return all((
            self.string == object.string,
            self.number == object.number,
            self.sort_num == object.number
        ))
```

This is useful if we want to be able to use operators on our objects. However, if all we want to do is customize our sort orders, even this is overkill. For such a use case, the `sort` method can take an optional `key` argument. This argument is a function that can translate each object in a list into an object that can somehow be compared. For example, we can use `str.lower` as the key argument to perform a case-insensitive sort on a list of strings:

```
>>> l = ["hello", "HELP", "Helo"]
>>> l.sort()
>>> l
['HELP', 'Helo', 'hello']
>>> l.sort(key=str.lower)
>>> l
['hello', 'Helo', 'HELP']
```

Remember, even though `lower` is a method on string objects, it is also a function that can accept a single argument, `self`. In other words, `str.lower(item)` is equivalent to `item.lower()`. When we pass this function as a key, it performs the comparison on lowercase values instead of doing the default case-sensitive comparison.

There are a few sort key operations that are so common that the Python team has supplied them so you don't have to write them yourself. For example, it is often common to sort a list of tuples by something other than the first item in the list. The `operator.itemgetter` method can be used as a key to do this:

```
>>> from operator import itemgetter
>>> l = [('h', 4), ('n', 6), ('o', 5), ('p', 1), ('t', 3), ('y', 2)]
>>> l.sort(key=itemgetter(1))
>>> l
[('p', 1), ('y', 2), ('t', 3), ('h', 4), ('o', 5), ('n', 6)]
```

The `itemgetter` function is the most commonly used one (it works if the objects are dictionaries, too), but you will sometimes find use for `attrgetter` and `methodcaller`, which return attributes on an object and the results of method calls on objects for the same purpose. See the `operator` module documentation for more information.

Sets

Lists are extremely versatile tools that suit most container object applications. But they are not useful when we want to ensure objects in the list are unique. For example, a song library may contain many songs by the same artist. If we want to sort through the library and create a list of all the artists, we would have to check the list to see if we've added the artist already, before we add them again.

This is where sets come in. Sets come from mathematics, where they represent an unordered group of (usually) unique numbers. We can add a number to a set five times, but it will show up in the set only once.

In Python, sets can hold any hashable object, not just numbers. Hashable objects are the same objects that can be used as keys in dictionaries; so again, lists and dictionaries are out. Like mathematical sets, they can store only one copy of each object. So if we're trying to create a list of song artists, we can create a set of string names and simply add them to the set. This example starts with a list of (song, artist) tuples and creates a set of the artists:

```
song_library = [("Phantom Of The Opera", "Sarah Brightman"),
        ("Knocking On Heaven's Door", "Guns N' Roses"),
        ("Captain Nemo", "Sarah Brightman"),
        ("Patterns In The Ivy", "Opeth"),
        ("November Rain", "Guns N' Roses"),
        ("Beautiful", "Sarah Brightman"),
        ("Mal's Song", "Vixy and Tony")]

artists = set()
for song, artist in song_library:
    artists.add(artist)

print(artists)
```

There is no built-in syntax for an empty set as there is for lists and dictionaries; we create a set using the `set()` constructor. However, we can use the curly braces (borrowed from dictionary syntax) to create a set, so long as the set contains values. If we use colons to separate pairs of values, it's a dictionary, as in `{'key': 'value', 'key2': 'value2'}`. If we just separate values with commas, it's a set, as in `{'value', 'value2'}`. Items can be added individually to the set using its `add` method. If we run this script, we see that the set works as advertised:

```
{'Sarah Brightman', "Guns N' Roses", 'Vixy and Tony', 'Opeth'}
```

If you're paying attention to the output, you'll notice that the items are not printed in the order they were added to the sets. Sets, like dictionaries, are unordered. They both use an underlying hash-based data structure for efficiency. Because they are unordered, sets cannot have items looked up by index. The primary purpose of a set is to divide the world into two groups: "things that are in the set", and, "things that are not in the set". It is easy to check whether an item is in the set or to loop over the items in a set, but if we want to sort or order them, we'll have to convert the set to a list. This output shows all three of these activities:

```
>>> "Opeth" in artists
True
>>> for artist in artists:
...     print("{} plays good music".format(artist))
...
Sarah Brightman plays good music
Guns N' Roses plays good music
Vixy and Tony play good music
Opeth plays good music
>>> alphabetical = list(artists)
>>> alphabetical.sort()
>>> alphabetical
["Guns N' Roses", 'Opeth', 'Sarah Brightman', 'Vixy and Tony']
```

While the primary *feature* of a set is uniqueness, that is not its primary *purpose*. Sets are most useful when two or more of them are used in combination. Most of the methods on the set type operate on other sets, allowing us to efficiently combine or compare the items in two or more sets. These methods have strange names, since they use the same terminology used in mathematics. We'll start with three methods that return the same result, regardless of which is the calling set and which is the called set.

The `union` method is the most common and easiest to understand. It takes a second set as a parameter and returns a new set that contains all elements that are in *either* of the two sets; if an element is in both original sets, it will, of course, only show up once in the new set. Union is like a logical `or` operation, indeed, the `|` operator can be used on two sets to perform the union operation, if you don't like calling methods.

Conversely, the intersection method accepts a second set and returns a new set that contains only those elements that are in *both* sets. It is like a logical `and` operation, and can also be referenced using the `&` operator.

Finally, the `symmetric_difference` method tells us what's left; it is the set of objects that are in one set or the other, but not both. The following example illustrates these methods by comparing some artists from my song library to those in my sister's:

```
my_artists = {"Sarah Brightman", "Guns N' Roses",
        "Opeth", "Vixy and Tony"}

auburns_artists = {"Nickelback", "Guns N' Roses",
        "Savage Garden"}

print("All: {}".format(my_artists.union(auburns_artists)))
print("Both: {}".format(auburns_artists.intersection(my_artists)))
print("Either but not both: {}".format(
    my_artists.symmetric_difference(auburns_artists)))
```

If we run this code, we see that these three methods do what the print statements suggest they will do:

```
All: {'Sarah Brightman', "Guns N' Roses", 'Vixy and Tony',
'Savage Garden', 'Opeth', 'Nickelback'}
Both: {"Guns N' Roses"}
Either but not both: {'Savage Garden', 'Opeth', 'Nickelback',
'Sarah Brightman', 'Vixy and Tony'}
```

These methods all return the same result, regardless of which set calls the other. We can say `my_artists.union(auburns_artists)` or `auburns_artists.union(my_artists)` and get the same result. There are also methods that return different results depending on who is the caller and who is the argument.

These methods include `issubset` and `issuperset`, which are the inverse of each other. Both return a `bool`. The `issubset` method returns `True`, if all of the items in the calling set are also in the set passed as an argument. The `issuperset` method returns `True` if all of the items in the argument are also in the calling set. Thus `s.issubset(t)` and `t.issuperset(s)` are identical. They will both return `True` if `t` contains all the elements in `s`.

Finally, the `difference` method returns all the elements that are in the calling set, but not in the set passed as an argument; this is like half a `symmetric_difference`. The `difference` method can also be represented by the `-` operator. The following code illustrates these methods in action:

```
my_artists = {"Sarah Brightman", "Guns N' Roses",
        "Opeth", "Vixy and Tony"}

bands = {"Guns N' Roses", "Opeth"}

print("my_artists is to bands:")
print("issuperset: {}".format(my_artists.issuperset(bands)))
print("issubset: {}".format(my_artists.issubset(bands)))
print("difference: {}".format(my_artists.difference(bands)))
print("*"*20)
print("bands is to my_artists:")
print("issuperset: {}".format(bands.issuperset(my_artists)))
print("issubset: {}".format(bands.issubset(my_artists)))
print("difference: {}".format(bands.difference(my_artists)))
```

This code simply prints out the response of each method when called from one set on the other. Running it gives us the following output:

```
my_artists is to bands:
issuperset: True
issubset: False
difference: {'Sarah Brightman', 'Vixy and Tony'}
********************
bands is to my_artists:
issuperset: False
issubset: True
difference: set()
```

The `difference` method, in the second case, returns an empty set, since there are no items in `bands` that are not in `my_artists`.

The `union`, `intersection`, and `difference` methods can all take multiple sets as arguments; they will return, as we might expect, the set that is created when the operation is called on all the parameters.

So the methods on sets clearly suggest that sets are meant to operate on other sets, and that they are not just containers. If we have data coming in from two different sources and need to quickly combine them in some way, to determine where the data overlaps or is different, we can use set operations to efficiently compare them. Or if we have data incoming that may contain duplicates of data that has already been processed, we can use sets to compare the two and process only the new data.

Finally, it is valuable to know that sets are much more efficient than lists when checking for membership using the `in` keyword. If you use the syntax `value in container` on a set or a list, it will return `True` if one of the elements in `container` is equal to `value` and `False` otherwise. However, in a list, it will look at every object in the container until it finds the value, whereas in a set, it simply hashes the value and checks for membership. This means that a set will find the value in the same amount of time no matter how big the container is, but a list will take longer and longer to search for a value as the list contains more and more values.

Extending built-ins

We discussed briefly in *Chapter 3*, *When Objects Are Alike*, how built-in data types can be extended using inheritance. Now, we'll go into more detail as to when we would want to do that.

When we have a built-in container object that we want to add functionality to, we have two options. We can either create a new object, which holds that container as an attribute (composition), or we can subclass the built-in object and add or adapt methods on it to do what we want (inheritance).

Composition is usually the best alternative if all we want to do is use the container to store some objects using that container's features. That way, it's easy to pass that data structure into other methods and they will know how to interact with it. But we need to use inheritance if we want to change the way the container actually works. For example, if we want to ensure every item in a `list` is a string with exactly five characters, we need to extend `list` and override the `append()` method to raise an exception for invalid input. We'd also minimally have to override `__setitem__(self, index, value)`, a special method on lists that is called whenever we use the `x[index] = "value"` syntax, and the `extend()` method.

Yes, lists are objects. All that special non-object-oriented looking syntax we've been looking at for accessing lists or dictionary keys, looping over containers, and similar tasks is actually "syntactic sugar" that maps to an object-oriented paradigm underneath. We might ask the Python designers why they did this. Isn't object-oriented programming *always* better? That question is easy to answer. In the following hypothetical examples, which is easier to read, as a programmer? Which requires less typing?

```
c = a + b
c = a.add(b)

l[0] = 5
l.setitem(0, 5)
d[key] = value
d.setitem(key, value)

for x in alist:
    #do something with x
it = alist.iterator()
while it.has_next():
    x = it.next()
    #do something with x
```

The highlighted sections show what object-oriented code might look like (in practice, these methods actually exist as special double-underscore methods on associated objects). Python programmers agree that the non-object-oriented syntax is easier both to read and to write. Yet all of the preceding Python syntaxes map to object-oriented methods underneath the hood. These methods have special names (with double-underscores before and after) to remind us that there is a better syntax out there. However, it gives us the means to override these behaviors. For example, we can make a special integer that always returns `0` when we add two of them together:

```
class SillyInt(int):
    def __add__(self, num):
        return 0
```

This is an extremely bizarre thing to do, granted, but it perfectly illustrates these object-oriented principles in action:

```
>>> a = SillyInt(1)
>>> b = SillyInt(2)
>>> a + b
0
```

The awesome thing about the `__add__` method is that we can add it to any class we write, and if we use the `+` operator on instances of that class, it will be called. This is how string, tuple, and list concatenation works, for example.

This is true of all the special methods. If we want to use `x in myobj` syntax for a custom-defined object, we can implement `__contains__`. If we want to use `myobj[i] = value` syntax, we supply a `__setitem__` method and if we want to use `something = myobj[i]`, we implement `__getitem__`.

There are 33 of these special methods on the `list` class. We can use the `dir` function to see all of them:

```
>>> dir(list)


['__add__', '__class__', '__contains__', '__delattr__','__delitem__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__
getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__
new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
'__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__
subclasshook__', 'append', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort'
```

Further, if we desire additional information on how any of these methods works, we can use the `help` function:

```
>>> help(list.__add__)
Help on wrapper_descriptor:


__add__(self, value, /)
    Return self+value.
```

The plus operator on lists concatenates two lists. We don't have room to discuss all of the available special functions in this book, but you are now able to explore all this functionality with `dir` and `help`. The official online Python reference (`https://docs.python.org/3/`) has plenty of useful information as well. Focus, especially, on the abstract base classes discussed in the `collections` module.

So, to get back to the earlier point about when we would want to use composition versus inheritance: if we need to somehow change any of the methods on the class—including the special methods—we definitely need to use inheritance. If we used composition, we could write methods that do the validation or alterations and ask the caller to use those methods, but there is nothing stopping them from accessing the property directly. They could insert an item into our list that does not have five characters, and that might confuse other methods in the list.

Often, the need to extend a built-in data type is an indication that we're using the wrong sort of data type. It is not always the case, but if we are looking to extend a built-in, we should carefully consider whether or not a different data structure would be more suitable.

For example, consider what it takes to create a dictionary that remembers the order in which keys were inserted. One way to do this is to keep an ordered list of keys that is stored in a specially derived subclass of `dict`. Then we can override the methods `keys`, `values`, `__iter__`, and `items` to return everything in order. Of course, we'll also have to override `__setitem__` and `setdefault` to keep our list up to date. There are likely to be a few other methods in the output of `dir(dict)` that need overriding to keep the list and dictionary consistent (`clear` and `__delitem__` come to mind, to track when items are removed), but we won't worry about them for this example.

So we'll be extending `dict` and adding a list of ordered keys. Trivial enough, but where do we create the actual list? We could include it in the `__init__` method, which would work just fine, but we have no guarantees that any subclass will call that initializer. Remember the `__new__` method we discussed in *Chapter 2*, *Objects in Python*? I said it was generally only useful in very special cases. This is one of those special cases. We know `__new__` will be called exactly once, and we can create a list on the new instance that will always be available to our class. With that in mind, here is our entire sorted dictionary:

```
from collections import KeysView, ItemsView, ValuesView
class DictSorted(dict):
    def __new__(*args, **kwargs):
        new_dict = dict.__new__(*args, **kwargs)
        new_dict.ordered_keys = []
        return new_dict

    def __setitem__(self, key, value):
        '''self[key] = value syntax'''
        if key not in self.ordered_keys:
            self.ordered_keys.append(key)
        super().__setitem__(key, value)

    def setdefault(self, key, value):
        if key not in self.ordered_keys:
            self.ordered_keys.append(key)
        return super().setdefault(key, value)

    def keys(self):
        return KeysView(self)

    def values(self):
```

```
        return ValuesView(self)

    def items(self):
        return ItemsView(self)

    def __iter__(self):
        '''for x in self syntax'''
        return self.ordered_keys.__iter__()
```

The `__new__` method creates a new dictionary and then puts an empty list on that object. We don't override `__init__`, as the default implementation works (actually, this is only true if we initialize an empty `DictSorted` object, which is standard behavior. If we want to support other variations of the `dict` constructor, which accept dictionaries or lists of tuples, we'd need to fix `__init__` to also update our `ordered_keys` list). The two methods for setting items are very similar; they both update the list of keys, but only if the item hasn't been added before. We don't want duplicates in the list, but we can't use a set here; it's unordered!

The `keys`, `items`, and `values` methods all return views onto the dictionary. The collections library provides three read-only `View` objects onto the dictionary; they use the `__iter__` method to loop over the keys, and then use `__getitem__` (which we didn't need to override) to retrieve the values. So, we only need to define our custom `__iter__` method to make these three views work. You would think the superclass would create these views properly using polymorphism, but if we don't override these three methods, they don't return properly ordered views.

Finally, the `__iter__` method is the really special one; it ensures that if we loop over the dictionary's keys (using `for...in` syntax), it will return the values in the correct order. It does this by returning the `__iter__` of the `ordered_keys` list, which returns the same iterator object that would be used if we used `for...in` on the list instead. Since `ordered_keys` is a list of all available keys (due to the way we overrode other methods), this is the correct iterator object for the dictionary as well.

Let's look at a few of these methods in action, compared to a normal dictionary:

```
>>> ds = DictSorted()
>>> d = {}
>>> ds['a'] = 1
>>> ds['b'] = 2
>>> ds.setdefault('c', 3)
3
>>> d['a'] = 1
>>> d['b'] = 2
>>> d.setdefault('c', 3)
3
```

```
>>> for k,v in ds.items():
...     print(k,v)
...
a 1
b 2
c 3
>>> for k,v in d.items():
...     print(k,v)
...
a 1
c 3
b 2
```

Ah, our dictionary is sorted and the normal dictionary is not. Hurray!

If you wanted to use this class in production, you'd have to override several other special methods to ensure the keys are up to date in all cases. However, you don't need to do this; the functionality this class provides is already available in Python, using the `OrderedDict` object in the `collections` module. Try importing the class from `collections`, and use `help(OrderedDict)` to find out more about it.

Queues

Queues are peculiar data structures because, like sets, their functionality can be handled entirely using lists. However, while lists are extremely versatile general-purpose tools, they are occasionally not the most efficient data structure for container operations. If your program is using a small dataset (up to hundreds or even thousands of elements on today's processors), then lists will probably cover all your use cases. However, if you need to scale your data into the millions, you may need a more efficient container for your particular use case. Python therefore provides three types of queue data structures, depending on what kind of access you are looking for. All three utilize the same API, but differ in both behavior and data structure.

Before we start our queues, however, consider the trusty list data structure. Python lists are the most advantageous data structure for many use cases:

- They support efficient random access to any element in the list

- They have strict ordering of elements
- They support the append operation efficiently

They tend to be slow, however, if you are inserting elements anywhere but the end of the list (especially so if it's the beginning of the list). As we discussed in the section on sets, they are also slow for checking if an element exists in the list, and by extension, searching. Storing data in a sorted order or reordering the data can also be inefficient.

Let's look at the three types of containers provided by the Python `queue` module.

FIFO queues

FIFO stands for **First In First Out** and represents the most commonly understood definition of the word "queue". Imagine a line of people standing in line at a bank or cash register. The first person to enter the line gets served first, the second person in line gets served second, and if a new person desires service, they join the end of the line and wait their turn.

The Python `Queue` class is just like that. It is typically used as a sort of communication medium when one or more objects is producing data and one or more other objects is consuming the data in some way, probably at a different rate. Think of a messaging application that is receiving messages from the network, but can only display one message at a time to the user. The other messages can be buffered in a queue in the order they are received. FIFO queues are utilized a lot in such concurrent applications. (We'll talk more about concurrency in *Chapter 12*, *Testing Object-oriented Programs*.)

The `Queue` class is a good choice when you don't need to access any data inside the data structure except the next object to be consumed. Using a list for this would be less efficient because under the hood, inserting data at (or removing from) the beginning of a list can require shifting every other element in the list.

Queues have a very simple API. A `Queue` can have "infinite" (until the computer runs out of memory) capacity, but it is more commonly bounded to some maximum size. The primary methods are `put()` and `get()`, which add an element to the back of the line, as it were, and retrieve them from the front, in order. Both of these methods accept optional arguments to govern what happens if the operation cannot successfully complete because the queue is either empty (can't get) or full (can't put). The default behavior is to block or idly wait until the `Queue` object has data or room available to complete the operation. You can have it raise exceptions instead by passing the `block=False` parameter. Or you can have it wait a defined amount of time before raising an exception by passing a `timeout` parameter.

The class also has methods to check whether the `Queue` is `full()` or `empty()` and there are a few additional methods to deal with concurrent access that we won't discuss here. Here is a interactive session demonstrating these principles:

```
>>> from queue import Queue
>>> lineup = Queue(maxsize=3)
>>> lineup.get(block=False)
Traceback (most recent call last):
  File "<ipython-input-5-a1c8d8492c59>", line 1, in <module>
    lineup.get(block=False)
  File "/usr/lib64/python3.3/queue.py", line 164, in get
    raise Empty
queue.Empty
>>> lineup.put("one")
>>> lineup.put("two")
>>> lineup.put("three")
>>> lineup.put("four", timeout=1)
Traceback (most recent call last):
  File "<ipython-input-9-4b9db399883d>", line 1, in <module>
    lineup.put("four", timeout=1)
  File "/usr/lib64/python3.3/queue.py", line 144, in put
raise Full
queue.Full
>>> lineup.full()
True
>>> lineup.get()
'one'
>>> lineup.get()
'two'
>>> lineup.get()
'three'
>>> lineup.empty()
True
```

Underneath the hood, Python implements queues on top of the `collections.deque` data structure. Deques are advanced data structures that permits efficient access to both ends of the collection. It provides a more flexible interface than is exposed by `Queue`. I refer you to the Python documentation if you'd like to experiment more with it.

LIFO queues

LIFO (**Last In First Out**) queues are more frequently called **stacks**. Think of a stack of papers where you can only access the top-most paper. You can put another paper on top of the stack, making it the new top-most paper, or you can take the top-most paper away to reveal the one beneath it.

Traditionally, the operations on stacks are named push and pop, but the Python `queue` module uses the exact same API as for FIFO queues: `put()` and `get()`. However, in a LIFO queue, these methods operate on the "top" of the stack instead of at the front and back of a line. This is an excellent example of polymorphism. If you look at the `Queue` source code in the Python standard library, you'll actually see that there is a superclass with subclasses for FIFO and LIFO queues that implement the few operations (operating on the top of a stack instead of front and back of a `deque` instance) that are critically different between the two.

Here's an example of the LIFO queue in action:

```
>>> from queue import LifoQueue
>>> stack = LifoQueue(maxsize=3)
>>> stack.put("one")
>>> stack.put("two")
>>> stack.put("three")
>>> stack.put("four", block=False)
Traceback (most recent call last):
  File "<ipython-input-21-5473b359e5a8>", line 1, in <module>
    stack.put("four", block=False)
  File "/usr/lib64/python3.3/queue.py", line 133, in put
    raise Full
queue.Full


>>> stack.get()
'three'
>>> stack.get()
```

```
'two'
>>> stack.get()
'one'
>>> stack.empty()
True
>>> stack.get(timeout=1)
Traceback (most recent call last):
  File "<ipython-input-26-28e084a84a10>", line 1, in <module>
    stack.get(timeout=1)
  File "/usr/lib64/python3.3/queue.py", line 175, in get
    raise Empty
queue.Empty
```

You might wonder why you couldn't just use the `append()` and `pop()` methods on a standard list. Quite frankly, that's probably what I would do. I rarely have occasion to use the `LifoQueue` class in production code. Working with the end of a list is an efficient operation; so efficient, in fact, that the `LifoQueue` uses a standard list under the hood!

There are a couple of reasons that you might want to use `LifoQueue` instead of a list. The most important one is that `LifoQueue` supports clean concurrent access from multiple threads. If you need stack-like behavior in a concurrent setting, you should leave the list at home. Second, `LifoQueue` enforces the stack interface. You can't unwittingly insert a value to the wrong position in a `LifoQueue`, for example (although, as an exercise, you can work out how to do this completely wittingly).

Priority queues

The priority queue enforces a very different style of ordering from the previous queue implementations. Once again, they follow the exact same `get()` and `put()` API, but instead of relying on the order that items arrive to determine when they should be returned, the most "important" item is returned. By convention, the most important, or highest priority item is the one that sorts lowest using the less than operator.

A common convention is to store tuples in the priority queue, where the first element in the tuple is the priority for that element, and the second element is the data. Another common paradigm is to implement the `__lt__` method, as we discussed earlier in this chapter. It is perfectly acceptable to have multiple elements with the same priority in the queue, although there are no guarantees on which one will be returned first.

A priority queue might be used, for example, by a search engine to ensure it refreshes the content of the most popular web pages before crawling sites that are less likely to be searched for. A product recommendation tool might use one to display information about the most highly ranked products while still loading data for the lower ranks.

Note that a priority queue will always return the most important element currently in the queue. The `get()` method will block (by default) if the queue is empty, but it will not block and wait for a higher priority element to be added if there is already something in the queue. The queue knows nothing about elements that have not been added yet (or even about elements that have been previously extracted), and only makes decisions based on the current contents of the queue.

This interactive session shows a priority queue in action, using tuples as weights to determine what order items are processed in:

```
>>> heap.put((3, "three"))
>>> heap.put((4, "four"))
>>> heap.put((1, "one") )
>>> heap.put((2, "two"))
>>> heap.put((5, "five"), block=False)
Traceback (most recent call last):
  File "<ipython-input-23-d4209db364ed>", line 1, in <module>
    heap.put((5, "five"), block=False)
  File "/usr/lib64/python3.3/queue.py", line 133, in put
    raise Full
Full
>>> while not heap.empty():
    print(heap.get())
(1, 'one')
(2, 'two')
(3, 'three')
(4, 'four')
```

Priority queues are almost universally implemented using the `heap` data structure. Python's implementation utilizes the `heapq` module to effectively store a heap inside a normal list. I direct you to an algorithm and data-structure's textbook for more information on heaps, not to mention many other fascinating structures we haven't covered here. No matter what the data structure, you can use object-oriented principles to wrap relevant algorithms (behaviors), such as those supplied in the `heapq` module, around the data they are structuring in the computer's memory, just as the `queue` module has done on our behalf in the standard library.

Case study

To tie everything together, we'll be writing a simple link collector, which will visit a website and collect every link on every page it finds in that site. Before we start, though, we'll need some test data to work with. Simply write some HTML files to work with that contain links to each other and to other sites on the Internet, something like this:

```
<html>
    <body>
        <a href="contact.html">Contact us</a>
        <a href="blog.html">Blog</a>
        <a href="esme.html">My Dog</a>
        <a href="/hobbies.html">Some hobbies</a>
        <a href="/contact.html">Contact AGAIN</a>
        <a href="http://www.archlinux.org/">Favorite OS</a>
    </body>
</html>
```

Name one of the files `index.html` so it shows up first when pages are served. Make sure the other files exist, and keep things complicated so there is lots of linking between them. The examples for this chapter include a directory called `case_study_serve` (one of the lamest personal websites in existence!) if you would rather not set them up yourself.

Now, start a simple web server by entering the directory containing all these files and run the following command:

```
python3 -m http.server
```

This will start a server running on port 8000; you can see the pages you made by visiting `http://localhost:8000/` in your web browser.

I doubt anyone can get a website up and running with less work! Never let it be said, "you can't do that easily with Python."

The goal will be to pass our collector the base URL for the site (in this case: `http://localhost:8000/`), and have it create a list containing every unique link on the site. We'll need to take into account three types of URLs (links to external sites, which start with `http://`, absolute internal links, which start with a `/` character, and relative links, for everything else). We also need to be aware that pages may link to each other in a loop; we need to be sure we don't process the same page multiple times, or it may never end. With all this uniqueness going on, it sounds like we're going to need some sets.

Before we get into that, let's start with the basics. What code do we need to connect to a page and parse all the links from that page?

```
from urllib.request import urlopen
from urllib.parse import urlparse
import re
import sys
LINK_REGEX = re.compile(
        "<a [^>]*href=['\"]([^'\"]+)['\"][^>]*>")

class LinkCollector:
    def __init__(self, url):
        self.url = "" + urlparse(url).netloc

    def collect_links(self, path="/"):
        full_url = self.url + path
        page = str(urlopen(full_url).read())
        links = LINK_REGEX.findall(page)
        print(links)

if __name__ == "__main__":
    LinkCollector(sys.argv[1]).collect_links()
```

This is a short piece of code, considering what it's doing. It connects to the server in the argument passed on the command line, downloads the page, and extracts all the links on that page. The `__init__` method uses the `urlparse` function to extract just the hostname from the URL; so even if we pass in `http://localhost:8000/some/page.html`, it will still operate on the top level of the host `http://localhost:8000/`. This makes sense, because we want to collect all the links on the site, although it assumes every page is connected to the index by some sequence of links.

The `collect_links` method connects to and downloads the specified page from the server, and uses a regular expression to find all the links in the page. Regular expressions are an extremely powerful string processing tool. Unfortunately, they have a steep learning curve; if you haven't used them before, I strongly recommend studying any of the entire books or websites on the topic. If you don't think they're worth knowing, try writing the preceding code without them and you'll change your mind.

The example also stops in the middle of the `collect_links` method to print the value of links. This is a common way to test a program as we're writing it: stop and output the value to ensure it is the value we expect. Here's what it outputs for our example:

```
['contact.html', 'blog.html', 'esme.html', '/hobbies.html',
'/contact.html', 'http://www.archlinux.org/']
```

So now we have a collection of all the links in the first page. What can we do with it? We can't just pop the links into a set to remove duplicates because links may be relative or absolute. For example, `contact.html` and `/contact.html` point to the same page. So the first thing we should do is normalize all the links to their full URL, including hostname and relative path. We can do this by adding a `normalize_url` method to our object:

```
    def normalize_url(self, path, link):
        if link.startswith("http://"):
            return link
        elif link.startswith("/"):
            return self.url + link
        else:
            return self.url + path.rpartition(
                '/')[0] + '/' + link
```

This method converts each URL to a complete address that includes protocol and hostname. Now the two contact pages have the same value and we can store them in a set. We'll have to modify `__init__` to create the set, and `collect_links` to put all the links into it.

Then, we'll have to visit all the non-external links and collect them too. But wait a minute; if we do this, how do we keep from revisiting a link when we encounter the same page twice? It looks like we're actually going to need two sets: a set of collected links, and a set of visited links. This suggests that we were wise to choose a set to represent our data; we know that sets are most useful when we're manipulating more than one of them. Let's set these up:

```
class LinkCollector:
    def __init__(self, url):
        self.url = "http://+" + urlparse(url).netloc
        self.collected_links = set()
        self.visited_links = set()

    def collect_links(self, path="/"):
        full_url = self.url + path
        self.visited_links.add(full_url)
        page = str(urlopen(full_url).read())
        links = LINK_REGEX.findall(page)
        links = {self.normalize_url(path, link
            ) for link in links}
        self.collected_links = links.union(
                self.collected_links)
        unvisited_links = links.difference(
                self.visited_links)
```

```
            print(links, self.visited_links,
                    self.collected_links, unvisited_links)
```

The line that creates the normalized list of links uses a `set` comprehension, no different from a list comprehension, except that the result is a set of values. We'll be covering these in detail in the next chapter. Once again, the method stops to print out the current values, so we can verify that we don't have our sets confused, and that `difference` really was the method we wanted to call to collect `unvisited_links`. We can then add a few lines of code that loop over all the unvisited links and add them to the collection as well:

```
            for link in unvisited_links:
                if link.startswith(self.url):
                    self.collect_links(urlparse(link).path)
```

The `if` statement ensures that we are only collecting links from the one website; we don't want to go off and collect all the links from all the pages on the Internet (unless we're Google or the Internet Archive!). If we modify the main code at the bottom of the program to output the collected links, we can see it seems to have collected them all:

```
if __name__ == "__main__":
    collector = LinkCollector(sys.argv[1])
    collector.collect_links()
    for link in collector.collected_links:
        print(link)
```

It displays all the links we've collected, and only once, even though many of the pages in my example linked to each other multiple times:

```
$ python3 link_collector.py http://localhost:8000
http://localhost:8000/
http://en.wikipedia.org/wiki/Cavalier_King_Charles_Spaniel
http://beluminousyoga.com
http://archlinux.me/dusty/
http://localhost:8000/blog.html
http://ccphillips.net/
http://localhost:8000/contact.html
http://localhost:8000/taichi.html
http://www.archlinux.org/
http://localhost:8000/esme.html
http://localhost:8000/hobbies.html
```

Even though it collected links *to* external pages, it didn't go off collecting links *from* any of the external pages we linked to. This is a great little program if we want to collect all the links in a site. But it doesn't give me all the information I might need to build a site map; it tells me which pages I have, but it doesn't tell me which pages link to other pages. If we want to do that instead, we're going to have to make some modifications.

The first thing we should do is look at our data structures. The set of collected links doesn't work anymore; we want to know which links were linked to from which pages. The first thing we could do, then, is turn that set into a dictionary of sets for each page we visit. The dictionary keys will represent the exact same data that is currently in the set. The values will be sets of all the links on that page. Here are the changes:

```
from urllib.request import urlopen
from urllib.parse import urlparse
import re
import sys
LINK_REGEX = re.compile(
        "<a [^>]*href=['\"]([^'\"]+)['\"][^>]*>")

class LinkCollector:
    def __init__(self, url):
        self.url = "http://%s" % urlparse(url).netloc
        self.collected_links = {}
        self.visited_links = set()

    def collect_links(self, path="/"):
        full_url = self.url + path
        self.visited_links.add(full_url)
        page = str(urlopen(full_url).read())
        links = LINK_REGEX.findall(page)
        links = {self.normalize_url(path, link
            ) for link in links}
        self.collected_links[full_url] = links
        for link in links:
            self.collected_links.setdefault(link, set())
        unvisited_links = links.difference(
                self.visited_links)
        for link in unvisited_links:
            if link.startswith(self.url):
                self.collect_links(urlparse(link).path)

    def normalize_url(self, path, link):
        if link.startswith("http://"):
```

```
            return link
        elif link.startswith("/"):
            return self.url + link
        else:
            return self.url + path.rpartition('/'
                    )[0] + '/' + link
if __name__ == "__main__":
    collector = LinkCollector(sys.argv[1])
    collector.collect_links()
    for link, item in collector.collected_links.items():
        print("{}: {}".format(link, item))
```

It is a surprisingly small change; the line that originally created a union of two sets has been replaced with three lines that update the dictionary. The first of these simply tells the dictionary what the collected links for that page are. The second creates an empty set for any items in the dictionary that have not already been added to the dictionary, using `setdefault`. The result is a dictionary that contains all the links as its keys, mapped to sets of links for all the internal links, and empty sets for the external links.

Finally, instead of recursively calling `collect_links`, we can use a queue to store the links that haven't been processed yet. This implementation won't support it, but this would be a good first step to creating a multithreaded version that makes multiple requests in parallel to save time.

```
from urllib.request import urlopen
from urllib.parse import urlparse
import re
import sys
from queue import Queue
LINK_REGEX = re.compile("<a [^>]*href=['\"]([^'\"]+)['\"][^>]*>")


class LinkCollector:
    def __init__(self, url):
        self.url = "http://%s" % urlparse(url).netloc
        self.collected_links = {}
        self.visited_links = set()

    def collect_links(self):
        queue = Queue()
        queue.put(self.url)
        while not queue.empty():
            url = queue.get().rstrip('/')
            self.visited_links.add(url)
            page = str(urlopen(url).read())
```

```
                links = LINK_REGEX.findall(page)
                links = {
                    self.normalize_url(urlparse(url).path, link)
                    for link in links
                }
                self.collected_links[url] = links
                for link in links:
                    self.collected_links.setdefault(link, set())
                unvisited_links = links.difference(self.visited_links)
                for link in unvisited_links:
                    if link.startswith(self.url):
                        queue.put(link)

        def normalize_url(self, path, link):
            if link.startswith("http://"):
                return link.rstrip('/')
            elif link.startswith("/"):
                return self.url + link.rstrip('/')
            else:
                return self.url + path.rpartition('/')[0] + '/' + link.
    rstrip('/')

    if __name__ == "__main__":
        collector = LinkCollector(sys.argv[1])
        collector.collect_links()
        for link, item in collector.collected_links.items():
            print("%s: %s" % (link, item))
```

I had to manually strip any trailing forward slashes in the `normalize_url` method to remove duplicates in this version of the code.

Because the end result is an unsorted dictionary, there is no restriction on what order the links should be processed in. Therefore, we could just as easily have used a `LifoQueue` instead of a `Queue` here. A priority queue probably wouldn't make a lot of sense since there is no obvious priority to attach to a link in this case.

Exercises

The best way to learn how to choose the correct data structure is to do it wrong a few times. Take some code you've recently written, or write some new code that uses a list. Try rewriting it using some different data structures. Which ones make more sense? Which ones don't? Which have the most elegant code?

Try this with a few different pairs of data structures. You can look at examples you've done for previous chapter exercises. Are there objects with methods where you could have used `namedtuple` or `dict` instead? Attempt both and see. Are there dictionaries that could have been sets because you don't really access the values? Do you have lists that check for duplicates? Would a set suffice? Or maybe several sets? Would one of the queue implementations be more efficient? Is it useful to restrict the API to the top of a stack rather than allowing random access to the list?

If you want some specific examples to work with, try adapting the link collector to also save the title used for each link. Perhaps you can generate a site map in HTML that lists all the pages on the site, and contains a list of links to other pages, named with the same link titles.

Have you written any container objects recently that you could improve by inheriting a built-in and overriding some of the "special" double-underscore methods? You may have to do some research (using `dir` and `help`, or the Python library reference) to find out which methods need overriding. Are you sure inheritance is the correct tool to apply; could a composition-based solution be more effective? Try both (if it's possible) before you decide. Try to find different situations where each method is better than the other.

If you were familiar with the various Python data structures and their uses before you started this chapter, you may have been bored. But if that is the case, there's a good chance you use data structures too much! Look at some of your old code and rewrite it to use more self-made objects. Carefully consider the alternatives and try them all out; which one makes for the most readable and maintainable system?

Always critically evaluate your code and design decisions. Make a habit of reviewing old code and take note if your understanding of "good design" has changed since you've written it. Software design has a large aesthetic component, and like artists with oil on canvas, we all have to find the style that suits us best.

Summary

We've covered several built-in data structures and attempted to understand how to choose one for specific applications. Sometimes, the best thing we can do is create a new class of objects, but often, one of the built-ins provides exactly what we need. When it doesn't, we can always use inheritance or composition to adapt them to our use cases. We can even override special methods to completely change the behavior of built-in syntaxes.

In the next chapter, we'll discuss how to integrate the object-oriented and not-so-object-oriented aspects of Python. Along the way, we'll discover that it's more object-oriented than it looks at first sight!

7
Python Object-oriented Shortcuts

There are many aspects of Python that appear more reminiscent of structural or functional programming than object-oriented programming. Although object-oriented programming has been the most visible paradigm of the past two decades, the old models have seen a recent resurgence. As with Python's data structures, most of these tools are syntactic sugar over an underlying object-oriented implementation; we can think of them as a further abstraction layer built on top of the (already abstracted) object-oriented paradigm. In this chapter, we'll be covering a grab bag of Python features that are not strictly object-oriented:

- Built-in functions that take care of common tasks in one call
- File I/O and context managers
- An alternative to method overloading
- Functions as objects

Python built-in functions

There are numerous functions in Python that perform a task or calculate a result on certain types of objects without being methods on the underlying class. They usually abstract common calculations that apply to multiple types of classes. This is duck typing at its best; these functions accept objects that have certain attributes or methods, and are able to perform generic operations using those methods. Many, but not all, of these are special double underscore methods. We've used many of the built-in functions already, but let's quickly go through the important ones and pick up a few neat tricks along the way.

The len() function

The simplest example is the `len()` function, which counts the number of items in some kind of container object, such as a dictionary or list. You've seen it before:

```
>>> len([1,2,3,4])
4
```

Why don't these objects have a length property instead of having to call a function on them? Technically, they do. Most objects that `len()` will apply to have a method called `__len__()` that returns the same value. So `len(myobj)` seems to call `myobj.__len__()`.

Why should we use the `len()` function instead of the `__len__` method? Obviously `__len__` is a special double-underscore method, suggesting that we shouldn't call it directly. There must be an explanation for this. The Python developers don't make such design decisions lightly.

The main reason is efficiency. When we call `__len__` on an object, the object has to look the method up in its namespace, and, if the special `__getattribute__` method (which is called every time an attribute or method on an object is accessed) is defined on that object, it has to be called as well. Further, `__getattribute__` for that particular method may have been written to do something nasty, like refusing to give us access to special methods such as `__len__`! The `len()` function doesn't encounter any of this. It actually calls the `__len__` function on the underlying class, so `len(myobj)` maps to `MyObj.__len__(myobj)`.

Another reason is maintainability. In the future, the Python developers may want to change `len()` so that it can calculate the length of objects that don't have `__len__`, for example, by counting the number of items returned in an iterator. They'll only have to change one function instead of countless `__len__` methods across the board.

There is one other extremely important and often overlooked reason for `len()` being an external function: backwards compatibility. This is often cited in articles as "for historical reasons", which is a mildly dismissive phrase that an author will use to say something is the way it is because a mistake was made long ago and we're stuck with it. Strictly speaking, `len()` isn't a mistake, it's a design decision, but that decision was made in a less object-oriented time. It has stood the test of time and has some benefits, so do get used to it.

Reversed

The `reversed()` function takes any sequence as input, and returns a copy of that sequence in reverse order. It is normally used in `for` loops when we want to loop over items from back to front.

Similar to `len`, `reversed` calls the `__reversed__()` function on the class for the parameter. If that method does not exist, `reversed` builds the reversed sequence itself using calls to `__len__` and `__getitem__`, which are used to define a sequence. We only need to override `__reversed__` if we want to somehow customize or optimize the process:

```
normal_list=[1,2,3,4,5]

class CustomSequence():
    def __len__(self):
        return 5


    def __getitem__(self, index):
        return "x{0}".format(index)

class FunkyBackwards():

    def __reversed__(self):
        return "BACKWARDS!"

for seq in normal_list, CustomSequence(), FunkyBackwards():
    print("\n{}: ".format(seq.__class__.__name__), end="")
    for item in reversed(seq):
        print(item, end=", ")
```

The `for` loops at the end print the reversed versions of a normal list, and instances of the two custom sequences. The output shows that `reversed` works on all three of them, but has very different results when we define `__reversed__` ourselves:

```
list: 5, 4, 3, 2, 1,
CustomSequence: x4, x3, x2, x1, x0,
FunkyBackwards: B, A, C, K, W, A, R, D, S, !,
```

When we reverse `CustomSequence`, the `__getitem__` method is called for each item, which just inserts an `x` before the index. For `FunkyBackwards`, the `__reversed__` method returns a string, each character of which is output individually in the `for` loop.

The preceding two classes aren't very good sequences as they don't define a proper version of `__iter__`, so a forward `for` loop over them will never end.

Enumerate

Sometimes, when we're looping over a container in a `for` loop, we want access to the index (the current position in the list) of the current item being processed. The `for` loop doesn't provide us with indexes, but the `enumerate` function gives us something better: it creates a sequence of tuples, where the first object in each tuple is the index and the second is the original item.

This is useful if we need to use index numbers directly. Consider some simple code that outputs each of the lines in a file with line numbers:

```
import sys
filename = sys.argv[1]

with open(filename) as file:
    for index, line in enumerate(file):
        print("{0}: {1}".format(index+1, line), end='')
```

Running this code using it's own filename as the input file shows how it works:

```
1: import sys
2: filename = sys.argv[1]
3:
4: with open(filename) as file:
5:     for index, line in enumerate(file):
6:         print("{0}: {1}".format(index+1, line), end='')
```

The `enumerate` function returns a sequence of tuples, our `for` loop splits each tuple into two values, and the `print` statement formats them together. It adds one to the index for each line number, since `enumerate`, like all sequences, is zero-based.

We've only touched on a few of the more important Python built-in functions. As you can see, many of them call into object-oriented concepts, while others subscribe to purely functional or procedural paradigms. There are numerous others in the standard library; some of the more interesting ones include:

- `all` and `any`, which accept an iterable object and return `True` if all, or any, of the items evaluate to true (such as a nonempty string or list, a nonzero number, an object that is not `None`, or the literal `True`).
- `eval`, `exec`, and `compile`, which execute string as code inside the interpreter. Be careful with these ones; they are not safe, so don't execute code an unknown user has supplied to you (in general, assume all unknown users are malicious, foolish, or both).

- `hasattr`, `getattr`, `setattr`, and `delattr`, which allow attributes on an object to be manipulated by their string names.
- `zip`, which takes two or more sequences and returns a new sequence of tuples, where each tuple contains a single value from each sequence.
- And many more! See the interpreter help documentation for each of the functions listed in `dir(__builtins__)`.

File I/O

Our examples so far that touch the filesystem have operated entirely on text files without much thought to what is going on under the hood. Operating systems, however, actually represent files as a sequence of bytes, not text. We'll do a deep dive into the relationship between bytes and text in *Chapter 8*, *Strings and Serialization*. For now, be aware that reading textual data from a file is a fairly involved process. Python, especially Python 3, takes care of most of this work for us behind the scenes. Aren't we lucky?

The concept of files has been around since long before anyone coined the term object-oriented programming. However, Python has wrapped the interface that operating systems provide in a sweet abstraction that allows us to work with file (or file-like, vis-á-vis duck typing) objects.

The `open()` built-in function is used to open a file and return a file object. For reading text from a file, we only need to pass the name of the file into the function. The file will be opened for reading, and the bytes will be converted to text using the platform default encoding.

Of course, we don't always want to read files; often we want to write data to them! To open a file for writing, we need to pass a `mode` argument as the second positional argument, with a value of `"w"`:

```
contents = "Some file contents"
file = open("filename", "w")
file.write(contents)
file.close()
```

We could also supply the value `"a"` as a mode argument, to append to the end of the file, rather than completely overwriting existing file contents.

These files with built-in wrappers for converting bytes to text are great, but it'd be awfully inconvenient if the file we wanted to open was an image, executable, or other binary file, wouldn't it?

To open a binary file, we modify the mode string to append `'b'`. So, `'wb'` would open a file for writing bytes, while `'rb'` allows us to read them. They will behave like text files, but without the automatic encoding of text to bytes. When we read such a file, it will return `bytes` objects instead of `str`, and when we write to it, it will fail if we try to pass a text object.

These mode strings for controlling how files are opened are rather cryptic and are neither pythonic nor object-oriented. However, they are consistent with virtually every other programming language out there. File I/O is one of the fundamental jobs an operating system has to handle, and all programming languages have to talk to the OS using the same system calls. Just be glad that Python returns a file object with useful methods instead of the integer that most major operating systems use to identify a file handle!

Once a file is opened for reading, we can call the `read`, `readline`, or `readlines` methods to get the contents of the file. The `read` method returns the entire contents of the file as a `str` or `bytes` object, depending on whether there is `'b'` in the mode. Be careful not to use this method without arguments on huge files. You don't want to find out what happens if you try to load that much data into memory!

It is also possible to read a fixed number of bytes from a file; we pass an integer argument to the `read` method describing how many bytes we want to read. The next call to `read` will load the next sequence of bytes, and so on. We can do this inside a `while` loop to read the entire file in manageable chunks.

The `readline` method returns a single line from the file (where each line ends in a newline, a carriage return, or both, depending on the operating system on which the file was created). We can call it repeatedly to get additional lines. The plural `readlines` method returns a list of all the lines in the file. Like the `read` method, it's not safe to use on very large files. These two methods even work when the file is open in `bytes` mode, but it only makes sense if we are parsing text-like data that has newlines at reasonable positions. An image or audio file, for example, will not have newline characters in it (unless the newline byte happened to represent a certain pixel or sound), so applying `readline` wouldn't make sense.

For readability, and to avoid reading a large file into memory at once, it is often better to use a `for` loop directly on a file object. For text files, it will read each line, one at a time, and we can process it inside the loop body. For binary files, it's better to read fixed-sized chunks of data using the `read()` method, passing a parameter for the maximum number of bytes to read.

Writing to a file is just as easy; the `write` method on file objects writes a string (or bytes, for binary data) object to the file. It can be called repeatedly to write multiple strings, one after the other. The `writelines` method accepts a sequence of strings and writes each of the iterated values to the file. The `writelines` method does *not* append a new line after each item in the sequence. It is basically a poorly named convenience function to write the contents of a sequence of strings without having to explicitly iterate over it using a `for` loop.

Lastly, and I do mean lastly, we come to the `close` method. This method should be called when we are finished reading or writing the file, to ensure any buffered writes are written to the disk, that the file has been properly cleaned up, and that all resources associated with the file are released back to the operating system. Technically, this will happen automatically when the script exits, but it's better to be explicit and clean up after ourselves, especially in long-running processes.

Placing it in context

The need to close files when we are finished with them can make our code quite ugly. Because an exception may occur at any time during file I/O, we ought to wrap all calls to a file in a `try...finally` clause. The file should be closed in the `finally` clause, regardless of whether I/O was successful. This isn't very Pythonic. Of course, there is a more elegant way to do it.

If we run `dir` on a file-like object, we see that it has two special methods named `__enter__` and `__exit__`. These methods turn the file object into what is known as a **context manager**. Basically, if we use a special syntax called the `with` statement, these methods will be called before and after nested code is executed. On file objects, the `__exit__` method ensures the file is closed, even if an exception is raised. We no longer have to explicitly manage the closing of the file. Here is what the `with` statement looks like in practice:

```
with open('filename') as file:
    for line in file:
        print(line, end='')
```

The `open` call returns a file object, which has `__enter__` and `__exit__` methods. The returned object is assigned to the variable named `file` by the `as` clause. We know the file will be closed when the code returns to the outer indentation level, and that this will happen even if an exception is raised.

The `with` statement is used in several places in the standard library where startup or cleanup code needs to be executed. For example, the `urlopen` call returns an object that can be used in a `with` statement to clean up the socket when we're done. Locks in the threading module can automatically release the lock when the statement has been executed.

Most interestingly, because the `with` statement can apply to any object that has the appropriate special methods, we can use it in our own frameworks. For example, remember that strings are immutable, but sometimes you need to build a string from multiple parts. For efficiency, this is usually done by storing the component strings in a list and joining them at the end. Let's create a simple context manager that allows us to construct a sequence of characters and automatically convert it to a string upon exit:

```
class StringJoiner(list):
    def __enter__(self):
        return self

    def __exit__(self, type, value, tb):
        self.result = "".join(self)
```

This code adds the two special methods required of a context manager to the `list` class it inherits from. The `__enter__` method performs any required setup code (in this case, there isn't any) and then returns the object that will be assigned to the variable after `as` in the `with` statement. Often, as we've done here, this is just the context manager object itself. The `__exit__` method accepts three arguments. In a normal situation, these are all given a value of `None`. However, if an exception occurs inside the `with` block, they will be set to values related to the type, value, and traceback for the exception. This allows the `__exit__` method to do any cleanup code that may be required, even if an exception occurred. In our example, we take the irresponsible path and create a result string by joining the characters in the string, regardless of whether an exception was thrown.

While this is one of the simplest context managers we could write, and its usefulness is dubious, it does work with a `with` statement. Have a look at it in action:

```
import random, string
with StringJoiner() as joiner:
    for i in range(15):
        joiner.append(random.choice(string.ascii_letters))

print(joiner.result)
```

This code constructs a string of 15 random characters. It appends these to a `StringJoiner` using the `append` method it inherited from `list`. When the `with` statement goes out of scope (back to the outer indentation level), the `__exit__` method is called, and the `result` attribute becomes available on the joiner object. We print this value to see a random string.

An alternative to method overloading

One prominent feature of many object-oriented programming languages is a tool called **method overloading**. Method overloading simply refers to having multiple methods with the same name that accept different sets of arguments. In statically typed languages, this is useful if we want to have a method that accepts either an integer or a string, for example. In non-object-oriented languages, we might need two functions, called `add_s` and `add_i`, to accommodate such situations. In statically typed object-oriented languages, we'd need two methods, both called `add`, one that accepts strings, and one that accepts integers.

In Python, we only need one method, which accepts any type of object. It may have to do some testing on the object type (for example, if it is a string, convert it to an integer), but only one method is required.

However, method overloading is also useful when we want a method with the same name to accept different numbers or sets of arguments. For example, an e-mail message method might come in two versions, one of which accepts an argument for the "from" e-mail address. The other method might look up a default "from" e-mail address instead. Python doesn't permit multiple methods with the same name, but it does provide a different, equally flexible, interface.

We've seen some of the possible ways to send arguments to methods and functions in previous examples, but now we'll cover all the details. The simplest function accepts no arguments. We probably don't need an example, but here's one for completeness:

```
def no_args():
    pass
```

Here's how it's called:

```
no_args()
```

A function that does accept arguments will provide the names of those arguments in a comma-separated list. Only the name of each argument needs to be supplied.

When calling the function, these positional arguments must be specified in order, and none can be missed or skipped. This is the most common way we've specified arguments in our previous examples:

```
def mandatory_args(x, y, z):
    pass
```

To call it:

```
mandatory_args("a string", a_variable, 5)
```

Any type of object can be passed as an argument: an object, a container, a primitive, even functions and classes. The preceding call shows a hardcoded string, an unknown variable, and an integer passed into the function.

Default arguments

If we want to make an argument optional, rather than creating a second method with a different set of arguments, we can specify a default value in a single method, using an equals sign. If the calling code does not supply this argument, it will be assigned a default value. However, the calling code can still choose to override the default by passing in a different value. Often, a default value of `None`, or an empty string or list is suitable.

Here's a function definition with default arguments:

```
def default_arguments(x, y, z, a="Some String", b=False):
    pass
```

The first three arguments are still mandatory and must be passed by the calling code. The last two parameters have default arguments supplied.

There are several ways we can call this function. We can supply all arguments in order as though all the arguments were positional arguments:

```
default_arguments("a string", variable, 8, "", True)
```

Alternatively, we can supply just the mandatory arguments in order, leaving the keyword arguments to be assigned their default values:

```
default_arguments("a longer string", some_variable, 14)
```

We can also use the equals sign syntax when calling a function to provide values in a different order, or to skip default values that we aren't interested in. For example, we can skip the first keyword arguments and supply the second one:

```
default_arguments("a string", variable, 14, b=True)
```

Surprisingly, we can even use the equals sign syntax to mix up the order of positional arguments, so long as all of them are supplied:

```
>>> default_arguments(y=1,z=2,x=3,a="hi")
3 1 2 hi False
```

With so many options, it may seem hard to pick one, but if you think of the positional arguments as an ordered list, and keyword arguments as sort of like a dictionary, you'll find that the correct layout tends to fall into place. If you need to require the caller to specify an argument, make it mandatory; if you have a sensible default, then make it a keyword argument. Choosing how to call the method normally takes care of itself, depending on which values need to be supplied, and which can be left at their defaults.

One thing to take note of with keyword arguments is that anything we provide as a default argument is evaluated when the function is first interpreted, not when it is called. This means we can't have dynamically generated default values. For example, the following code won't behave quite as expected:

```
number = 5
def funky_function(number=number):
    print(number)

number=6
funky_function(8)
funky_function()
print(number)
```

If we run this code, it outputs the number 8 first, but then it outputs the number 5 for the call with no arguments. We had set the variable to the number 6, as evidenced by the last line of output, but when the function is called, the number 5 is printed; the default value was calculated when the function was defined, not when it was called.

This is tricky with empty containers such as lists, sets, and dictionaries. For example, it is common to ask calling code to supply a list that our function is going to manipulate, but the list is optional. We'd like to make an empty list as a default argument. We can't do this; it will create only one list, when the code is first constructed:

```
>>> def hello(b=[]):
...     b.append('a')
...     print(b)
...
```

```
>>> hello()
['a']
>>> hello()
['a', 'a']
```

Whoops, that's not quite what we expected! The usual way to get around this is to make the default value `None`, and then use the idiom `iargument = argument if argument else []` inside the method. Pay close attention!

Variable argument lists

Default values alone do not allow us all the flexible benefits of method overloading. The thing that makes Python really slick is the ability to write methods that accept an arbitrary number of positional or keyword arguments without explicitly naming them. We can also pass arbitrary lists and dictionaries into such functions.

For example, a function to accept a link or list of links and download the web pages could use such variadic arguments, or **varargs**. Instead of accepting a single value that is expected to be a list of links, we can accept an arbitrary number of arguments, where each argument is a different link. We do this by specifying the `*` operator in the function definition:

```
def get_pages(*links):
    for link in links:
        #download the link with urllib
        print(link)
```

The `*links` parameter says "I'll accept any number of arguments and put them all in a list named `links`". If we supply only one argument, it'll be a list with one element; if we supply no arguments, it'll be an empty list. Thus, all these function calls are valid:

```
get_pages()
get_pages('http://www.archlinux.org')
get_pages('http://www.archlinux.org',
        'http://ccphillips.net/')
```

We can also accept arbitrary keyword arguments. These arrive into the function as a dictionary. They are specified with two asterisks (as in `**kwargs`) in the function declaration. This tool is commonly used in configuration setups. The following class allows us to specify a set of options with default values:

```
class Options:
    default_options = {
```

```
            'port': 21,
            'host': 'localhost',
            'username': None,
            'password': None,
            'debug': False,
            }
    def __init__(self, **kwargs):
        self.options = dict(Options.default_options)
        self.options.update(kwargs)

    def __getitem__(self, key):
        return self.options[key]
```

All the interesting stuff in this class happens in the `__init__` method. We have a dictionary of default options and values at the class level. The first thing the `__init__` method does is make a copy of this dictionary. We do that instead of modifying the dictionary directly in case we instantiate two separate sets of options. (Remember, class-level variables are shared between instances of the class.) Then, `__init__` uses the `update` method on the new dictionary to change any non-default values to those supplied as keyword arguments. The `__getitem__` method simply allows us to use the new class using indexing syntax. Here's a session demonstrating the class in action:

```
>>> options = Options(username="dusty", password="drowssap",
        debug=True)
>>> options['debug']
True
>>> options['port']
21
>>> options['username']
'dusty'
```

We're able to access our options instance using dictionary indexing syntax, and the dictionary includes both default values and the ones we set using keyword arguments.

The keyword argument syntax can be dangerous, as it may break the "explicit is better than implicit" rule. In the preceding example, it's possible to pass arbitrary keyword arguments to the `Options` initializer to represent options that don't exist in the default dictionary. This may not be a bad thing, depending on the purpose of the class, but it makes it hard for someone using the class to discover what valid options are available. It also makes it easy to enter a confusing typo ("Debug" instead of "debug", for example) that adds two options where only one should have existed.

Keyword arguments are also very useful when we need to accept arbitrary arguments to pass to a second function, but we don't know what those arguments will be. We saw this in action in *Chapter 3*, *When Objects Are Alike*, when we were building support for multiple inheritance. We can, of course, combine the variable argument and variable keyword argument syntax in one function call, and we can use normal positional and default arguments as well. The following example is somewhat contrived, but demonstrates the four types in action:

```
import shutil
import os.path
def augmented_move(target_folder, *filenames,
        verbose=False, **specific):
    '''Move all filenames into the target_folder, allowing
    specific treatment of certain files.'''

    def print_verbose(message, filename):
        '''print the message only if verbose is enabled'''
        if verbose:
            print(message.format(filename))

    for filename in filenames:
        target_path = os.path.join(target_folder, filename)
        if filename in specific:
            if specific[filename] == 'ignore':
                print_verbose("Ignoring {0}", filename)
            elif specific[filename] == 'copy':
                print_verbose("Copying {0}", filename)
                shutil.copyfile(filename, target_path)
        else:
            print_verbose("Moving {0}", filename)
            shutil.move(filename, target_path)
```

This example will process an arbitrary list of files. The first argument is a target folder, and the default behavior is to move all remaining non-keyword argument files into that folder. Then there is a keyword-only argument, `verbose`, which tells us whether to print information on each file processed. Finally, we can supply a dictionary containing actions to perform on specific filenames; the default behavior is to move the file, but if a valid string action has been specified in the keyword arguments, it can be ignored or copied instead. Notice the ordering of the parameters in the function; first the positional argument is specified, then the `*filenames` list, then any specific keyword-only arguments, and finally, a `**specific` dictionary to hold remaining keyword arguments.

We create an inner helper function, `print_verbose`, which will print messages only if the `verbose` key has been set. This function keeps code readable by encapsulating this functionality into a single location.

In common cases, assuming the files in question exist, this function could be called as:

```
>>> augmented_move("move_here", "one", "two")
```

This command would move the files `one` and `two` into the `move_here` directory, assuming they exist (there's no error checking or exception handling in the function, so it would fail spectacularly if the files or target directory didn't exist). The move would occur without any output, since `verbose` is `False` by default.

If we want to see the output, we can call it with:

```
>>> augmented_move("move_here", "three", verbose=True)
Moving three
```

This moves one file named `three`, and tells us what it's doing. Notice that it is impossible to specify `verbose` as a positional argument in this example; we must pass a keyword argument. Otherwise, Python would think it was another filename in the `*filenames` list.

If we want to copy or ignore some of the files in the list, instead of moving them, we can pass additional keyword arguments:

```
>>> augmented_move("move_here", "four", "five", "six",
        four="copy", five="ignore")
```

This will move the sixth file and copy the fourth, but won't display any output, since we didn't specify `verbose`. Of course, we can do that too, and keyword arguments can be supplied in any order:

```
>>> augmented_move("move_here", "seven", "eight", "nine",
        seven="copy", verbose=True, eight="ignore")
Copying seven
Ignoring eight
Moving nine
```

Unpacking arguments

There's one more nifty trick involving variable arguments and keyword arguments. We've used it in some of our previous examples, but it's never too late for an explanation. Given a list or dictionary of values, we can pass those values into a function as if they were normal positional or keyword arguments. Have a look at this code:

```
def show_args(arg1, arg2, arg3="THREE"):
    print(arg1, arg2, arg3)

some_args = range(3)
more_args = {
        "arg1": "ONE",
        "arg2": "TWO"}

print("Unpacking a sequence:", end=" ")

show_args(*some_args)
print("Unpacking a dict:", end=" ")

show_args(**more_args)
```

Here's what it looks like when we run it:

```
Unpacking a sequence: 0 1 2
Unpacking a dict: ONE TWO THREE
```

The function accepts three arguments, one of which has a default value. But when we have a list of three arguments, we can use the `*` operator inside a function call to unpack it into the three arguments. If we have a dictionary of arguments, we can use the `**` syntax to unpack it as a collection of keyword arguments.

This is most often useful when mapping information that has been collected from user input or from an outside source (for example, an Internet page or a text file) to a function or method call.

Remember our earlier example that used headers and lines in a text file to create a list of dictionaries with contact information? Instead of just adding the dictionaries to a list, we could use keyword unpacking to pass the arguments to the `__init__` method on a specially built `Contact` object that accepts the same set of arguments. See if you can adapt the example to make this work.

Functions are objects too

Programming languages that overemphasize object-oriented principles tend to frown on functions that are not methods. In such languages, you're expected to create an object to sort of wrap the single method involved. There are numerous situations where we'd like to pass around a small object that is simply called to perform an action. This is most frequently done in event-driven programming, such as graphical toolkits or asynchronous servers; we'll see some design patterns that use it in *Chapter 10*, *Python Design Patterns I* and *Chapter 11*, *Python Design Patterns II*.

In Python, we don't need to wrap such methods in an object, because functions already are objects! We can set attributes on functions (though this isn't a common activity), and we can pass them around to be called at a later date. They even have a few special properties that can be accessed directly. Here's yet another contrived example:

```
def my_function():
    print("The Function Was Called")
my_function.description = "A silly function"

def second_function():
    print("The second was called")
second_function.description = "A sillier function."

def another_function(function):
    print("The description:", end=" ")
    print(function.description)
    print("The name:", end=" ")
    print(function.__name__)
    print("The class:", end=" ")
    print(function.__class__)
    print("Now I'll call the function passed in")
    function()

another_function(my_function)
another_function(second_function)
```

If we run this code, we can see that we were able to pass two different functions into our third function, and get different output for each one:

```
The description: A silly function
The name: my_function
The class: <class 'function'>
Now I'll call the function passed in
```

```
The Function Was Called
The description: A sillier function.
The name: second_function
The class: <class 'function'>
Now I'll call the function passed in
The second was called
```

We set an attribute on the function, named `description` (not very good descriptions, admittedly). We were also able to see the function's `__name__` attribute, and to access its class, demonstrating that the function really is an object with attributes. Then we called the function by using the callable syntax (the parentheses).

The fact that functions are top-level objects is most often used to pass them around to be executed at a later date, for example, when a certain condition has been satisfied. Let's build an event-driven timer that does just this:

```
import datetime
import time

class TimedEvent:
    def __init__(self, endtime, callback):
        self.endtime = endtime
        self.callback = callback

    def ready(self):
        return self.endtime <= datetime.datetime.now()

class Timer:
    def __init__(self):
        self.events = []

    def call_after(self, delay, callback):
        end_time = datetime.datetime.now() + \
                datetime.timedelta(seconds=delay)

        self.events.append(TimedEvent(end_time, callback))

    def run(self):
        while True:
            ready_events = (e for e in self.events if e.ready())
            for event in ready_events:
                event.callback(self)
                self.events.remove(event)
            time.sleep(0.5)
```

In production, this code should definitely have extra documentation using docstrings! The `call_after` method should at least mention that the `delay` parameter is in seconds, and that the `callback` function should accept one argument: the timer doing the calling.

We have two classes here. The `TimedEvent` class is not really meant to be accessed by other classes; all it does is store `endtime` and `callback`. We could even use a `tuple` or `namedtuple` here, but as it is convenient to give the object a behavior that tells us whether or not the event is ready to run, we use a class instead.

The `Timer` class simply stores a list of upcoming events. It has a `call_after` method to add a new event. This method accepts a `delay` parameter representing the number of seconds to wait before executing the callback, and the `callback` function itself: a function to be executed at the correct time. This `callback` function should accept one argument.

The `run` method is very simple; it uses a generator expression to filter out any events whose time has come, and executes them in order. The timer loop then continues indefinitely, so it has to be interrupted with a keyboard interrupt (*Ctrl + C* or *Ctrl + Break*). We sleep for half a second after each iteration so as to not grind the system to a halt.

The important things to note here are the lines that touch callback functions. The function is passed around like any other object and the timer never knows or cares what the original name of the function is or where it was defined. When it's time to call the function, the timer simply applies the parenthesis syntax to the stored variable.

Here's a set of callbacks that test the timer:

```
from timer import Timer
import datetime

def format_time(message, *args):
    now = datetime.datetime.now().strftime("%I:%M:%S")
    print(message.format(*args, now=now))

def one(timer):
    format_time("{now}: Called One")

def two(timer):
    format_time("{now}: Called Two")

def three(timer):
```

```
        format_time("{now}: Called Three")

    class Repeater:
        def __init__(self):
            self.count = 0
        def repeater(self, timer):
            format_time("{now}: repeat {0}", self.count)
            self.count += 1
            timer.call_after(5, self.repeater)

    timer = Timer()
    timer.call_after(1, one)
    timer.call_after(2, one)
    timer.call_after(2, two)
    timer.call_after(4, two)
    timer.call_after(3, three)
    timer.call_after(6, three)
    repeater = Repeater()
    timer.call_after(5, repeater.repeater)
    format_time("{now}: Starting")
    timer.run()
```

This example allows us to see how multiple callbacks interact with the timer. The first function is the `format_time` function. It uses the string `format` method to add the current time to the message, and illustrates variable arguments in action. The `format_time` method will accept any number of positional arguments, using variable argument syntax, which are then forwarded as positional arguments to the string's `format` method. After this, we create three simple callback methods that simply output the current time and a short message telling us which callback has been fired.

The `Repeater` class demonstrates that methods can be used as callbacks too, since they are really just functions. It also shows why the `timer` argument to the callback functions is useful: we can add a new timed event to the timer from inside a presently running callback. We then create a timer and add several events to it that are called after different amounts of time. Finally, we start the timer running; the output shows that events are run in the expected order:

```
02:53:35: Starting
02:53:36: Called One
02:53:37: Called One
02:53:37: Called Two
02:53:38: Called Three
02:53:39: Called Two
02:53:40: repeat 0
```

```
02:53:41: Called Three
02:53:45: repeat 1
02:53:50: repeat 2
02:53:55: repeat 3
02:54:00: repeat 4
```

Python 3.4 introduces a generic event-loop architecture similar to this. We'll be discussing it later in *Chapter 13, Concurrency*.

Using functions as attributes

One of the interesting effects of functions being objects is that they can be set as callable attributes on other objects. It is possible to add or change a function to an instantiated object:

```
class A:
    def print(self):
        print("my class is A")

def fake_print():
    print("my class is not A")

a = A()
a.print()
a.print = fake_print
a.print()
```

This code creates a very simple class with a `print` method that doesn't tell us anything we didn't know. Then we create a new function that tells us something we don't believe.

When we call `print` on an instance of the `A` class, it behaves as expected. If we then set the `print` method to point at a new function, it tells us something different:

```
my class is A
my class is not A
```

It is also possible to replace methods on classes instead of objects, although in that case we have to add the `self` argument to the parameter list. This will change the method for all instances of that object, even ones that have already been instantiated. Obviously, replacing methods like this can be both dangerous and confusing to maintain. Somebody reading the code will see that a method has been called and look up that method on the original class. But the method on the original class is not the one that was called. Figuring out what really happened can become a tricky, frustrating debugging session.

It does have its uses though. Often, replacing or adding methods at run time (called **monkey-patching**) is used in automated testing. If testing a client-server application, we may not want to actually connect to the server while testing the client; this may result in accidental transfers of funds or embarrassing test e-mails being sent to real people. Instead, we can set up our test code to replace some of the key methods on the object that sends requests to the server, so it only records that the methods have been called.

Monkey-patching can also be used to fix bugs or add features in third-party code that we are interacting with, and does not behave quite the way we need it to. It should, however, be applied sparingly; it's almost always a "messy hack". Sometimes, though, it is the only way to adapt an existing library to suit our needs.

Callable objects

Just as functions are objects that can have attributes set on them, it is possible to create an object that can be called as though it were a function.

Any object can be made callable by simply giving it a `__call__` method that accepts the required arguments. Let's make our `Repeater` class, from the timer example, a little easier to use by making it a callable:

```
class Repeater:
    def __init__(self):
        self.count = 0


    def __call__(self, timer):
        format_time("{now}: repeat {0}", self.count)
        self.count += 1

        timer.call_after(5, self)

timer = Timer()

timer.call_after(5, Repeater())
format_time("{now}: Starting")
timer.run()
```

This example isn't much different from the earlier class; all we did was change the name of the `repeater` function to `__call__` and pass the object itself as a callable. Note that when we make the `call_after` call, we pass the argument `Repeater()`. Those two parentheses are creating a new instance of the class; they are not explicitly calling the class. This happens later, inside the timer. If we want to execute the `__call__` method on a newly instantiated object, we'd use a rather odd syntax: `Repeater()()`. The first set of parentheses constructs the object; the second set executes the `__call__` method. If we find ourselves doing this, we may not be using the correct abstraction. Only implement the `__call__` function on an object if the object is meant to be treated like a function.

Case study

To tie together some of the principles presented in this chapter, let's build a mailing list manager. The manager will keep track of e-mail addresses categorized into named groups. When it's time to send a message, we can pick a group and send the message to all e-mail addresses assigned to that group.

Now, before we start working on this project, we ought to have a safe way to test it, without sending e-mails to a bunch of real people. Luckily, Python has our back here; like the test HTTP server, it has a built-in **Simple Mail Transfer Protocol** (**SMTP**) server that we can instruct to capture any messages we send without actually sending them. We can run the server with the following command:

```
python -m smtpd -n -c DebuggingServer localhost:1025
```

Running this command at a command prompt will start an SMTP server running on port 1025 on the local machine. But we've instructed it to use the `DebuggingServer` class (it comes with the built-in SMTP module), which, instead of sending mails to the intended recipients, simply prints them on the terminal screen as it receives them. Neat, eh?

Now, before writing our mailing list, let's write some code that actually sends mail. Of course, Python supports this in the standard library, too, but it's a bit of an odd interface, so we'll write a new function to wrap it all cleanly:

```
import smtplib
from email.mime.text import MIMEText

def send_email(subject, message, from_addr, *to_addrs,
```

```
            host="localhost", port=1025, **headers):

        email = MIMEText(message)
        email['Subject'] = subject
        email['From'] = from_addr
        for header, value in headers.items():
            email[header] = value

        sender = smtplib.SMTP(host, port)
        for addr in to_addrs:
            del email['To']
            email['To'] = addr
            sender.sendmail(from_addr, addr, email.as_string())
        sender.quit()
```

We won't cover the code inside this method too thoroughly; the documentation in the standard library can give you all the information you need to use the `smtplib` and `email` modules effectively.

We've used both variable argument and keyword argument syntax in the function call. The variable argument list allows us to supply a single string in the default case of having a single `to` address, as well as permitting multiple addresses to be supplied if required. Any extra keyword arguments are mapped to e-mail headers. This is an exciting use of variable arguments and keyword arguments, but it's not really a great interface for the person calling the function. In fact, it makes many things the programmer will want to do impossible.

The headers passed into the function represent auxiliary headers that can be attached to a method. Such headers might include `Reply-To`, `Return-Path`, or *X-pretty-much-anything*. But in order to be a valid identifier in Python, a name cannot include the `-` character. In general, that character represents subtraction. So, it's not possible to call a function with `Reply-To = my@email.com`. It appears we were too eager to use keyword arguments because they are a new tool we just learned about in this chapter.

We'll have to change the argument to a normal dictionary; this will work because any string can be used as a key in a dictionary. By default, we'd want this dictionary to be empty, but we can't make the default parameter an empty dictionary. So, we'll have to make the default argument `None`, and then set up the dictionary at the beginning of the method:

```
def send_email(subject, message, from_addr, *to_addrs,
```

```
            host="localhost", port=1025, headers=None):

        headers = {} if headers is None else headers
```

If we have our debugging SMTP server running in one terminal, we can test this code in a Python interpreter:

```
>>> send_email("A model subject", "The message contents",
 "from@example.com", "to1@example.com", "to2@example.com")
```

Then, if we check the output from the debugging SMTP server, we get the following:

```
---------- MESSAGE FOLLOWS ----------
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: A model subject
From: from@example.com
To: to1@example.com
X-Peer: 127.0.0.1

The message contents
------------ END MESSAGE ------------
---------- MESSAGE FOLLOWS ----------
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: A model subject
From: from@example.com
To: to2@example.com
X-Peer: 127.0.0.1

The message contents
------------ END MESSAGE ------------
```

Excellent, it has "sent" our e-mail to the two expected addresses with subject and message contents included. Now that we can send messages, let's work on the e-mail group management system. We'll need an object that somehow matches e-mail addresses with the groups they are in. Since this is a many-to-many relationship (any one e-mail address can be in multiple groups; any one group can be associated with multiple e-mail addresses), none of the data structures we've studied seems quite ideal. We could try a dictionary of group-names matched to a list of associated e-mail addresses, but that would duplicate e-mail addresses. We could also try a dictionary of e-mail addresses matched to groups, resulting in a duplication of groups. Neither seems optimal. Let's try this latter version, even though intuition tells me the groups to e-mail address solution would be more straightforward.

Since the values in our dictionary will always be collections of unique e-mail addresses, we should probably store them in a `set` container. We can use `defaultdict` to ensure that there is always a `set` container available for each key:

```
from collections import defaultdict
class MailingList:
    '''Manage groups of e-mail addresses for sending e-mails.'''
    def __init__(self):
        self.email_map = defaultdict(set)

    def add_to_group(self, email, group):
        self.email_map[email].add(group)
```

Now, let's add a method that allows us to collect all the e-mail addresses in one or more groups. This can be done by converting the list of groups to a set:

```
    def emails_in_groups(self, *groups):
        groups = set(groups)
        emails = set()
        for e, g in self.email_map.items():
            if g & groups:
                emails.add(e)
        return emails
```

First, look at what we're iterating over: `self.email_map.items()`. This method, of course, returns a tuple of key-value pairs for each item in the dictionary. The values are sets of strings representing the groups. We split these into two variables named `e` and `g`, short for e-mail and groups. We add the e-mail address to the set of return values only if the passed in groups intersect with the e-mail address groups. The `g & groups` syntax is a shortcut for `g.intersection(groups)`; the `set` class does this by implementing the special `__and__` method to call `intersection`.

This code could be made a wee bit more concise using a set comprehension, which we'll discuss in *Chapter 9, The Iterator Pattern*.

Now, with these building blocks, we can trivially add a method to our `MailingList` class that sends messages to specific groups:

```
def send_mailing(self, subject, message, from_addr,
        *groups, headers=None):
    emails = self.emails_in_groups(*groups)
    send_email(subject, message, from_addr,
            *emails, headers=headers)
```

This function relies on variable argument lists. As input, it takes a list of groups as variable arguments. It gets the list of e-mails for the specified groups and passes those as variable arguments into `send_email`, along with other arguments that were passed into this method.

The program can be tested by ensuring the SMTP debugging server is running in one command prompt, and, in a second prompt, loading the code using:

```
python -i mailing_list.py
```

Create a `MailingList` object with:

```
>>> m = MailingList()
```

Then create a few fake e-mail addresses and groups, along the lines of:

```
>>> m.add_to_group("friend1@example.com", "friends")
>>> m.add_to_group("friend2@example.com", "friends")
>>> m.add_to_group("family1@example.com", "family")
>>> m.add_to_group("pro1@example.com", "professional")
```

Finally, use a command like this to send e-mails to specific groups:

```
>>> m.send_mailing("A Party",
"Friends and family only: a party", "me@example.com", "friends",
"family", headers={"Reply-To": "me2@example.com"})
```

E-mails to each of the addresses in the specified groups should show up in the console on the SMTP server.

The mailing list works fine as it is, but it's kind of useless; as soon as we exit the program, our database of information is lost. Let's modify it to add a couple of methods to load and save the list of e-mail groups from and to a file.

In general, when storing structured data on disk, it is a good idea to put a lot of thought into how it is stored. One of the reasons myriad database systems exist is that if someone else has put this thought into how data is stored, you don't have to. We'll be looking at some data serialization mechanisms in the next chapter, but for this example, let's keep it simple and go with the first solution that could possibly work.

The data format I have in mind is to store each e-mail address followed by a space, followed by a comma-separated list of groups. This format seems reasonable, and we're going to go with it because data formatting isn't the topic of this chapter. However, to illustrate just why you need to think hard about how you format data on disk, let's highlight a few problems with the format.

First, the space character is technically legal in e-mail addresses. Most e-mail providers prohibit it (with good reason), but the specification defining e-mail addresses says an e-mail can contain a space if it is in quotation marks. If we are to use a space as a sentinel in our data format, we should technically be able to differentiate between that space and a space that is part of an e-mail. We're going to pretend this isn't true, for simplicity's sake, but real-life data encoding is full of stupid issues like this. Second, consider the comma-separated list of groups. What happens if someone decides to put a comma in a group name? If we decide to make commas illegal in group names, we should add validation to ensure this to our `add_to_group` method. For pedagogical clarity, we'll ignore this problem too. Finally, there are many security implications we need to consider: can someone get themselves into the wrong group by putting a fake comma in their e-mail address? What does the parser do if it encounters an invalid file?

The takeaway from this discussion is to try to use a data-storage method that has been field tested, rather than designing your own data serialization protocol. There are a ton of bizarre edge cases you might overlook, and it's better to use code that has already encountered and fixed those edge cases.

But forget that, let's just write some basic code that uses an unhealthy dose of wishful thinking to pretend this simple data format is safe:

```
email1@mydomain.com group1,group2
email2@mydomain.com group2,group3
```

The code to do this is as follows:

```
        def save(self):
            with open(self.data_file, 'w') as file:
                for email, groups in self.email_map.items():
                    file.write(
```

```
                    '{} {}\n'.format(email, ','.join(groups))
                )

    def load(self):
        self.email_map = defaultdict(set)
        try:
            with open(self.data_file) as file:
                for line in file:
                    email, groups = line.strip().split(' ')
                    groups = set(groups.split(','))
                    self.email_map[email] = groups
        except IOError:
            pass
```

In the `save` method, we open the file in a context manager and write the file as a formatted string. Remember the newline character; Python doesn't add that for us. The `load` method first resets the dictionary (in case it contains data from a previous call to `load`) uses the `for...in` syntax, which loops over each line in the file. Again, the newline character is included in the line variable, so we have to call `.strip()` to take it off. We'll learn more about such string manipulation in the next chapter.

Before using these methods, we need to make sure the object has a `self.data_file` attribute, which can be done by modifying `__init__`:

```
    def __init__(self, data_file):
        self.data_file = data_file
        self.email_map = defaultdict(set)
```

We can test these two methods in the interpreter as follows:

```
>>> m = MailingList('addresses.db')
>>> m.add_to_group('friend1@example.com', 'friends')
>>> m.add_to_group('family1@example.com', 'friends')
>>> m.add_to_group('family1@example.com', 'family')
>>> m.save()
```

The resulting `addresses.db` file contains the following lines, as expected:

```
friend1@example.com friends
family1@example.com friends,family
```

We can also load this data back into a `MailingList` object successfully:

```
>>> m = MailingList('addresses.db')
>>> m.email_map
defaultdict(<class 'set'>, {})
>>> m.load()
>>> m.email_map
defaultdict(<class 'set'>, {'friend2@example.com': {'friends\n'},
'family1@example.com': {'family\n'}, 'friend1@example.com':
{'friends\n'}})
```

As you can see, I forgot to do the `load` command, and it might be easy to forget the `save` command as well. To make this a little easier for anyone who wants to use our `MailingList` API in their own code, let's provide the methods to support a context manager:

```
    def __enter__(self):
        self.load()
        return self

    def __exit__(self, type, value, tb):
        self.save()
```

These simple methods just delegate their work to load and save, but we can now write code like this in the interactive interpreter and know that all the previously stored addresses were loaded on our behalf, and that the whole list will be saved to the file when we are done:

```
>>> with MailingList('addresses.db') as ml:
...     ml.add_to_group('friend2@example.com', 'friends')
...     ml.send_mailing("What's up", "hey friends, how's it going", 'me@
example.com', 'friends')
```

Exercises

If you haven't encountered the `with` statements and context managers before, I encourage you, as usual, to go through your old code and find all the places you were opening files, and make sure they are safely closed using the `with` statement. Look for places that you could write your own context managers as well. Ugly or repetitive `try...finally` clauses are a good place to start, but you may find them useful any time you need to do before and/or after tasks in context.

You've probably used many of the basic built-in functions before now. We covered several of them, but didn't go into a great deal of detail. Play with `enumerate`, `zip`, `reversed`, `any` and `all`, until you know you'll remember to use them when they are the right tool for the job. The `enumerate` function is especially important; because not using it results in some pretty ugly code.

Also explore some applications that pass functions around as callable objects, as well as using the `__call__` method to make your own objects callable. You can get the same effect by attaching attributes to functions or by creating a `__call__` method on an object. In which case would you use one syntax, and when would it be more suitable to use the other?

Our mailing list object could overwhelm an e-mail server if there is a massive number of e-mails to be sent out. Try refactoring it so that you can use different `send_email` functions for different purposes. One such function could be the version we used here. A different version might put the e-mails in a queue to be sent by a server in a different thread or process. A third version could just output the data to the terminal, obviating the need for a dummy SMTP server. Can you construct the mailing list with a callback such that the `send_mailing` function uses whatever is passed in? It would default to the current version if no callback is supplied.

The relationship between arguments, keyword arguments, variable arguments, and variable keyword arguments can be a bit confusing. We saw how painfully they can interact when we covered multiple inheritance. Devise some other examples to see how they can work well together, as well as to understand when they don't.

Summary

We covered a grab bag of topics in this chapter. Each represented an important non-object-oriented feature that is popular in Python. Just because we can use object-oriented principles does not always mean we should!

However, we also saw that Python typically implements such features by providing a syntax shortcut to traditional object-oriented syntax. Knowing the object-oriented principles underlying these tools allows us to use them more effectively in our own classes.

We discussed a series of built-in functions and file I/O operations. There are a whole bunch of different syntaxes available to us when calling functions with arguments, keyword arguments, and variable argument lists. Context managers are useful for the common pattern of sandwiching a piece of code between two method calls. Even functions are objects, and, conversely, any normal object can be made callable.

In the next chapter, we'll learn more about string and file manipulation, and even spend some time with one of the least object-oriented topics in the standard library: regular expressions.

8
Strings and Serialization

Before we get involved with higher level design patterns, let's take a deep dive into one of Python's most common objects: the string. We'll see that there is a lot more to the string than meets the eye, and also cover searching strings for patterns and serializing data for storage or transmission.

In particular, we'll visit:

- The complexities of strings, bytes, and byte arrays
- The ins and outs of string formatting
- A few ways to serialize data
- The mysterious regular expression

Strings

Strings are a basic primitive in Python; we've used them in nearly every example we've discussed so far. All they do is represent an immutable sequence of characters. However, though you may not have considered it before, "character" is a bit of an ambiguous word; can Python strings represent sequences of accented characters? Chinese characters? What about Greek, Cyrillic, or Farsi?

In Python 3, the answer is yes. Python strings are all represented in Unicode, a character definition standard that can represent virtually any character in any language on the planet (and some made-up languages and random characters as well). This is done seamlessly, for the most part. So, let's think of Python 3 strings as an immutable sequence of Unicode characters. So what can we do with this immutable sequence? We've touched on many of the ways strings can be manipulated in previous examples, but let's quickly cover it all in one place: a crash course in string theory!

String manipulation

As you know, strings can be created in Python by wrapping a sequence of characters in single or double quotes. Multiline strings can easily be created using three quote characters, and multiple hardcoded strings can be concatenated together by placing them side by side. Here are some examples:

```
a = "hello"
b = 'world'
c = '''a multiple
line string'''
d = """More
multiple"""
e = ("Three " "Strings "
        "Together")
```

That last string is automatically composed into a single string by the interpreter. It is also possible to concatenate strings using the `+` operator (as in `"hello " + "world"`). Of course, strings don't have to be hardcoded. They can also come from various outside sources such as text files, user input, or encoded on the network.

The automatic concatenation of adjacent strings can make for some hilarious bugs when a comma is missed. It is, however, extremely useful when a long string needs to be placed inside a function call without exceeding the 79 character line-length limit suggested by the Python style guide.

Like other sequences, strings can be iterated over (character by character), indexed, sliced, or concatenated. The syntax is the same as for lists.

The `str` class has numerous methods on it to make manipulating strings easier. The `dir` and `help` commands in the Python interpreter can tell us how to use all of them; we'll consider some of the more common ones directly.

Several Boolean convenience methods help us identify whether or not the characters in a string match a certain pattern. Here is a summary of these methods. Most of these, such as `isalpha`, `isupper`/`islower`, and `startswith`/`endswith` have obvious interpretations. The `isspace` method is also fairly obvious, but remember that all whitespace characters (including tab, newline) are considered, not just the space character.

The `istitle` method returns `True` if the first character of each word is capitalized and all other characters are lowercase. Note that it does not strictly enforce the English grammatical definition of title formatting. For example, Leigh Hunt's poem "The Glove and the Lions" should be a valid title, even though not all words are capitalized. Robert Service's "The Cremation of Sam McGee" should also be a valid title, even though there is an uppercase letter in the middle of the last word.

Be careful with the `isdigit`, `isdecimal`, and `isnumeric` methods, as they are more nuanced than you would expect. Many Unicode characters are considered numbers besides the ten digits we are used to. Worse, the period character that we use to construct floats from strings is not considered a decimal character, so `'45.2'.isdecimal()` returns `False`. The real decimal character is represented by Unicode value 0660, as in 45.2, (or `45\u06602`). Further, these methods do not verify whether the strings are valid numbers; "127.0.0.1" returns `True` for all three methods. We might think we should use that decimal character instead of a period for all numeric quantities, but passing that character into the `float()` or `int()` constructor converts that decimal character to a zero:

```
>>> float('45\u06602')
4502.0
```

Other methods useful for pattern matching do not return Booleans. The `count` method tells us how many times a given substring shows up in the string, while `find`, `index`, `rfind`, and `rindex` tell us the position of a given substring within the original string. The two 'r' (for 'right' or 'reverse') methods start searching from the end of the string. The `find` methods return `-1` if the substring can't be found, while `index` raises a `ValueError` in this situation. Have a look at some of these methods in action:

```
>>> s = "hello world"
>>> s.count('l')
3
>>> s.find('l')
2
>>> s.rindex('m')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

Most of the remaining string methods return transformations of the string. The `upper`, `lower`, `capitalize`, and `title` methods create new strings with all alphabetic characters in the given format. The `translate` method can use a dictionary to map arbitrary input characters to specified output characters.

For all of these methods, note that the input string remains unmodified; a brand new `str` instance is returned instead. If we need to manipulate the resultant string, we should assign it to a new variable, as in `new_value = value.capitalize()`. Often, once we've performed the transformation, we don't need the old value anymore, so a common idiom is to assign it to the same variable, as in `value = value.title()`.

Finally, a couple of string methods return or operate on lists. The `split` method accepts a substring and splits the string into a list of strings wherever that substring occurs. You can pass a number as a second parameter to limit the number of resultant strings. The `rsplit` behaves identically to `split` if you don't limit the number of strings, but if you do supply a limit, it starts splitting from the end of the string. The `partition` and `rpartition` methods split the string at only the first or last occurrence of the substring, and return a tuple of three values: characters before the substring, the substring itself, and the characters after the substring.

As the inverse of `split`, the `join` method accepts a list of strings, and returns all of those strings combined together by placing the original string between them. The `replace` method accepts two arguments, and returns a string where each instance of the first argument has been replaced with the second. Here are some of these methods in action:

```
>>> s = "hello world, how are you"
>>> s2 = s.split(' ')
>>> s2
['hello', 'world,', 'how', 'are', 'you']
>>> '#'.join(s2)
'hello#world,#how#are#you'
>>> s.replace(' ', '**')
'hello**world,**how**are**you'
>>> s.partition(' ')
('hello', ' ', 'world, how are you')
```

There you have it, a whirlwind tour of the most common methods on the `str` class! Now, let's look at Python 3's method for composing strings and variables to create new strings.

String formatting

Python 3 has a powerful string formatting and templating mechanism that allows us to construct strings comprised of hardcoded text and interspersed variables. We've used it in many previous examples, but it is much more versatile than the simple formatting specifiers we've used.

Any string can be turned into a format string by calling the `format()` method on it. This method returns a new string where specific characters in the input string have been replaced with values provided as arguments and keyword arguments passed into the function. The `format` method does not require a fixed set of arguments; internally, it uses the `*args` and `**kwargs` syntax that we discussed in *Chapter 7*, *Python Object-oriented Shortcuts*.

The special characters that are replaced in formatted strings are the opening and closing brace characters: { and }. We can insert pairs of these in a string and they will be replaced, in order, by any positional arguments passed to the `str.format` method:

```
template = "Hello {}, you are currently {}."
print(template.format('Dusty', 'writing'))
```

If we run these statements, it replaces the braces with variables, in order:

```
Hello Dusty, you are currently writing.
```

This basic syntax is not terribly useful if we want to reuse variables within one string or decide to use them in a different position. We can place zero-indexed integers inside the curly braces to tell the formatter which positional variable gets inserted at a given position in the string. Let's repeat the name:

```
template = "Hello {0}, you are {1}. Your name is {0}."
print(template.format('Dusty', 'writing'))
```

If we use these integer indexes, we have to use them in all the variables. We can't mix empty braces with positional indexes. For example, this code fails with an appropriate `ValueError` exception:

```
template = "Hello {}, you are {}. Your name is {0}."
print(template.format('Dusty', 'writing'))
```

Escaping braces

Brace characters are often useful in strings, aside from formatting. We need a way to escape them in situations where we want them to be displayed as themselves, rather than being replaced. This can be done by doubling the braces. For example, we can use Python to format a basic Java program:

```
template = """
public class {0} {{
    public static void main(String[] args) {{
        System.out.println("{1}");
    }}
}}"""

print(template.format("MyClass", "print('hello world')"));
```

Wherever we see the {{ or }} sequence in the template, that is, the braces enclosing the Java class and method definition, we know the `format` method will replace them with single braces, rather than some argument passed into the `format` method. Here's the output:

```
public class MyClass {
    public static void main(String[] args) {
        System.out.println("print('hello world')");
    }
}
```

The class name and contents of the output have been replaced with two parameters, while the double braces have been replaced with single braces, giving us a valid Java file. Turns out, this is about the simplest possible Python program to print the simplest possible Java program that can print the simplest possible Python program!

Keyword arguments

If we're formatting complex strings, it can become tedious to remember the order of the arguments or to update the template if we choose to insert a new argument. The `format` method therefore allows us to specify names inside the braces instead of numbers. The named variables are then passed to the `format` method as keyword arguments:

```
template = """
From: <{from_email}>
To: <{to_email}>
Subject: {subject}

{message}"""
print(template.format(
    from_email = "a@example.com",
    to_email = "b@example.com",
    message = "Here's some mail for you. "
    " Hope you enjoy the message!",
    subject = "You have mail!"
    ))
```

We can also mix index and keyword arguments (as with all Python function calls, the keyword arguments must follow the positional ones). We can even mix unlabeled positional braces with keyword arguments:

```
print("{} {label} {}".format("x", "y", label="z"))
```

As expected, this code outputs:

```
x z y
```

Container lookups

We aren't restricted to passing simple string variables into the `format` method. Any primitive, such as integers or floats can be printed. More interestingly, complex objects, including lists, tuples, dictionaries, and arbitrary objects can be used, and we can access indexes and variables (but not methods) on those objects from within the `format` string.

For example, if our e-mail message had grouped the from and to e-mail addresses into a tuple, and placed the subject and message in a dictionary, for some reason (perhaps because that's the input required for an existing `send_mail` function we want to use), we can format it like this:

```
emails = ("a@example.com", "b@example.com")
message = {
        'subject': "You Have Mail!",
        'message': "Here's some mail for you!"
        }
template = """
From: <{0[0]}>
To: <{0[1]}>
Subject: {message[subject]}
{message[message]}"""
print(template.format(emails, message=message))
```

The variables inside the braces in the template string look a little weird, so let's look at what they're doing. We have passed one argument as a position-based parameter and one as a keyword argument. The two e-mail addresses are looked up by `0[x]`, where `x` is either `0` or `1`. The initial zero represents, as with other position-based arguments, the first positional argument passed to `format` (the `emails` tuple, in this case).

The square brackets with a number inside are the same kind of index lookup we see in regular Python code, so `0[0]` maps to `emails[0]`, in the `emails` tuple. The indexing syntax works with any indexable object, so we see similar behavior when we access `message[subject]`, except this time we are looking up a string key in a dictionary. Notice that unlike in Python code, we do not need to put quotes around the string in the dictionary lookup.

We can even do multiple levels of lookup if we have nested data structures. I would recommend against doing this often, as template strings rapidly become difficult to understand. If we have a dictionary that contains a tuple, we can do this:

```
emails = ("a@example.com", "b@example.com")
message = {
        'emails': emails,
        'subject': "You Have Mail!",
        'message': "Here's some mail for you!"
        }
template = """
From: <{0[emails][0]}>
To: <{0[emails][1]}>
Subject: {0[subject]}
{0[message]}"""
print(template.format(message))
```

Object lookups

Indexing makes `format` lookup powerful, but we're not done yet! We can also pass arbitrary objects as parameters, and use the dot notation to look up attributes on those objects. Let's change our e-mail message data once again, this time to a class:

```
class EMail:
    def __init__(self, from_addr, to_addr, subject, message):
        self.from_addr = from_addr
        self.to_addr = to_addr
        self.subject = subject
        self.message = message

email = EMail("a@example.com", "b@example.com",
        "You Have Mail!",
         "Here's some mail for you!")

template = """
From: <{0.from_addr}>
To: <{0.to_addr}>
Subject: {0.subject}

{0.message}"""
print(template.format(email))
```

The template in this example may be more readable than the previous examples, but the overhead of creating an e-mail class adds complexity to the Python code. It would be foolish to create a class for the express purpose of including the object in a template. Typically, we'd use this sort of lookup if the object we are trying to format already exists. This is true of all the examples; if we have a tuple, list, or dictionary, we'll pass it into the template directly. Otherwise, we'd just create a simple set of positional and keyword arguments.

Making it look right

It's nice to be able to include variables in template strings, but sometimes the variables need a bit of coercion to make them look right in the output. For example, if we are doing calculations with currency, we may end up with a long decimal that we don't want to show up in our template:

```
subtotal = 12.32
tax = subtotal * 0.07
total = subtotal + tax

print("Sub: ${0} Tax: ${1} Total: ${total}".format(
    subtotal, tax, total=total))
```

If we run this formatting code, the output doesn't quite look like proper currency:

```
Sub: $12.32 Tax: $0.8624 Total: $13.182400000000001
```

Technically, we should never use floating-point numbers in currency calculations like this; we should construct `decimal.Decimal()` objects instead. Floats are dangerous because their calculations are inherently inaccurate beyond a specific level of precision. But we're looking at strings, not floats, and currency is a great example for formatting!

To fix the preceding `format` string, we can include some additional information inside the curly braces to adjust the formatting of the parameters. There are tons of things we can customize, but the basic syntax inside the braces is the same; first, we use whichever of the earlier layouts (positional, keyword, index, attribute access) is suitable to specify the variable that we want to place in the template string. We follow this with a colon, and then the specific syntax for the formatting. Here's an improved version:

```
print("Sub: ${0:0.2f} Tax: ${1:0.2f} "
        "Total: ${total:0.2f}".format(
            subtotal, tax, total=total))
```

The `0.2f` format specifier after the colons basically says, from left to right: for values lower than one, make sure a zero is displayed on the left side of the decimal point; show two places after the decimal; format the input value as a float.

We can also specify that each number should take up a particular number of characters on the screen by placing a value before the period in the precision. This can be useful for outputting tabular data, for example:

```
orders = [('burger', 2, 5),
        ('fries', 3.5, 1),
        ('cola', 1.75, 3)]

print("PRODUCT    QUANTITY    PRICE    SUBTOTAL")
for product, price, quantity in orders:
    subtotal = price * quantity
    print("{0:10s}{1: ^9d}    ${2: <8.2f}${3: >7.2f}".format(
        product, quantity, price, subtotal))
```

Ok, that's a pretty scary looking format string, so let's see how it works before we break it down into understandable parts:

```
PRODUCT    QUANTITY    PRICE    SUBTOTAL
burger        5        $2.00    $  10.00
fries         1        $3.50    $   3.50
cola          3        $1.75    $   5.25
```

Nifty! So, how is this actually happening? We have four variables we are formatting, in each line in the `for` loop. The first variable is a string and is formatted with `{0:10s}`. The `s` means it is a string variable, and the `10` means it should take up ten characters. By default, with strings, if the string is shorter than the specified number of characters, it appends spaces to the right side of the string to make it long enough (beware, however: if the original string is too long, it won't be truncated!). We can change this behavior (to fill with other characters or change the alignment in the format string), as we do for the next value, `quantity`.

The formatter for the `quantity` value is `{1: ^9d}`. The `d` represents an integer value. The `9` tells us the value should take up nine characters. But with integers, instead of spaces, the extra characters are zeros, by default. That looks kind of weird. So we explicitly specify a space (immediately after the colon) as a padding character. The caret character ^ tells us that the number should be aligned in the center of this available padding; this makes the column look a bit more professional. The specifiers have to be in the right order, although all are optional: fill first, then align, then the size, and finally, the type.

We do similar things with the specifiers for price and subtotal. For `price`, we use `{2:<8.2f}` and for `subtotal`, `{3:>7.2f}`. In both cases, we're specifying a space as the fill character, but we use the `<` and `>` symbols, respectively, to represent that the numbers should be aligned to the left or right within the minimum space of eight or seven characters. Further, each float should be formatted to two decimal places.

The "type" character for different types can affect formatting output as well. We've seen the `s`, `d`, and `f` types, for strings, integers, and floats. Most of the other format specifiers are alternative versions of these; for example, `o` represents octal format and `X` represents hexadecimal for integers. The `n` type specifier can be useful for formatting integer separators in the current locale's format. For floating-point numbers, the `%` type will multiply by 100 and format a float as a percentage.

While these standard formatters apply to most built-in objects, it is also possible for other objects to define nonstandard specifiers. For example, if we pass a `datetime` object into `format`, we can use the specifiers used in the `datetime.strftime` function, as follows:

```
import datetime
print("{0:%Y-%m-%d %I:%M%p }".format(
    datetime.datetime.now()))
```

It is even possible to write custom formatters for objects we create ourselves, but that is beyond the scope of this book. Look into overriding the `__format__` special method if you need to do this in your code. The most comprehensive instructions can be found in PEP 3101 at `http://www.python.org/dev/peps/pep-3101/`, although the details are a bit dry. You can find more digestible tutorials using a web search.

The Python formatting syntax is quite flexible but it is a difficult mini-language to remember. I use it every day and still occasionally have to look up forgotten concepts in the documentation. It also isn't powerful enough for serious templating needs, such as generating web pages. There are several third-party templating libraries you can look into if you need to do more than basic formatting of a few strings.

Strings are Unicode

At the beginning of this section, we defined strings as collections of immutable Unicode characters. This actually makes things very complicated at times, because Unicode isn't really a storage format. If you get a string of bytes from a file or a socket, for example, they won't be in Unicode. They will, in fact, be the built-in type `bytes`. Bytes are immutable sequences of... well, bytes. Bytes are the lowest-level storage format in computing. They represent 8 bits, usually described as an integer between 0 and 255, or a hexadecimal equivalent between 0 and FF. Bytes don't represent anything specific; a sequence of bytes may store characters of an encoded string, or pixels in an image.

If we print a byte object, any bytes that map to ASCII representations will be printed as their original character, while non-ASCII bytes (whether they are binary data or other characters) are printed as hex codes escaped by the `\x` escape sequence. You may find it odd that a byte, represented as an integer, can map to an ASCII character. But ASCII is really just a code where each letter is represented by a different byte pattern, and therefore, a different integer. The character "a" is represented by the same byte as the integer 97, which is the hexadecimal number 0x61. Specifically, all of these are an interpretation of the binary pattern 01100001.

Many I/O operations only know how to deal with `bytes`, even if the bytes object refers to textual data. It is therefore vital to know how to convert between `bytes` and Unicode.

The problem is that there are many ways to map `bytes` to Unicode text. Bytes are machine-readable values, while text is a human-readable format. Sitting in between is an encoding that maps a given sequence of bytes to a given sequence of text characters.

However, there are multiple such encodings (ASCII is only one of them). The same sequence of bytes represents completely different text characters when mapped using different encodings! So, `bytes` must be decoded using the same character set with which they were encoded. It's not possible to get text from bytes without knowing how the bytes should be decoded. If we receive unknown bytes without a specified encoding, the best we can do is guess what format they are encoded in, and we may be wrong.

Converting bytes to text

If we have an array of `bytes` from somewhere, we can convert it to Unicode using the `.decode` method on the `bytes` class. This method accepts a string for the name of the character encoding. There are many such names; common ones for Western languages include ASCII, UTF-8, and latin-1.

The sequence of bytes (in hex), 63 6c 69 63 68 e9, actually represents the characters of the word cliché in the latin-1 encoding. The following example will encode this sequence of bytes and convert it to a Unicode string using the latin-1 encoding:

```
characters = b'\x63\x6c\x69\x63\x68\xe9'
print(characters)
print(characters.decode("latin-1"))
```

The first line creates a `bytes` object; the `b` character immediately before the string tells us that we are defining a `bytes` object instead of a normal Unicode string. Within the string, each byte is specified using—in this case—a hexadecimal number. The `\x` character escapes within the byte string, and each say, "the next two characters represent a byte using hexadecimal digits."

Provided we are using a shell that understands the latin-1 encoding, the two `print` calls will output the following strings:

```
b'clich\xe9'
cliché
```

The first `print` statement renders the bytes for ASCII characters as themselves. The unknown (unknown to ASCII, that is) character stays in its escaped hex format. The output includes a `b` character at the beginning of the line to remind us that it is a `bytes` representation, not a string.

The next call decodes the string using latin-1 encoding. The `decode` method returns a normal (Unicode) string with the correct characters. However, if we had decoded this same string using the Cyrillic "iso8859-5" encoding, we'd have ended up with the string 'clichщ'! This is because the `\xe9` byte maps to different characters in the two encodings.

Converting text to bytes

If we need to convert incoming bytes into Unicode, clearly we're also going to have situations where we convert outgoing Unicode into byte sequences. This is done with the `encode` method on the `str` class, which, like the `decode` method, requires a character set. The following code creates a Unicode string and encodes it in different character sets:

```
characters = "cliché"
print(characters.encode("UTF-8"))
print(characters.encode("latin-1"))
print(characters.encode("CP437"))
print(characters.encode("ascii"))
```

The first three encodings create a different set of bytes for the accented character. The fourth one can't even handle that byte:

```
b'clich\xc3\xa9'
b'clich\xe9'
b'clich\x82'
Traceback (most recent call last):
```

```
  File "1261_10_16_decode_unicode.py", line 5, in <module>
    print(characters.encode("ascii"))
UnicodeEncodeError: 'ascii' codec can't encode character '\xe9' in
position 5: ordinal not in range(128)
```

Do you understand the importance of encoding now? The accented character is represented as a different byte for each encoding; if we use the wrong one when we are decoding bytes to text, we get the wrong character.

The exception in the last case is not always the desired behavior; there may be cases where we want the unknown characters to be handled in a different way. The `encode` method takes an optional string argument named `errors` that can define how such characters should be handled. This string can be one of the following:

- `strict`
- `replace`
- `ignore`
- `xmlcharrefreplace`

The `strict` replacement strategy is the default we just saw. When a byte sequence is encountered that does not have a valid representation in the requested encoding, an exception is raised. When the `replace` strategy is used, the character is replaced with a different character; in ASCII, it is a question mark; other encodings may use different symbols, such as an empty box. The `ignore` strategy simply discards any bytes it doesn't understand, while the `xmlcharrefreplace` strategy creates an `xml` entity representing the Unicode character. This can be useful when converting unknown strings for use in an XML document. Here's how each of the strategies affects our sample word:

Strategy	"cliché".encode("ascii", strategy)
`replace`	`b'clich?'`
`ignore`	`b'clich'`
`xmlcharrefreplace`	`b'clich&#233;'`

It is possible to call the `str.encode` and `bytes.decode` methods without passing an encoding string. The encoding will be set to the default encoding for the current platform. This will depend on the current operating system and locale or regional settings; you can look it up using the `sys.getdefaultencoding()` function. It is usually a good idea to specify the encoding explicitly, though, since the default encoding for a platform may change, or the program may one day be extended to work on text from a wider variety of sources.

If you are encoding text and don't know which encoding to use, it is best to use the UTF-8 encoding. UTF-8 is able to represent any Unicode character. In modern software, it is a de facto standard encoding to ensure documents in any language—or even multiple languages—can be exchanged. The various other possible encodings are useful for legacy documents or in regions that still use different character sets by default.

The UTF-8 encoding uses one byte to represent ASCII and other common characters, and up to four bytes for more complex characters. UTF-8 is special because it is backwards-compatible with ASCII; any ASCII document encoded using UTF-8 will be identical to the original ASCII document.

I can never remember whether to use `encode` or `decode` to convert from binary bytes to Unicode. I always wished these methods were named "to_binary" and "from_binary" instead. If you have the same problem, try mentally replacing the word "code" with "binary"; "enbinary" and "debinary" are pretty close to "to_binary" and "from_binary". I have saved a lot of time by not looking up the method help files since devising this mnemonic.

Mutable byte strings

The `bytes` type, like `str`, is immutable. We can use index and slice notation on a `bytes` object and search for a particular sequence of bytes, but we can't extend or modify them. This can be very inconvenient when dealing with I/O, as it is often necessary to buffer incoming or outgoing bytes until they are ready to be sent. For example, if we are receiving data from a socket, it may take several `recv` calls before we have received an entire message.

This is where the `bytearray` built-in comes in. This type behaves something like a list, except it only holds bytes. The constructor for the class can accept a `bytes` object to initialize it. The `extend` method can be used to append another `bytes` object to the existing array (for example, when more data comes from a socket or other I/O channel).

Slice notation can be used on `bytearray` to modify the item inline. For example, this code constructs a `bytearray` from a `bytes` object and then replaces two bytes:

```
b = bytearray(b"abcdefgh")
b[4:6] = b"\x15\xa3"
print(b)
```

The output looks like this:

```
bytearray(b'abcd\x15\xa3gh')
```

Be careful; if we want to manipulate a single element in the `bytearray`, it will expect us to pass an integer between 0 and 255 inclusive as the value. This integer represents a specific `bytes` pattern. If we try to pass a character or `bytes` object, it will raise an exception.

A single byte character can be converted to an integer using the `ord` (short for ordinal) function. This function returns the integer representation of a single character:

```
b = bytearray(b'abcdef')
b[3] = ord(b'g')
b[4] = 68
print(b)
```

The output looks like this:

```
bytearray(b'abcgDf')
```

After constructing the array, we replace the character at index `3` (the fourth character, as indexing starts at `0`, as with lists) with byte 103. This integer was returned by the `ord` function and is the ASCII character for the lowercase `g`. For illustration, we also replaced the next character up with the byte number `68`, which maps to the ASCII character for the uppercase `D`.

The `bytearray` type has methods that allow it to behave like a list (we can append integer bytes to it, for example), but also like a `bytes` object; we can use methods like `count` and `find` the same way they would behave on a `bytes` or `str` object. The difference is that `bytearray` is a mutable type, which can be useful for building up complex sequences of bytes from a specific input source.

Regular expressions

You know what's really hard to do using object-oriented principles? Parsing strings to match arbitrary patterns, that's what. There have been a fair number of academic papers written in which object-oriented design is used to set up string parsing, but the result is always very verbose and hard to read, and they are not widely used in practice.

In the real world, string parsing in most programming languages is handled by regular expressions. These are not verbose, but, boy, are they ever hard to read, at least until you learn the syntax. Even though regular expressions are not object oriented, the Python regular expression library provides a few classes and objects that you can use to construct and run regular expressions.

Regular expressions are used to solve a common problem: Given a string, determine whether that string matches a given pattern and, optionally, collect substrings that contain relevant information. They can be used to answer questions like:

- Is this string a valid URL?
- What is the date and time of all warning messages in a log file?
- Which users in `/etc/passwd` are in a given group?
- What username and document were requested by the URL a visitor typed?

There are many similar scenarios where regular expressions are the correct answer. Many programmers have made the mistake of implementing complicated and fragile string parsing libraries because they didn't know or wouldn't learn regular expressions. In this section, we'll gain enough knowledge of regular expressions to not make such mistakes!

Matching patterns

Regular expressions are a complicated mini-language. They rely on special characters to match unknown strings, but let's start with literal characters, such as letters, numbers, and the space character, which always match themselves. Let's see a basic example:

```
import re

search_string = "hello world"
pattern = "hello world"

match = re.match(pattern, search_string)

if match:
    print("regex matches")
```

The Python Standard Library module for regular expressions is called `re`. We import it and set up a search string and pattern to search for; in this case, they are the same string. Since the search string matches the given pattern, the conditional passes and the `print` statement executes.

Bear in mind that the `match` function matches the pattern to the beginning of the string. Thus, if the pattern were `"ello world"`, no match would be found. With confusing asymmetry, the parser stops searching as soon as it finds a match, so the pattern `"hello wo"` matches successfully. Let's build a small example program to demonstrate these differences and help us learn other regular expression syntax:

```
import sys
```

```
import re

pattern = sys.argv[1]
search_string = sys.argv[2]
match = re.match(pattern, search_string)

if match:
    template = "'{}' matches pattern '{}'"
else:
    template = "'{}' does not match pattern '{}'"

print(template.format(search_string, pattern))
```

This is just a generic version of the earlier example that accepts the pattern and search string from the command line. We can see how the start of the pattern must match, but a value is returned as soon as a match is found in the following command-line interaction:

```
$ python regex_generic.py "hello worl" "hello world"
'hello world' matches pattern 'hello worl'
$ python regex_generic.py "ello world" "hello world"
'hello world' does not match pattern 'ello world'
```

We'll be using this script throughout the next few sections. While the script is always invoked with the command line `python regex_generic.py "<pattern>" "<string>"`, we'll only see the output in the following examples, to conserve space.

If you need control over whether items happen at the beginning or end of a line (or if there are no newlines in the string, at the beginning and end of the string), you can use the `^` and `$` characters to represent the start and end of the string respectively. If you want a pattern to match an entire string, it's a good idea to include both of these:

```
'hello world' matches pattern '^hello world$'
'hello worl' does not match pattern '^hello world$'
```

Matching a selection of characters

Let's start with matching an arbitrary character. The period character, when used in a regular expression pattern, can match any single character. Using a period in the string means you don't care what the character is, just that there is a character there. For example:

```
'hello world' matches pattern 'hel.o world'
'helpo world' matches pattern 'hel.o world'
```

```
'hel o world' matches pattern 'hel.o world'
'helo world' does not match pattern 'hel.o world'
```

Notice how the last example does not match because there is no character at the period's position in the pattern.

That's all well and good, but what if we only want a few specific characters to match? We can put a set of characters inside square brackets to match any one of those characters. So if we encounter the string `[abc]` in a regular expression pattern, we know that those five (including the two square brackets) characters will only match one character in the string being searched, and further, that this one character will be either an `a`, a `b`, or a `c`. See a few examples:

```
'hello world' matches pattern 'hel[lp]o world'
'helpo world' matches pattern 'hel[lp]o world'
'helPo world' does not match pattern 'hel[lp]o world'
```

These square bracket sets should be named character sets, but they are more often referred to as **character classes**. Often, we want to include a large range of characters inside these sets, and typing them all out can be monotonous and error-prone. Fortunately, the regular expression designers thought of this and gave us a shortcut. The dash character, in a character set, will create a range. This is especially useful if you want to match "all lower case letters", "all letters", or "all numbers" as follows:

```
'hello   world' does not match pattern 'hello [a-z] world'
'hello b world' matches pattern 'hello [a-z] world'
'hello B world' matches pattern 'hello [a-zA-Z] world'
'hello 2 world' matches pattern 'hello [a-zA-Z0-9] world'
```

There are other ways to match or exclude individual characters, but you'll need to find a more comprehensive tutorial via a web search if you want to find out what they are!

Escaping characters

If putting a period character in a pattern matches any arbitrary character, how do we match just a period in a string? One way might be to put the period inside square brackets to make a character class, but a more generic method is to use backslashes to escape it. Here's a regular expression to match two digit decimal numbers between 0.00 and 0.99:

```
'0.05' matches pattern '0\.[0-9][0-9]'
'005' does not match pattern '0\.[0-9][0-9]'
'0,05' does not match pattern '0\.[0-9][0-9]'
```

For this pattern, the two characters `\.` match the single `.` character. If the period character is missing or is a different character, it does not match.

This backslash escape sequence is used for a variety of special characters in regular expressions. You can use `\[` to insert a square bracket without starting a character class, and `\(` to insert a parenthesis, which we'll later see is also a special character.

More interestingly, we can also use the escape symbol followed by a character to represent special characters such as newlines (`\n`), and tabs (`\t`). Further, some character classes can be represented more succinctly using escape strings; `\s` represents whitespace characters, `\w` represents letters, numbers, and underscore, and `\d` represents a digit:

```
'(abc]' matches pattern '\(abc\]'
' 1a' matches pattern '\s\d\w'
'\t5n' does not match pattern '\s\d\w'
'5n' matches pattern '\s\d\w'
```

Matching multiple characters

With this information, we can match most strings of a known length, but most of the time we don't know how many characters to match inside a pattern. Regular expressions can take care of this, too. We can modify a pattern by appending one of several hard-to-remember punctuation symbols to match multiple characters.

The asterisk (`*`) character says that the previous pattern can be matched zero or more times. This probably sounds silly, but it's one of the most useful repetition characters. Before we explore why, consider some silly examples to make sure we understand what it does:

```
'hello' matches pattern 'hel*o'
'heo' matches pattern 'hel*o'
'hellllllo' matches pattern 'hel*o'
```

So, the `*` character in the pattern says that the previous pattern (the `l` character) is optional, and if present, can be repeated as many times as possible to match the pattern. The rest of the characters (`h`, `e`, and `o`) have to appear exactly once.

It's pretty rare to want to match a single letter multiple times, but it gets more interesting if we combine the asterisk with patterns that match multiple characters. `.*`, for example, will match any string, whereas `[a-z]*` matches any collection of lowercase words, including the empty string.

For example:

```
'A string.' matches pattern '[A-Z][a-z]* [a-z]*\.'
'No .' matches pattern '[A-Z][a-z]* [a-z]*\.'
'' matches pattern '[a-z]*.*'
```

The plus (+) sign in a pattern behaves similarly to an asterisk; it states that the previous pattern can be repeated one or more times, but, unlike the asterisk is not optional. The question mark (?) ensures a pattern shows up exactly zero or one times, but not more. Let's explore some of these by playing with numbers (remember that `\d` matches the same character class as `[0-9]`:

```
'0.4' matches pattern '\d+\.\d+'
'1.002' matches pattern '\d+\.\d+'
'1.' does not match pattern '\d+\.\d+'
'1%' matches pattern '\d?\d%'
'99%' matches pattern '\d?\d%'
'999%' does not match pattern '\d?\d%'
```

Grouping patterns together

So far we've seen how we can repeat a pattern multiple times, but we are restricted in what patterns we can repeat. If we want to repeat individual characters, we're covered, but what if we want a repeating sequence of characters? Enclosing any set of patterns in parenthesis allows them to be treated as a single pattern when applying repetition operations. Compare these patterns:

```
'abccc' matches pattern 'abc{3}'
'abccc' does not match pattern '(abc){3}'
'abcabcabc' matches pattern '(abc){3}'
```

Combined with complex patterns, this grouping feature greatly expands our pattern-matching repertoire. Here's a regular expression that matches simple English sentences:

```
'Eat.' matches pattern '[A-Z][a-z]*( [a-z]+)*\.$'
'Eat more good food.' matches pattern '[A-Z][a-z]*( [a-z]+)*\.$'
'A good meal.' matches pattern '[A-Z][a-z]*( [a-z]+)*\.$'
```

The first word starts with a capital, followed by zero or more lowercase letters. Then, we enter a parenthetical that matches a single space followed by a word of one or more lowercase letters. This entire parenthetical is repeated zero or more times, and the pattern is terminated with a period. There cannot be any other characters after the period, as indicated by the `$` matching the end of string.

We've seen many of the most basic patterns, but the regular expression language supports many more. I spent my first few years using regular expressions looking up the syntax every time I needed to do something. It is worth bookmarking Python's documentation for the `re` module and reviewing it frequently. There are very few things that regular expressions cannot match, and they should be the first tool you reach for when parsing strings.

Getting information from regular expressions

Let's now focus on the Python side of things. The regular expression syntax is the furthest thing from object-oriented programming. However, Python's `re` module provides an object-oriented interface to enter the regular expression engine.

We've been checking whether the `re.match` function returns a valid object or not. If a pattern does not match, that function returns `None`. If it does match, however, it returns a useful object that we can introspect for information about the pattern.

So far, our regular expressions have answered questions such as "Does this string match this pattern?" Matching patterns is useful, but in many cases, a more interesting question is, "If this string matches this pattern, what is the value of a relevant substring?" If you use groups to identify parts of the pattern that you want to reference later, you can get them out of the match return value as illustrated in the next example:

```
pattern = "^[a-zA-Z.]+@([a-z.]*\.[a-z]+)$"
search_string = "some.user@example.com"
match = re.match(pattern, search_string)

if match:
    domain = match.groups()[0]
    print(domain)
```

The specification describing valid e-mail addresses is extremely complicated, and the regular expression that accurately matches all possibilities is obscenely long. So we cheated and made a simple regular expression that matches some common e-mail addresses; the point is that we want to access the domain name (after the `@` sign) so we can connect to that address. This is done easily by wrapping that part of the pattern in parenthesis and calling the `groups()` method on the object returned by match.

The `groups` method returns a tuple of all the groups matched inside the pattern, which you can index to access a specific value. The groups are ordered from left to right. However, bear in mind that groups can be nested, meaning you can have one or more groups inside another group. In this case, the groups are returned in the order of their left-most brackets, so the outermost group will be returned before its inner matching groups.

In addition to the match function, the `re` module provides a couple other useful functions, `search`, and `findall`. The `search` function finds the first instance of a matching pattern, relaxing the restriction that the pattern start at the first letter of the string. Note that you can get a similar effect by using match and putting a `^.*` character at the front of the pattern to match any characters between the start of the string and the pattern you are looking for.

The `findall` function behaves similarly to search, except that it finds all non-overlapping instances of the matching pattern, not just the first one. Basically, it finds the first match, then it resets the search to the end of that matching string and finds the next one.

Instead of returning a list of match objects, as you would expect, it returns a list of matching strings. Or tuples. Sometimes it's strings, sometimes it's tuples. It's not a very good API at all! As with all bad APIs, you'll have to memorize the differences and not rely on intuition. The type of the return value depends on the number of bracketed groups inside the regular expression:

- If there are no groups in the pattern, `re.findall` will return a list of strings, where each value is a complete substring from the source string that matches the pattern
- If there is exactly one group in the pattern, `re.findall` will return a list of strings where each value is the contents of that group
- If there are multiple groups in the pattern, then `re.findall` will return a list of tuples where each tuple contains a value from a matching group, in order

When you are designing function calls in your own Python libraries, try to make the function always return a consistent data structure. It is often good to design functions that can take arbitrary inputs and process them, but the return value should not switch from single value to a list, or a list of values to a list of tuples depending on the input. Let `re.findall` be a lesson!

The examples in the following interactive session will hopefully clarify the differences:

```
>>> import re
>>> re.findall('a.', 'abacadefagah')
['ab', 'ac', 'ad', 'ag', 'ah']
>>> re.findall('a(.)', 'abacadefagah')
['b', 'c', 'd', 'g', 'h']
>>> re.findall('(a)(.)', 'abacadefagah')
```

```
[('a', 'b'), ('a', 'c'), ('a', 'd'), ('a', 'g'), ('a', 'h')]
>>> re.findall('((a)(.))', 'abacadefagah')
[('ab', 'a', 'b'), ('ac', 'a', 'c'), ('ad', 'a', 'd'), ('ag', 'a', 'g'),
('ah', 'a', 'h')]
```

Making repeated regular expressions efficient

Whenever you call one of the regular expression methods, the engine has to convert the pattern string into an internal structure that makes searching strings fast. This conversion takes a non-trivial amount of time. If a regular expression pattern is going to be reused multiple times (for example, inside a `for` or `while` loop), it would be better if this conversion step could be done only once.

This is possible with the `re.compile` method. It returns an object-oriented version of the regular expression that has been compiled down and has the methods we've explored (`match`, `search`, `findall`) already, among others. We'll see examples of this in the case study.

This has definitely been a condensed introduction to regular expressions. At this point, we have a good feel for the basics and will recognize when we need to do further research. If we have a string pattern matching problem, regular expressions will almost certainly be able to solve them for us. However, we may need to look up new syntaxes in a more comprehensive coverage of the topic. But now we know what to look for! Let's move on to a completely different topic: serializing data for storage.

Serializing objects

Nowadays, we take the ability to write data to a file and retrieve it at an arbitrary later date for granted. As convenient as this is (imagine the state of computing if we couldn't store anything!), we often find ourselves converting data we have stored in a nice object or design pattern in memory into some kind of clunky text or binary format for storage, transfer over the network, or remote invocation on a distant server.

The Python `pickle` module is an object-oriented way to store objects directly in a special storage format. It essentially converts an object (and all the objects it holds as attributes) into a sequence of bytes that can be stored or transported however we see fit.

For basic work, the `pickle` module has an extremely simple interface. It is comprised of four basic functions for storing and loading data; two for manipulating file-like objects, and two for manipulating `bytes` objects (the latter are just shortcuts to the file-like interface, so we don't have to create a `BytesIO` file-like object ourselves).

The `dump` method accepts an object to be written and a file-like object to write the serialized bytes to. This object must have a `write` method (or it wouldn't be file-like), and that method must know how to handle a `bytes` argument (so a file opened for text output wouldn't work).

The `load` method does exactly the opposite; it reads a serialized object from a file-like object. This object must have the proper file-like `read` and `readline` arguments, each of which must, of course, return `bytes`. The `pickle` module will load the object from these bytes and the `load` method will return the fully reconstructed object. Here's an example that stores and then loads some data in a list object:

```
import pickle

some_data = ["a list", "containing", 5,
        "values including another list",
        ["inner", "list"]]

with open("pickled_list", 'wb') as file:
    pickle.dump(some_data, file)

with open("pickled_list", 'rb') as file:
    loaded_data = pickle.load(file)

print(loaded_data)
assert loaded_data == some_data
```

This code works as advertised: the objects are stored in the file and then loaded from the same file. In each case, we open the file using a `with` statement so that it is automatically closed. The file is first opened for writing and then a second time for reading, depending on whether we are storing or loading data.

The `assert` statement at the end would raise an error if the newly loaded object is not equal to the original object. Equality does not imply that they are the same object. Indeed, if we print the `id()` of both objects, we would discover they are different. However, because they are both lists whose contents are equal, the two lists are also considered equal.

The `dumps` and `loads` functions behave much like their file-like counterparts, except they return or accept `bytes` instead of file-like objects. The `dumps` function requires only one argument, the object to be stored, and it returns a serialized `bytes` object. The `loads` function requires a `bytes` object and returns the restored object. The `'s'` character in the method names is short for string; it's a legacy name from ancient versions of Python, where `str` objects were used instead of `bytes`.

Both `dump` methods accept an optional `protocol` argument. If we are saving and loading pickled objects that are only going to be used in Python 3 programs, we don't need to supply this argument. Unfortunately, if we are storing objects that may be loaded by older versions of Python, we have to use an older and less efficient protocol. This should not normally be an issue. Usually, the only program that would load a pickled object would be the same one that stored it. Pickle is an unsafe format, so we don't want to be sending it unsecured over the Internet to unknown interpreters.

The argument supplied is an integer version number. The default version is number 3, representing the current highly efficient storage system used by Python 3 pickling. The number 2 is the older version, which will store an object that can be loaded on all interpreters back to Python 2.3. As 2.6 is the oldest of Python that is still widely used in the wild, version 2 pickling is normally sufficient. Versions 0 and 1 are supported on older interpreters; 0 is an ASCII format, while 1 is a binary format. There is also an optimized version 4 that may one day become the default.

As a rule of thumb, then, if you know that the objects you are pickling will only be loaded by a Python 3 program (for example, only your program will be loading them), use the default pickling protocol. If they may be loaded by unknown interpreters, pass a protocol value of 2, unless you really believe they may need to be loaded by an archaic version of Python.

If we do pass a protocol to `dump` or `dumps`, we should use a keyword argument to specify it: `pickle.dumps(my_object, protocol=2)`. This is not strictly necessary, as the method only accepts two arguments, but typing out the full keyword argument reminds readers of our code what the purpose of the number is. Having a random integer in the method call would be hard to read. Two what? Store two copies of the object, maybe? Remember, code should always be readable. In Python, less code is often more readable than longer code, but not always. Be explicit.

It is possible to call `dump` or `load` on a single open file more than once. Each call to `dump` will store a single object (plus any objects it is composed of or contains), while a call to `load` will load and return just one object. So for a single file, each separate call to `dump` when storing the object should have an associated call to `load` when restoring at a later date.

Customizing pickles

With most common Python objects, pickling "just works". Basic primitives such as integers, floats, and strings can be pickled, as can any container object, such as lists or dictionaries, provided the contents of those containers are also picklable. Further, and importantly, any object can be pickled, so long as all of its attributes are also picklable.

So what makes an attribute unpicklable? Usually, it has something to do with time-sensitive attributes that it would not make sense to load in the future. For example, if we have an open network socket, open file, running thread, or database connection stored as an attribute on an object, it would not make sense to pickle these objects; a lot of operating system state would simply be gone when we attempted to reload them later. We can't just pretend a thread or socket connection exists and make it appear! No, we need to somehow customize how such transient data is stored and restored.

Here's a class that loads the contents of a web page every hour to ensure that they stay up to date. It uses the `threading.Timer` class to schedule the next update:

```
from threading import Timer
import datetime
from urllib.request import urlopen

class UpdatedURL:
    def __init__(self, url):
        self.url = url
        self.contents = ''
        self.last_updated = None
        self.update()

    def update(self):
        self.contents = urlopen(self.url).read()
        self.last_updated = datetime.datetime.now()
        self.schedule()

    def schedule(self):
        self.timer = Timer(3600, self.update)
        self.timer.setDaemon(True)
        self.timer.start()
```

The `url`, `contents`, and `last_updated` are all pickleable, but if we try to pickle an instance of this class, things go a little nutty on the `self.timer` instance:

```
>>> u = UpdatedURL("http://news.yahoo.com/")
>>> import pickle
>>> serialized = pickle.dumps(u)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    serialized = pickle.dumps(u)
_pickle.PicklingError: Can't pickle <class '_thread.lock'>: attribute
lookup lock on _thread failed
```

That's not a very useful error, but it looks like we're trying to pickle something we shouldn't be. That would be the `Timer` instance; we're storing a reference to `self.timer` in the schedule method, and that attribute cannot be serialized.

When `pickle` tries to serialize an object, it simply tries to store the object's `__dict__` attribute; `__dict__` is a dictionary mapping all the attribute names on the object to their values. Luckily, before checking `__dict__`, `pickle` checks to see whether a `__getstate__` method exists. If it does, it will store the return value of that method instead of the `__dict__`.

Let's add a `__getstate__` method to our `UpdatedURL` class that simply returns a copy of the `__dict__` without a timer:

```
def __getstate__(self):
    new_state = self.__dict__.copy()
    if 'timer' in new_state:
        del new_state['timer']
    return new_state
```

If we pickle the object now, it will no longer fail. And we can even successfully restore that object using `loads`. However, the restored object doesn't have a timer attribute, so it will not be refreshing the content like it is designed to do. We need to somehow create a new timer (to replace the missing one) when the object is unpickled.

As we might expect, there is a complementary `__setstate__` method that can be implemented to customize unpickling. This method accepts a single argument, which is the object returned by `__getstate__`. If we implement both methods, `__getstate__` is not required to return a dictionary, since `__setstate__` will know what to do with whatever object `__getstate__` chooses to return. In our case, we simply want to restore the `__dict__`, and then create a new timer:

```
def __setstate__(self, data):
    self.__dict__ = data
    self.schedule()
```

The `pickle` module is very flexible and provides other tools to further customize the pickling process if you need them. However, these are beyond the scope of this book. The tools we've covered are sufficient for many basic pickling tasks. Objects to be pickled are normally relatively simple data objects; we would not likely pickle an entire running program or complicated design pattern, for example.

Serializing web objects

It is not a good idea to load a pickled object from an unknown or untrusted source. It is possible to inject arbitrary code into a pickled file to maliciously attack a computer via the pickle. Another disadvantage of pickles is that they can only be loaded by other Python programs, and cannot be easily shared with services written in other languages.

There are many formats that have been used for this purpose over the years. XML (Extensible Markup Language) used to be very popular, especially with Java developers. YAML (Yet Another Markup Language) is another format that you may see referenced occasionally. Tabular data is frequently exchanged in the CSV (Comma Separated Value) format. Many of these are fading into obscurity and there are many more that you will encounter over time. Python has solid standard or third-party libraries for all of them.

Before using such libraries on untrusted data, make sure to investigate security concerns with each of them. XML and YAML, for example, both have obscure features that, used maliciously, can allow arbitrary commands to be executed on the host machine. These features may not be turned off by default. Do your research.

JavaScript Object Notation (**JSON**) is a human readable format for exchanging primitive data. JSON is a standard format that can be interpreted by a wide array of heterogeneous client systems. Hence, JSON is extremely useful for transmitting data between completely decoupled systems. Further, JSON does not have any support for executable code, only data can be serialized; thus, it is more difficult to inject malicious statements into it.

Because JSON can be easily interpreted by JavaScript engines, it is often used for transmitting data from a web server to a JavaScript-capable web browser. If the web application serving the data is written in Python, it needs a way to convert internal data into the JSON format.

There is a module to do this, predictably named `json`. This module provides a similar interface to the `pickle` module, with `dump`, `load`, `dumps`, and `loads` functions. The default calls to these functions are nearly identical to those in `pickle`, so let us not repeat the details. There are a couple differences; obviously, the output of these calls is valid JSON notation, rather than a pickled object. In addition, the `json` functions operate on `str` objects, rather than `bytes`. Therefore, when dumping to or loading from a file, we need to create text files rather than binary ones.

The JSON serializer is not as robust as the `pickle` module; it can only serialize basic types such as integers, floats, and strings, and simple containers such as dictionaries and lists. Each of these has a direct mapping to a JSON representation, but JSON is unable to represent classes, methods, or functions. It is not possible to transmit complete objects in this format. Because the receiver of an object we have dumped to JSON format is normally not a Python object, it would not be able to understand classes or methods in the same way that Python does, anyway. In spite of the O for Object in its name, JSON is a **data** notation; objects, as you recall, are composed of both data and behavior.

If we do have objects for which we want to serialize only the data, we can always serialize the object's `__dict__` attribute. Or we can semiautomate this task by supplying custom code to create or parse a JSON serializable dictionary from certain types of objects.

In the `json` module, both the object storing and loading functions accept optional arguments to customize the behavior. The `dump` and `dumps` methods accept a poorly named `cls` (short for class, which is a reserved keyword) keyword argument. If passed, this should be a subclass of the `JSONEncoder` class, with the `default` method overridden. This method accepts an arbitrary object and converts it to a dictionary that `json` can digest. If it doesn't know how to process the object, we should call the `super()` method, so that it can take care of serializing basic types in the normal way.

The `load` and `loads` methods also accept such a `cls` argument that can be a subclass of the inverse class, `JSONDecoder`. However, it is normally sufficient to pass a function into these methods using the `object_hook` keyword argument. This function accepts a dictionary and returns an object; if it doesn't know what to do with the input dictionary, it can return it unmodified.

Let's look at an example. Imagine we have the following simple contact class that we want to serialize:

```
class Contact:
    def __init__(self, first, last):
        self.first = first
        self.last = last

    @property
    def full_name(self):
        return("{} {}".format(self.first, self.last))
```

We could just serialize the `__dict__` attribute:

```
>>> c = Contact("John", "Smith")
>>> json.dumps(c.__dict__)
'{"last": "Smith", "first": "John"}'
```

But accessing special (double-underscore) attributes in this fashion is kind of crude. Also, what if the receiving code (perhaps some JavaScript on a web page) wanted that `full_name` property to be supplied? Of course, we could construct the dictionary by hand, but let's create a custom encoder instead:

```
import json
class ContactEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, Contact):
            return {'is_contact': True,
                    'first': obj.first,
                    'last': obj.last,
                    'full': obj.full_name}
        return super().default(obj)
```

The `default` method basically checks to see what kind of object we're trying to serialize; if it's a contact, we convert it to a dictionary manually; otherwise, we let the parent class handle serialization (by assuming that it is a basic type, which `json` knows how to handle). Notice that we pass an extra attribute to identify this object as a contact, since there would be no way to tell upon loading it. This is just a convention; for a more generic serialization mechanism, it might make more sense to store a string type in the dictionary, or possibly even the full class name, including package and module. Remember that the format of the dictionary depends on the code at the receiving end; there has to be an agreement as to how the data is going to be specified.

We can use this class to encode a contact by passing the class (not an instantiated object) to the `dump` or `dumps` function:

```
>>> c = Contact("John", "Smith")
>>> json.dumps(c, cls=ContactEncoder)
'{"is_contact": true, "last": "Smith", "full": "John Smith",
"first": "John"}'
```

For decoding, we can write a function that accepts a dictionary and checks the existence of the `is_contact` variable to decide whether to convert it to a contact:

```
def decode_contact(dic):
        if dic.get('is_contact'):
            return Contact(dic['first'], dic['last'])
        else:
            return dic
```

We can pass this function to the `load` or `loads` function using the `object_hook` keyword argument:

```
>>> data = ('{"is_contact": true, "last": "smith",'
     '"full": "john smith", "first": "john"}')

>>> c = json.loads(data, object_hook=decode_contact)
>>> c
<__main__.Contact object at 0xa02918c>
>>> c.full_name
'john smith'
```

Case study

Let's build a basic regular expression-powered templating engine in Python. This engine will parse a text file (such as an HTML page) and replace certain directives with text calculated from the input to those directives. This is about the most complicated task we would want to do with regular expressions; indeed, a full-fledged version of this would likely utilize a proper language parsing mechanism.

Consider the following input file:

```
/** include header.html **/
<h1>This is the title of the front page</h1>
/** include menu.html **/
<p>My name is /** variable name **/.
This is the content of my front page. It goes below the menu.</p>
<table>
<tr><th>Favourite Books</th></tr>
/** loopover book_list **/
<tr><td>/** loopvar **/</td></tr>

/** endloop **/
</table>
/** include footer.html **/
Copyright &copy; Today
```

This file contains "tags" of the form `/** <directive> <data> **/` where the data is an optional single word and the directives are:

- `include`: Copy the contents of another file here
- `variable`: Insert the contents of a variable here
- `loopover`: Repeat the contents of the loop for a variable that is a list
- `endloop`: Signal the end of looped text
- `loopvar`: Insert a single value from the list being looped over

This template will render a different page depending which variables are passed into it. These variables will be passed in from a so-called context file. This will be encoded as a `json` object with keys representing the variables in question. My context file might look like this, but you would derive your own:

```
{
    "name": "Dusty",
    "book_list": [
        "Thief Of Time",
        "The Thief",
        "Snow Crash",
        "Lathe Of Heaven"
    ]
}
```

Before we get into the actual string processing, let's throw together some object-oriented boilerplate code for processing files and grabbing data from the command line:

```
import re
import sys
import json
from pathlib import Path

DIRECTIVE_RE = re.compile(
    r'/\*\*\s*(include|variable|loopover|endloop|loopvar)'
    r'\s*([^ *]*)\s*\*\*/')


class TemplateEngine:
    def __init__(self, infilename, outfilename, contextfilename):
        self.template = open(infilename).read()
        self.working_dir = Path(infilename).absolute().parent
        self.pos = 0
        self.outfile = open(outfilename, 'w')
```

```
            with open(contextfilename) as contextfile:
                self.context = json.load(contextfile)

        def process(self):
            print("PROCESSING...")


if __name__ == '__main__':
    infilename, outfilename, contextfilename = sys.argv[1:]
    engine = TemplateEngine(infilename, outfilename, contextfilename)
    engine.process()
```

This is all pretty basic, we create a class and initialize it with some variables passed in on the command line.

Notice how we try to make the regular expression a little bit more readable by breaking it across two lines? We use raw strings (the r prefix), so we don't have to double escape all our backslashes. This is common in regular expressions, but it's still a mess. (Regular expressions always are, but they're often worth it.)

The `pos` indicates the current character in the content that we are processing; we'll see a lot more of it in a moment.

Now "all that's left" is to implement that process method. There are a few ways to do this. Let's do it in a fairly explicit way.

The process method has to find each directive that matches the regular expression and do the appropriate work with it. However, it also has to take care of outputting the normal text before, after, and between each directive to the output file, unmodified.

One good feature of the compiled version of regular expressions is that we can tell the `search` method to start searching at a specific position by passing the `pos` keyword argument. If we temporarily define doing the appropriate work with a directive as "ignore the directive and delete it from the output file", our process loop looks quite simple:

```
def process(self):
    match = DIRECTIVE_RE.search(self.template, pos=self.pos)
    while match:
        self.outfile.write(self.template[self.pos:match.start()])
        self.pos = match.end()
        match = DIRECTIVE_RE.search(self.template, pos=self.pos)
    self.outfile.write(self.template[self.pos:])
```

In English, this function finds the first string in the text that matches the regular expression, outputs everything from the current position to the start of that match, and then advances the position to the end of aforesaid match. Once it's out of matches, it outputs everything since the last position.

Of course, ignoring the directive is pretty useless in a templating engine, so let's set up replace that position advancing line with code that delegates to a different method on the class depending on the directive:

```
def process(self):
    match = DIRECTIVE_RE.search(self.template, pos=self.pos)
    while match:
        self.outfile.write(self.template[self.pos:match.start()])
        directive, argument = match.groups()
        method_name = 'process_{}'.format(directive)
        getattr(self, method_name)(match, argument)
        match = DIRECTIVE_RE.search(self.template, pos=self.pos)
    self.outfile.write(self.template[self.pos:])
```

So we grab the directive and the single argument from the regular expression. The directive becomes a method name and we dynamically look up that method name on the `self` object (a little error processing here in case the template writer provides an invalid directive would be better). We pass the match object and argument into that method and assume that method will deal with everything appropriately, including moving the `pos` pointer.

Now that we've got our object-oriented architecture this far, it's actually pretty simple to implement the methods that are delegated to. The `include` and `variable` directives are totally straightforward:

```
def process_include(self, match, argument):
    with (self.working_dir / argument).open() as includefile:
        self.outfile.write(includefile.read())
        self.pos = match.end()

def process_variable(self, match, argument):
    self.outfile.write(self.context.get(argument, ''))
    self.pos = match.end()
```

The first simply looks up the included file and inserts the file contents, while the second looks up the variable name in the context dictionary (which was loaded from `json` in the `__init__` method), defaulting to an empty string if it doesn't exist.

The three methods that deal with looping are a bit more intense, as they have to share state between the three of them. For simplicity (I'm sure you're eager to see the end of this long chapter, we're almost there!), we'll handle this as instance variables on the class itself. As an exercise, you might want to consider better ways to architect this, especially after reading the next three chapters.

```
    def process_loopover(self, match, argument):
        self.loop_index = 0
        self.loop_list = self.context.get(argument, [])
        self.pos = self.loop_pos = match.end()

    def process_loopvar(self, match, argument):
        self.outfile.write(self.loop_list[self.loop_index])
        self.pos = match.end()

    def process_endloop(self, match, argument):
        self.loop_index += 1
        if self.loop_index >= len(self.loop_list):
            self.pos = match.end()
            del self.loop_index
            del self.loop_list
            del self.loop_pos
        else:
            self.pos = self.loop_pos
```

When we encounter the `loopover` directive, we don't have to output anything, but we do have to set the initial state on three variables. The `loop_list` variable is assumed to be a list pulled from the context dictionary. The `loop_index` variable indicates what position in that list should be output in this iteration of the loop, while `loop_pos` is stored so we know where to jump back to when we get to the end of the loop.

The `loopvar` directive outputs the value at the current position in the `loop_list` variable and skips to the end of the directive. Note that it doesn't increment the loop index because the `loopvar` directive could be called multiple times inside a loop.

The `endloop` directive is more complicated. It determines whether there are more elements in the `loop_list`; if there are, it just jumps back to the start of the loop, incrementing the index. Otherwise, it resets all the variables that were being used to process the loop and jumps to the end of the directive so the engine can carry on with the next match.

Note that this particular looping mechanism is very fragile; if a template designer were to try nesting loops or forget an `endloop` call, it would go poorly for them. We would need a lot more error checking and probably want to store more loop state to make this a production platform. But I promised that the end of the chapter was nigh, so let's just head to the exercises, after seeing how our sample template is rendered with its context:

```
<html>
    <body>

<h1>This is the title of the front page</h1>
<a href="link1.html">First Link</a>
<a href="link2.html">Second Link</a>

<p>My name is Dusty.
This is the content of my front page. It goes below the menu.</p>
<table>
<tr><th>Favourite Books</th></tr>

<tr><td>Thief Of Time</td></tr>

<tr><td>The Thief</td></tr>

<tr><td>Snow Crash</td></tr>

<tr><td>Lathe Of Heaven</td></tr>

</table>
    </body>
</html>

Copyright &copy; Today
```

There are some weird newline effects due to the way we planned our template, but it works as expected.

Exercises

We've covered a wide variety of topics in this chapter, from strings to regular expressions, to object serialization, and back again. Now it's time to consider how these ideas can be applied to your own code.

Python strings are very flexible, and Python is an extremely powerful tool for string-based manipulations. If you don't do a lot of string processing in your daily work, try designing a tool that is exclusively intended for manipulating strings. Try to come up with something innovative, but if you're stuck, consider writing a web log analyzer (how many requests per hour? How many people visit more than five pages?) or a template tool that replaces certain variable names with the contents of other files.

Spend a lot of time toying with the string formatting operators until you've got the syntax memorized. Write a bunch of template strings and objects to pass into the format function, and see what kind of output you get. Try the exotic formatting operators, such as percentage or hexadecimal notation. Try out the fill and alignment operators, and see how they behave differently for integers, strings, and floats. Consider writing a class of your own that has a `__format__` method; we didn't discuss this in detail, but explore just how much you can customize formatting.

Make sure you understand the difference between `bytes` and `str` objects. The distinction is very complicated in older versions of Python (there was no `bytes`, and `str` acted like both `bytes` and `str` unless we needed non-ASCII characters in which case there was a separate `unicode` object, which was similar to Python 3's `str` class. It's even more confusing than it sounds!). It's clearer nowadays; `bytes` is for binary data, and `str` is for character data. The only tricky part is knowing how and when to convert between the two. For practice, try writing text data to a file opened for writing `bytes` (you'll have to encode the text yourself), and then reading from the same file.

Do some experimenting with `bytearray`; see how it can act both like a bytes object and a list or container object at the same time. Try writing to a buffer that holds data in the bytes array until it is a certain length before returning it. You can simulate the code that puts data into the buffer by using `time.sleep` calls to ensure data doesn't arrive too quickly.

Study regular expressions online. Study them some more. Especially learn about named groups greedy versus lazy matching, and regex flags, three features that we didn't cover in this chapter. Make conscious decisions about when not to use them. Many people have very strong opinions about regular expressions and either overuse them or refuse to use them at all. Try to convince yourself to use them only when appropriate, and figure out when that is.

If you've ever written an adapter to load small amounts of data from a file or database and convert it to an object, consider using a pickle instead. Pickles are not efficient for storing massive amounts of data, but they can be useful for loading configuration or other simple objects. Try coding it multiple ways: using a pickle, a text file, or a small database. Which do you find easiest to work with?

Try experimenting with pickling data, then modifying the class that holds the data, and loading the pickle into the new class. What works? What doesn't? Is there a way to make drastic changes to a class, such as renaming an attribute or splitting it into two new attributes and still get the data out of an older pickle? (Hint: try placing a private pickle version number on each object and update it each time you change the class; you can then put a migration path in `__setstate__`.)

If you do any web development at all, do some experimenting with the JSON serializer. Personally, I prefer to serialize only standard JSON serializable objects, rather than writing custom encoders or `object_hooks`, but the desired effect really depends on the interaction between the frontend (JavaScript, typically) and backend code.

Create some new directives in the templating engine that take more than one or an arbitrary number of arguments. You might need to modify the regular expression or add new ones. Have a look at the Django project's online documentation, and see if there are any other template tags you'd like to work with. Try mimicking their filter syntax instead of using the variable tag. Revisit this chapter when you've studied iteration and coroutines and see if you can come up with a more compact way of representing the state between related directives, such as the loop.

Summary

We've covered string manipulation, regular expressions, and object serialization in this chapter. Hardcoded strings and program variables can be combined into outputtable strings using the powerful string formatting system. It is important to distinguish between binary and textual data and `bytes` and `str` have specific purposes that must be understood. Both are immutable, but the `bytearray` type can be used when manipulating bytes.

Regular expressions are a complex topic, but we scratched the surface. There are many ways to serialize Python data; pickles and JSON are two of the most popular.

In the next chapter, we'll look at a design pattern that is so fundamental to Python programming that it has been given special syntax support: the iterator pattern.

9
The Iterator Pattern

We've discussed how many of Python's built-ins and idioms that seem, at first blush, to be non-object-oriented are actually providing access to major objects under the hood. In this chapter, we'll discuss how the `for` loop that seems so structured is actually a lightweight wrapper around a set of object-oriented principles. We'll also see a variety of extensions to this syntax that automatically create even more types of object. We will cover:

- What design patterns are
- The iterator protocol—one of the most powerful design patterns
- List, set, and dictionary comprehensions
- Generators and coroutines

Design patterns in brief

When engineers and architects decide to build a bridge, or a tower, or a building, they follow certain principles to ensure structural integrity. There are various possible designs for bridges (suspension or cantilever, for example), but if the engineer doesn't use one of the standard designs, and doesn't have a brilliant new design, it is likely the bridge he/she designs will collapse.

Design patterns are an attempt to bring this same formal definition for correctly designed structures to software engineering. There are many different design patterns to solve different general problems. People who create design patterns first identify a common problem faced by developers in a wide variety of situations. They then suggest what might be considered the ideal solution for that problem, in terms of object-oriented design.

Knowing a design pattern and choosing to use it in our software does not, however, guarantee that we are creating a "correct" solution. In 1907, the Québec Bridge (to this day, the longest cantilever bridge in the world) collapsed before construction was completed, because the engineers who designed it grossly underestimated the weight of the steel used to construct it. Similarly, in software development, we may incorrectly choose or apply a design pattern, and create software that "collapses" under normal operating situations or when stressed beyond its original design limits.

Any one design pattern proposes a set of objects interacting in a specific way to solve a general problem. The job of the programmer is to recognize when they are facing a specific version of that problem, and to adapt the general design in their solution.

In this chapter, we'll be covering the iterator design pattern. This pattern is so powerful and pervasive that the Python developers have provided multiple syntaxes to access the object-oriented principles underlying the pattern. We will be covering other design patterns in the next two chapters. Some of them have language support and some don't, but none of them is as intrinsically a part of the Python coder's daily life as the iterator pattern.

Iterators

In typical design pattern parlance, an iterator is an object with a `next()` method and a `done()` method; the latter returns `True` if there are no items left in the sequence. In a programming language without built-in support for iterators, the iterator would be looped over like this:

```
while not iterator.done():
    item = iterator.next()
    # do something with the item
```

In Python, iteration is a special feature, so the method gets a special name, `__next__`. This method can be accessed using the `next(iterator)` built-in. Rather than a `done` method, the iterator protocol raises `StopIteration` to notify the loop that it has completed. Finally, we have the much more readable `for item in iterator` syntax to actually access items in an iterator instead of messing around with a `while` loop. Let's look at these in more detail.

The iterator protocol

The abstract base class `Iterator`, in the `collections.abc` module, defines the iterator protocol in Python. As mentioned, it must have a `__next__` method that the `for` loop (and other features that support iteration) can call to get a new element from the sequence. In addition, every iterator must also fulfill the `Iterable` interface. Any class that provides an `__iter__` method is iterable; that method must return an `Iterator` instance that will cover all the elements in that class. Since an iterator is already looping over elements, its `__iter__` function traditionally returns itself.

This might sound a bit confusing, so have a look at the following example, but note that this is a very verbose way to solve this problem. It clearly explains iteration and the two protocols in question, but we'll be looking at several more readable ways to get this effect later in this chapter:

```
class CapitalIterable:
    def __init__(self, string):
        self.string = string

    def __iter__(self):
        return CapitalIterator(self.string)


class CapitalIterator:
    def __init__(self, string):
        self.words = [w.capitalize() for w in string.split()]
        self.index = 0

    def __next__(self):
        if self.index == len(self.words):
            raise StopIteration()

        word = self.words[self.index]
        self.index += 1
        return word

    def __iter__(self):
        return self
```

This example defines an `CapitalIterable` class whose job is to loop over each of the words in a string and output them with the first letter capitalized. Most of the work of that iterable is passed to the `CapitalIterator` implementation. The canonical way to interact with this iterator is as follows:

```
>>> iterable = CapitalIterable('the quick brown fox jumps over the lazy
dog')
>>> iterator = iter(iterable)
>>> while True:
...     try:
...         print(next(iterator))
...     except StopIteration:
...         break
...
The
Quick
Brown
Fox
Jumps
Over
The
Lazy
Dog
```

This example first constructs an iterable and retrieves an iterator from it. The distinction may need explanation; the iterable is an object with elements that can be looped over. Normally, these elements can be looped over multiple times, maybe even at the same time or in overlapping code. The iterator, on the other hand, represents a specific location in that iterable; some of the items have been consumed and some have not. Two different iterators might be at different places in the list of words, but any one iterator can mark only one place.

Each time `next()` is called on the iterator, it returns another token from the iterable, in order. Eventually, the iterator will be exhausted (won't have any more elements to return), in which case `Stopiteration` is raised, and we break out of the loop.

Of course, we already know a much simpler syntax for constructing an iterator from an iterable:

```
>>> for i in iterable:
...     print(i)
...
The
Quick
Brown
Fox
Jumps
Over
The
Lazy
Dog
```

As you can see, the `for` statement, in spite of not looking terribly object-oriented, is actually a shortcut to some obviously object-oriented design principles. Keep this in mind as we discuss comprehensions, as they, too, appear to be the polar opposite of an object-oriented tool. Yet, they use the exact same iteration protocol as `for` loops and are just another kind of shortcut.

Comprehensions

Comprehensions are simple, but powerful, syntaxes that allow us to transform or filter an iterable object in as little as one line of code. The resultant object can be a perfectly normal list, set, or dictionary, or it can be a generator expression that can be efficiently consumed in one go.

List comprehensions

List comprehensions are one of the most powerful tools in Python, so people tend to think of them as advanced. They're not. Indeed, I've taken the liberty of littering previous examples with comprehensions and assuming you'd understand them. While it's true that advanced programmers use comprehensions a lot, it's not because they're advanced, it's because they're trivial, and handle some of the most common operations in software development.

Let's have a look at one of those common operations; namely, converting a list of items into a list of related items. Specifically, let's assume we just read a list of strings from a file, and now we want to convert it to a list of integers. We know every item in the list is an integer, and we want to do some activity (say, calculate an average) on those numbers. Here's one simple way to approach it:

```
input_strings = ['1', '5', '28', '131', '3']

output_integers = []
for num in input_strings:
    output_integers.append(int(num))
```

This works fine and it's only three lines of code. If you aren't used to comprehensions, you may not even think it looks ugly! Now, look at the same code using a list comprehension:

```
input_strings = ['1', '5', '28', '131', '3']output_integers =
[int(num) for num in input_strings]
```

We're down to one line and, importantly for performance, we've dropped an `append` method call for each item in the list. Overall, it's pretty easy to tell what's going on, even if you're not used to comprehension syntax.

The square brackets indicate, as always, that we're creating a list. Inside this list is a `for` loop that iterates over each item in the input sequence. The only thing that may be confusing is what's happening between the list's opening brace and the start of the `for` loop. Whatever happens here is applied to *each* of the items in the input list. The item in question is referenced by the `num` variable from the loop. So, it's converting each individual element to an `int` data type.

That's all there is to a basic list comprehension. They are not so advanced after all. Comprehensions are highly optimized C code; list comprehensions are far faster than `for` loops when looping over a huge number of items. If readability alone isn't a convincing reason to use them as much as possible, speed should be.

Converting one list of items into a related list isn't the only thing we can do with a list comprehension. We can also choose to exclude certain values by adding an `if` statement inside the comprehension. Have a look:

```
output_ints = [int(n) for n in input_strings if len(n) < 3]
```

I shortened the name of the variable from `num` to `n` and the result variable to `output_ints` so it would still fit on one line. Other than this, all that's different between this example and the previous one is the `if len(n) < 3` part. This extra code excludes any strings with more than two characters. The `if` statement is applied before the `int` function, so it's testing the length of a string. Since our input strings are all integers at heart, it excludes any number over 99. Now that is all there is to list comprehensions! We use them to map input values to output values, applying a filter along the way to include or exclude any values that meet a specific condition.

Any iterable can be the input to a list comprehension; anything we can wrap in a `for` loop can also be placed inside a comprehension. For example, text files are iterable; each call to `__next__` on the file's iterator will return one line of the file. We could load a tab delimited file where the first line is a header row into a dictionary using the `zip` function:

```
import sys
filename = sys.argv[1]

with open(filename) as file:
    header = file.readline().strip().split('\t')
    contacts = [
            dict(
                zip(header, line.strip().split('\t'))
                ) for line in file
            ]

for contact in contacts:
    print("email: {email} -- {last}, {first}".format(
        **contact))
```

This time, I've added some whitespace to make it somewhat more readable (list comprehensions don't *have* to fit on one line). This example creates a list of dictionaries from the zipped header and split lines for each line in the file.

Er, what? Don't worry if that code or explanation doesn't make sense; it's a bit confusing. One list comprehension is doing a pile of work here, and the code is hard to understand, read, and ultimately, maintain. This example shows that list comprehensions aren't always the best solution; most programmers would agree that a `for` loop would be more readable than this version.

Remember: the tools we are provided with should not be abused! Always pick the right tool for the job, which is always to write maintainable code.

Set and dictionary comprehensions

Comprehensions aren't restricted to lists. We can use a similar syntax with braces to create sets and dictionaries as well. Let's start with sets. One way to create a set is to wrap a list comprehension in the `set()` constructor, which converts it to a set. But why waste memory on an intermediate list that gets discarded, when we can create a set directly?

Here's an example that uses a named tuple to model author/title/genre triads, and then retrieves a set of all the authors that write in a specific genre:

```
from collections import namedtuple

Book = namedtuple("Book", "author title genre")
books = [
        Book("Pratchett", "Nightwatch", "fantasy"),
        Book("Pratchett", "Thief Of Time", "fantasy"),
        Book("Le Guin", "The Dispossessed", "scifi"),
        Book("Le Guin", "A Wizard Of Earthsea", "fantasy"),
        Book("Turner", "The Thief", "fantasy"),
        Book("Phillips", "Preston Diamond", "western"),
        Book("Phillips", "Twice Upon A Time", "scifi"),
        ]

fantasy_authors = {
        b.author for b in books if b.genre == 'fantasy'}
```

The highlighted set comprehension sure is short in comparison to the demo-data setup! If we were to use a list comprehension, of course, Terry Pratchett would have been listed twice.. As it is, the nature of sets removes the duplicates, and we end up with:

```
>>> fantasy_authors
{'Turner', 'Pratchett', 'Le Guin'}
```

We can introduce a colon to create a dictionary comprehension. This converts a sequence into a dictionary using *key: value* pairs. For example, it may be useful to quickly look up the author or genre in a dictionary if we know the title. We can use a dictionary comprehension to map titles to book objects:

```
fantasy_titles = {
        b.title: b for b in books if b.genre == 'fantasy'}
```

Now, we have a dictionary, and can look up books by title using the normal syntax.

In summary, comprehensions are not advanced Python, nor are they "non-object-oriented" tools that should be avoided. They are simply a more concise and optimized syntax for creating a list, set, or dictionary from an existing sequence.

Generator expressions

Sometimes we want to process a new sequence without placing a new list, set, or dictionary into system memory. If we're just looping over items one at a time, and don't actually care about having a final container object created, creating that container is a waste of memory. When processing one item at a time, we only need the current object stored in memory at any one moment. But when we create a container, all the objects have to be stored in that container before we start processing them.

For example, consider a program that processes log files. A very simple log might contain information in this format:

```
Jan 26, 2015 11:25:25    DEBUG        This is a debugging message.
Jan 26, 2015 11:25:36    INFO         This is an information method.
Jan 26, 2015 11:25:46    WARNING      This is a warning. It could be
serious.
Jan 26, 2015 11:25:52    WARNING      Another warning sent.
Jan 26, 2015 11:25:59    INFO         Here's some information.
Jan 26, 2015 11:26:13    DEBUG        Debug messages are only useful
if you want to figure something out.
Jan 26, 2015 11:26:32    INFO         Information is usually harmless,
but helpful.
Jan 26, 2015 11:26:40    WARNING      Warnings should be heeded.
Jan 26, 2015 11:26:54    WARNING      Watch for warnings.
```

Log files for popular web servers, databases, or e-mail servers can contain many gigabytes of data (I recently had to clean nearly 2 terabytes of logs off a misbehaving system). If we want to process each line in the log, we can't use a list comprehension; it would create a list containing every line in the file. This probably wouldn't fit in RAM and could bring the computer to its knees, depending on the operating system.

If we used a `for` loop on the log file, we could process one line at a time before reading the next one into memory. Wouldn't be nice if we could use comprehension syntax to get the same effect?

This is where generator expressions come in. They use the same syntax as comprehensions, but they don't create a final container object. To create a generator expression, wrap the comprehension in `()` instead of `[]` or `{}`.

The following code parses a log file in the previously presented format, and outputs a new log file that contains only the `WARNING` lines:

```
import sys

inname = sys.argv[1]
outname = sys.argv[2]

with open(inname) as infile:
    with open(outname, "w") as outfile:
        warnings = (l for l in infile if 'WARNING' in l)
        for l in warnings:
            outfile.write(l)
```

This program takes the two filenames on the command line, uses a generator expression to filter out the warnings (in this case, it uses the `if` syntax, and leaves the line unmodified), and then outputs the warnings to another file. If we run it on our sample file, the output looks like this:

```
Jan 26, 2015 11:25:46    WARNING     This is a warning. It could be
serious.
Jan 26, 2015 11:25:52    WARNING     Another warning sent.
Jan 26, 2015 11:26:40    WARNING     Warnings should be heeded.
Jan 26, 2015 11:26:54    WARNING     Watch for warnings.
```

Of course, with such a short input file, we could have safely used a list comprehension, but if the file is millions of lines long, the generator expression will have a huge impact on both memory and speed.

Generator expressions are frequently most useful inside function calls. For example, we can call `sum`, `min`, or `max`, on a generator expression instead of a list, since these functions process one object at a time. We're only interested in the result, not any intermediate container.

In general, a generator expression should be used whenever possible. If we don't actually need a list, set, or dictionary, but simply need to filter or convert items in a sequence, a generator expression will be most efficient. If we need to know the length of a list, or sort the result, remove duplicates, or create a dictionary, we'll have to use the comprehension syntax.

Generators

Generator expressions are actually a sort of comprehension too; they compress the more advanced (this time it really is more advanced!) generator syntax into one line. The greater generator syntax looks even less object-oriented than anything we've seen, but we'll discover that once again, it is a simple syntax shortcut to create a kind of object.

Let's take the log file example a little further. If we want to delete the `WARNING` column from our output file (since it's redundant: this file contains only warnings), we have several options, at various levels of readability. We can do it with a generator expression:

```
import sys
inname, outname = sys.argv[1:3]

with open(inname) as infile:
    with open(outname, "w") as outfile:
        warnings = (l.replace('\tWARNING', '')
                for l in infile if 'WARNING' in l)
        for l in warnings:
            outfile.write(l)
```

That's perfectly readable, though I wouldn't want to make the expression much more complicated than that. We could also do it with a normal `for` loop:

```
import sys
inname, outname = sys.argv[1:3]

with open(inname) as infile:
    with open(outname, "w") as outfile:
        for l in infile:
            if 'WARNING' in l:
                outfile.write(l.replace('\tWARNING', ''))
```

That's maintainable, but so many levels of indent in so few lines is kind of ugly. More alarmingly, if we wanted to do something different with the lines, rather than just printing them out, we'd have to duplicate the looping and conditional code, too. Now let's consider a truly object-oriented solution, without any shortcuts:

```
import sys
inname, outname = sys.argv[1:3]

class WarningFilter:
    def __init__(self, insequence):
```

```
            self.insequence = insequence
        def __iter__(self):
            return self
        def __next__(self):
            l = self.insequence.readline()
            while l and 'WARNING' not in l:
                l = self.insequence.readline()
            if not l:
                raise StopIteration
            return l.replace('\tWARNING', '')

    with open(inname) as infile:
        with open(outname, "w") as outfile:
            filter = WarningFilter(infile)
            for l in filter:
                outfile.write(l)
```

No doubt about it: that is so ugly and difficult to read that you may not even be able to tell what's going on. We created an object that takes a file object as input, and provides a `__next__` method like any iterator.

This `__next__` method reads lines from the file, discarding them if they are not `WARNING` lines. When it encounters a `WARNING` line, it returns it. Then the `for` loop will call `__next__` again to process the next `WARNING` line. When we run out of lines, we raise `StopIteration` to tell the loop we're finished iterating. It's pretty ugly compared to the other examples, but it's also powerful; now that we have a class in our hands, we can do whatever we want with it.

With that background behind us, we finally get to see generators in action. This next example does *exactly* the same thing as the previous one: it creates an object with a `__next__` method that raises `StopIteration` when it's out of inputs:

```
    import sys
    inname, outname = sys.argv[1:3]

    def warnings_filter(insequence):
        for l in insequence:
            if 'WARNING' in l:
                yield l.replace('\tWARNING', '')

    with open(inname) as infile:
        with open(outname, "w") as outfile:
```

```
filter = warnings_filter(infile)
for l in filter:
    outfile.write(l)
```

OK, that's pretty readable, maybe... at least it's short. But what on earth is going on here, it makes no sense whatsoever. And what is `yield`, anyway?

In fact, `yield` is the key to generators. When Python sees `yield` in a function, it takes that function and wraps it up in an object not unlike the one in our previous example. Think of the `yield` statement as similar to the `return` statement; it exits the function and returns a line. Unlike `return`, however, when the function is called again (via `next()`), it will start where it left off—on the line after the `yield` statement—instead of at the beginning of the function. In this example, there is no line "after" the `yield` statement, so it jumps to the next iteration of the `for` loop. Since the `yield` statement is inside an `if` statement, it only yields lines that contain `WARNING`.

While it looks like this is just a function looping over the lines, it is actually creating a special type of object, a generator object:

```
>>> print(warnings_filter([]))
<generator object warnings_filter at 0xb728c6bc>
```

I passed an empty list into the function to act as an iterator. All the function does is create and return a generator object. That object has `__iter__` and `__next__` methods on it, just like the one we created in the previous example. Whenever `__next__` is called, the generator runs the function until it finds a `yield` statement. It then returns the value from `yield`, and the next time `__next__` is called, it picks up where it left off.

This use of generators isn't that advanced, but if you don't realize the function is creating an object, it can seem like magic. This example was quite simple, but you can get really powerful effects by making multiple calls to `yield` in a single function; the generator will simply pick up at the most recent `yield` and continue to the next one.

Yield items from another iterable

Often, when we build a generator function, we end up in a situation where we want to yield data from another iterable object, possibly a list comprehension or generator expression we constructed inside the generator, or perhaps some external items that were passed into the function. This has always been possible by looping over the iterable and individually yielding each item. However, in Python version 3.3, the Python developers introduced a new syntax to make this a little more elegant.

Let's adapt the generator example a bit so that instead of accepting a sequence of lines, it accepts a filename. This would normally be frowned upon as it ties the object to a particular paradigm. When possible we should operate on iterators as input; this way the same function could be used regardless of whether the log lines came from a file, memory, or a web-based log aggregator. So the following example is contrived for pedagogical reasons.

This version of the code illustrates that your generator can do some basic setup before yielding information from another iterable (in this case, a generator expression):

```
import sys
inname, outname = sys.argv[1:3]

def warnings_filter(infilename):
    with open(infilename) as infile:
        yield from (
            l.replace('\tWARNING', '')
            for l in infile
            if 'WARNING' in l
        )

filter = warnings_filter(inname)
with open(outname, "w") as outfile:
    for l in filter:
        outfile.write(l)
```

This code combines the `for` loop from the previous example into a generator expression. Notice how I put the three clauses of the generator expression (the transformation, the loop, and the filter) on separate lines to make them more readable. Notice also that this transformation didn't help enough; the previous example with a `for` loop was more readable.

So let's consider an example that is more readable than its alternative. It can be useful to construct a generator that yields data from multiple other generators. The `itertools.chain` function, for example, yields data from iterables in sequence until they have all been exhausted. This can be implemented far too easily using the `yield from` syntax, so let's consider a classic computer science problem: walking a general tree.

A common implementation of the general tree data structure is a computer's filesystem. Let's model a few folders and files in a Unix filesystem so we can use `yield from` to walk them effectively:

```
class File:
    def __init__(self, name):
```

```
        self.name = name

class Folder(File):
    def __init__(self, name):
        super().__init__(name)
        self.children = []

root = Folder('')
etc = Folder('etc')
root.children.append(etc)
etc.children.append(File('passwd'))
etc.children.append(File('groups'))
httpd = Folder('httpd')
etc.children.append(httpd)
httpd.children.append(File('http.conf'))
var = Folder('var')
root.children.append(var)
log = Folder('log')
var.children.append(log)
log.children.append(File('messages'))
log.children.append(File('kernel'))
```

This setup code looks like a lot of work, but in a real filesystem, it would be even more involved. We'd have to read data from the hard drive and structure it into the tree. Once in memory, however, the code that outputs every file in the filesystem is quite elegant:

```
def walk(file):
    if isinstance(file, Folder):
        yield file.name + '/'
        for f in file.children:
            yield from walk(f)
    else:
        yield file.name
```

If this code encounters a directory, it recursively asks `walk()` to generate a list of all files subordinate to each of its children, and then yields all that data plus its own filename. In the simple case that it has encountered a normal file, it just yields that name.

As an aside, solving the preceding problem without using a generator is tricky enough that this problem is a common interview question. If you answer it as shown like this, be prepared for your interviewer to be both impressed and somewhat irritated that you answered it so easily. They will likely demand that you explain exactly what is going on. Of course, armed with the principles you've leaned in this chapter, you won't have any problem.

The `yield from` syntax is a useful shortcut when writing chained generators, but it is more commonly used for a different purpose: piping data through coroutines. We'll see many examples of this in *Chapter 13*, *Concurrency*, but for now, let's discover what a coroutine is.

Coroutines

Coroutines are extremely powerful constructs that are often confused with generators. Many authors inappropriately describe coroutines as "generators with a bit of extra syntax." This is an easy mistake to make, as, way back in Python 2.5, when coroutines were introduced, they were presented as "we added a `send` method to the generator syntax." This is further complicated by the fact that when you create a coroutine in Python, the object returned is a generator. The difference is actually a lot more nuanced and will make more sense after you've seen a few examples.

While coroutines in Python are currently tightly coupled to the generator syntax, they are only superficially related to the iterator protocol we have been discussing. The upcoming (as this is published) Python 3.5 release makes coroutines a truly standalone object and will provide a new syntax to work with them.

The other thing to bear in mind is that coroutines are pretty hard to understand. They are not used all that often in the wild, and you could likely skip this section and happily develop in Python for years without missing or even encountering them. There are a couple libraries that use coroutines extensively (mostly for concurrent or asynchronous programming), but they are normally written such that you can use coroutines without actually understanding how they work! So if you get lost in this section, don't despair.

But you won't get lost, having studied the following examples. Here's one of the simplest possible coroutines; it allows us to keep a running tally that can be increased by arbitrary values:

```
def tally():
    score = 0
```

```
        while True:
            increment = yield score
            score += increment
```

This code looks like black magic that couldn't possibly work, so we'll see it working before going into a line-by-line description. This simple object could be used by a scoring application for a baseball team. Separate tallies could be kept for each team, and their score could be incremented by the number of runs accumulated at the end of every half-inning. Look at this interactive session:

```
>>> white_sox = tally()
>>> blue_jays = tally()
>>> next(white_sox)
0
>>> next(blue_jays)
0
>>> white_sox.send(3)
3
>>> blue_jays.send(2)
2
>>> white_sox.send(2)
5
>>> blue_jays.send(4)
6
```

First we construct two `tally` objects, one for each team. Yes, they look like functions, but as with the generator objects in the previous section, the fact that there is a `yield` statement inside the function tells Python to put a great deal of effort into turning the simple function into an object.

We then call `next()` on each of the coroutine objects. This does the same thing as calling next on any generator, which is to say, it executes each line of code until it encounters a `yield` statement, returns the value at that point, and then *pauses* until the next `next()` call.

So far, then, there's nothing new. But look back at the `yield` statement in our coroutine:

```
    increment = yield score
```

Unlike with generators, this yield function looks like it's supposed to return a value and assign it to a variable. This is, in fact, exactly what's happening. The coroutine is still paused at the `yield` statement and waiting to be activated again by another call to `next()`.

Or rather, as you see in the interactive session, a call to a method called `send()`. The `send()` method does *exactly* the same thing as `next()` except that in addition to advancing the generator to the next `yield` statement. It also allows you to pass in a value from outside the generator. This value is assigned to the left side of the `yield` statement.

The thing that is really confusing for many people is the order in which this happens:

- `yield` occurs and the generator pauses
- `send()` occurs from outside the function and the generator wakes up
- The value sent in is assigned to the left side of the `yield` statement
- The generator continues processing until it encounters another `yield` statement

So, in this particular example, after we construct the coroutine and advance it to the `yield` statement with a call to `next()`, each successive call to `send()` passes a value into the coroutine, which adds this value to its score, goes back to the top of the `while` loop, and keeps processing until it hits the `yield` statement. The `yield` statement returns a value, and this value becomes the return value of the most recent call to `send`. Don't miss that: the `send()` method does not just submit a value to the generator, it also returns the value from the upcoming `yield` statement, just like `next()`. This is how we define the difference between a generator and a coroutine: a generator only produces values, while a coroutine can also consume them.

The behavior and syntax of `next(i)`, `i.__next__()`, and `i.send(value)` are rather unintuitive and frustrating. The first is a normal function, the second is a special method, and the last is a normal method. But all three do the same thing: advance the generator until it yields a value and pause. Further, the `next()` function and associated method can be replicated by calling `i.send(None)`. There is value to having two different method names here, since it helps the reader of our code easily see whether they are interacting with a coroutine or a generator. I just find the fact that in one case it's a function call and in the other it's a normal method somewhat irritating.

Back to log parsing

Of course, the previous example could easily have been coded using a couple integer variables and calling `x += increment` on them. Let's look at a second example where coroutines actually save us some code. This example is a somewhat simplified (for pedagogical reasons) version of a problem I had to solve in my real job. The fact that it logically follows from the earlier discussions about processing a log file is completely serendipitous; those examples were written for the first edition of this book, whereas this problem came up four years later!

The Linux kernel log contains lines that look somewhat, but not quite completely, unlike this:

```
unrelated log messages
sd 0:0:0:0 Attached Disk Drive
unrelated log messages
sd 0:0:0:0 (SERIAL=ZZ12345)
unrelated log messages
sd 0:0:0:0 [sda] Options
unrelated log messages
XFS ERROR [sda]
unrelated log messages
sd 2:0:0:1 Attached Disk Drive
unrelated log messages
sd 2:0:0:1 (SERIAL=ZZ67890)
unrelated log messages
sd 2:0:0:1 [sdb] Options
unrelated log messages
sd 3:0:1:8 Attached Disk Drive
unrelated log messages
sd 3:0:1:8 (SERIAL=WW11111)
unrelated log messages
sd 3:0:1:8 [sdc] Options
unrelated log messages
XFS ERROR [sdc]
unrelated log messages
```

There are a whole bunch of interspersed kernel log messages, some of which pertain to hard disks. The hard disk messages might be interspersed with other messages, but they occur in a predictable format and order, in which a specific drive with a known serial number is associated with a bus identifier (such as `0:0:0:0`), and a block device identifier (such as `sda`) is associated with that bus. Finally, if the drive has a corrupt filesystem, it might fail with an XFS error.

Now, given the preceding log file, the problem we need to solve is how to obtain the serial number of any drives that have XFS errors on them. This serial number might later be used by a data center technician to identify and replace the drive.

We know we can identify the individual lines using regular expressions, but we'll have to change the regular expressions as we loop through the lines, since we'll be looking for different things depending on what we found previously. The other difficult bit is that if we find an error string, the information about which bus contains that string, and what serial number is attached to the drive on that bus has already been processed. This can easily be solved by iterating through the lines of the file in reverse order.

Before you look at this example, be warned—the amount of code required for a coroutine-based solution is scarily small:

```
import re

def match_regex(filename, regex):
    with open(filename) as file:
        lines = file.readlines()
    for line in reversed(lines):
        match = re.match(regex, line)
        if match:
            regex = yield match.groups()[0]

def get_serials(filename):
    ERROR_RE = 'XFS ERROR (\[sd[a-z]\])'
    matcher = match_regex(filename, ERROR_RE)
    device = next(matcher)
    while True:
        bus = matcher.send(
            '(sd \S+) {}.*'.format(re.escape(device)))
        serial = matcher.send('{} \(SERIAL=([^)]*)\)'.format(bus))
        yield serial
        device = matcher.send(ERROR_RE)

for serial_number in get_serials('EXAMPLE_LOG.log'):
    print(serial_number)
```

This code neatly divides the job into two separate tasks. The first task is to loop over all the lines and spit out any lines that match a given regular expression. The second task is to interact with the first task and give it guidance as to what regular expression it is supposed to be searching for at any given time.

Look at the `match_regex` coroutine first. Remember, it doesn't execute any code when it is constructed; rather, it just creates a coroutine object. Once constructed, someone outside the coroutine will eventually call `next()` to start the code running, at which point it stores the state of two variables, `filename` and `regex`. It then reads all the lines in the file and iterates over them in reverse. Each line is compared to the regular expression that was passed in until it finds a match. When the match is found, the coroutine yields the first group from the regular expression and waits.

At some point in the future, other code will send in a new regular expression to search for. Note that the coroutine never cares what regular expression it is trying to match; it's just looping over lines and comparing them to a regular expression. It's somebody else's responsibility to decide what regular expression to supply.

In this case, that somebody else is the `get_serials` generator. It doesn't care about the lines in the file, in fact it isn't even aware of them. The first thing it does is create a `matcher` object from the `match_regex` coroutine constructor, giving it a default regular expression to search for. It advances the coroutine to its first `yield` and stores the value it returns. It then goes into a loop that instructs the matcher object to search for a bus ID based on the stored device ID, and then a serial number based on that bus ID.

It idly yields that serial number to the outside `for` loop before instructing the matcher to find another device ID and repeat the cycle.

Basically, the coroutine's (`match_regex`, as it uses the `regex = yield` syntax) job is to search for the next important line in the file, while the generator's (`get_serial`, which uses the `yield` syntax without assignment) job is to decide which line is important. The generator has information about this particular problem, such as what order lines will appear in the file. The coroutine, on the other hand, could be plugged into any problem that required searching a file for given regular expressions.

Closing coroutines and throwing exceptions

Normal generators signal their exit from inside by raising `StopIteration`. If we chain multiple generators together (for example, by iterating over one generator from inside another), the `StopIteration` exception will be propagated outward. Eventually, it will hit a `for` loop that will see the exception and know that it's time to exit the loop.

Coroutines don't normally follow the iteration mechanism; rather than pulling data through one until an exception is encountered, data is usually pushed into it (using `send`). The entity doing the pushing is normally the one in charge of telling the coroutine when it's finished; it does this by calling the `close()` method on the coroutine in question.

When called, the `close()` method will raise a `GeneratorExit` exception at the point the coroutine was waiting for a value to be sent in. It is normally good policy for coroutines to wrap their `yield` statements in a `try...finally` block so that any cleanup tasks (such as closing associated files or sockets) can be performed.

If we need to raise an exception inside a coroutine, we can use the `throw()` method in a similar way. It accepts an exception type with optional `value` and `traceback` arguments. The latter is useful when we encounter an exception in one coroutine and want to cause an exception to occur in an adjacent coroutine while maintaining the traceback.

Both of these features are vital if you're building robust coroutine-based libraries, but we are unlikely to encounter them in our day-to-day coding lives.

The relationship between coroutines, generators, and functions

We've seen coroutines in action, so now let's go back to that discussion of how they are related to generators. In Python, as is so often the case, the distinction is quite blurry. In fact, all coroutines are generator objects, and authors often use the two terms interchangeably. Sometimes, they describe coroutines as a subset of generators (only generators that return values from yield are considered coroutines). This is technically true in Python, as we've seen in the previous sections.

However, in the greater sphere of theoretical computer science, coroutines are considered the more general principles, and generators are a specific type of coroutine. Further, normal functions are yet another distinct subset of coroutines.

A coroutine is a routine that can have data passed in at one or more points and get it out at one or more points. In Python, the point where data is passed in and out is the `yield` statement.

A function, or subroutine, is the simplest type of coroutine. You can pass data in at one point, and get data out at one other point when the function returns. While a function can have multiple `return` statements, only one of them can be called for any given invocation of the function.

Finally, a generator is a type of coroutine that can have data passed in at one point, but can pass data out at multiple points. In Python, the data would be passed out at a `yield` statement, but you can't pass data back in. If you called `send`, the data would be silently discarded.

So in theory, generators are types of coroutines, functions are types of coroutines, and there are coroutines that are neither functions nor generators. That's simple enough, eh? So why does it feel more complicated in Python?

In Python, generators and coroutines are both constructed using a syntax that looks like we are constructing a function. But the resulting object is not a function at all; it's a totally different kind of object. Functions are, of course, also objects. But they have a different interface; functions are callable and return values, generators have data pulled out using `next()`, and coroutines have data pushed in using `send`.

Case study

One of the fields in which Python is the most popular these days is data science. Let's implement a basic machine learning algorithm! Machine learning is a huge topic, but the general idea is to make predictions or classifications about future data by using knowledge gained from past data. Uses of such algorithms abound, and data scientists are finding new ways to apply machine learning every day. Some important machine learning applications include computer vision (such as image classification or facial recognition), product recommendation, identifying spam, and speech recognition. We'll look at a simpler problem: given an RGB color definition, what name would humans identify that color as?

There are more than 16 million colors in the standard RGB color space, and humans have come up with names for only a fraction of them. While there are thousands of names (some quite ridiculous; just go to any car dealership or makeup store), let's build a classifier that attempts to divide the RGB space into the basic colors:

- Red
- Purple
- Blue
- Green
- Yellow
- Orange
- Grey
- White
- Pink

The first thing we need is a dataset to train our algorithm on. In a production system, you might scrape a *list of colors* website or survey thousands of people. Instead, I created a simple application that renders a random color and asks the user to select one of the preceding nine options to classify it. This application is included with the example code for this chapter in the `kivy_color_classifier` directory, but we won't be going into the details of this code as its only purpose here is to generate sample data.

Kivy has an incredibly well-engineered object-oriented API that you may want to explore on your own time. If you would like to develop graphical programs that run on many systems, from your laptop to your cell phone, you might want to check out my book, *Creating Apps In Kivy*, *O'Reilly*.

For the purposes of this case study, the important thing about that application is the output, which is a **comma-separated value** (**CSV**) file that contains four values per row: the red, green, and blue values (represented as a floating-point number between zero and one), and one of the preceding nine names that the user assigned to that color. The dataset looks something like this:

```
0.30928279150905513,0.7536768153744394,0.3244011790604804,Green
0.4991001855115986,0.6394567277907686,0.6340502030888825,Grey
0.21132621004927998,0.3307376167520666,0.704037576789711,Blue
0.7260420945787928,0.4025279573860123,0.49781705131696363,Pink
0.706469868610228,0.28530423638868196,0.7880240251003464,Purple
0.692243900051664,0.7053550777777416,0.1845069151913028,Yellow
0.3628979381122397,0.11079495501215897,0.26924540840045075,Purple
0.611273677646518,0.48798521783547677,0.5346130557761224,Purple
.
.
.
0.4014121109376566,0.42176706818252674,0.9601866228083298,Blue
0.17750449496124632,0.8008214961070862,0.5073944321437429,Green
```

I made 200 datapoints (a very few of them untrue) before I got bored and decided it was time to start machine learning on this dataset. These datapoints are shipped with the examples for this chapter if you would like to use my data (nobody's ever told me I'm color-blind, so it should be somewhat reasonable).

We'll be implementing one of the simpler machine-learning algorithms, referred to as k-nearest neighbor. This algorithm relies on some kind of "distance" calculation between points in the dataset (in our case, we can use a three-dimensional version of the Pythagorean theorem). Given a new datapoint, it finds a certain number (referred to as k, as in k-nearest neighbors) of datapoints that are closest to it when measured by that distance calculation. Then it combines those datapoints in some way (an average might work for linear calculations; for our classification problem, we'll use the mode), and returns the result.

We won't go into too much detail about what the algorithm does; rather, we'll focus on some of the ways we can apply the iterator pattern or iterator protocol to this problem.

Let's now write a program that performs the following steps in order:

1. Load the sample data from the file and construct a model from it.
2. Generate 100 random colors.
3. Classify each color and output it to a file in the same format as the input.

Once we have this second CSV file, another Kivy program can load the file and render each color, asking a human user to confirm or deny the accuracy of the prediction, thus informing us of how accurate our algorithm and initial data set are.

The first step is a fairly simple generator that loads CSV data and converts it into a format that is amenable to our needs:

```
import csv

dataset_filename = 'colors.csv'

def load_colors(filename):
    with open(filename) as dataset_file:
        lines = csv.reader(dataset_file)
        for line in lines:
            yield tuple(float(y) for y in line[0:3]), line[3]
```

We haven't seen the `csv.reader` function before. It returns an iterator over the lines in the file. Each value returned by the iterator is a list of strings. In our case, we could have just split on commas and been fine, but `csv.reader` also takes care of managing quotation marks and various other nuances of the comma-separated value format.

We then loop over these lines and convert them to a tuple of color and name, where the color is a tuple of three floating value integers. This tuple is constructed using a generator expression. There might be more readable ways to construct this tuple; do you think the code brevity and the speed of a generator expression is worth the obfuscation? Instead of returning a list of color tuples, it yields them one at a time, thus constructing a generator object.

Now, we need a hundred random colors. There are so many ways this can be done:

- A list comprehension with a nested generator expression: `[tuple(random() for r in range(3)) for r in range(100)]`
- A basic generator function
- A class that implements the `__iter__` and `__next__` protocols

- Push the data through a pipeline of coroutines
- Even just a basic `for` loop

The generator version seems to be most readable, so let's add that function to our program:

```
from random import random


def generate_colors(count=100):
    for i in range(count):
        yield (random(), random(), random())
```

Notice how we parameterize the number of colors to generate. We can now reuse this function for other color-generating tasks in the future.

Now, before we do the classification step, we need a function to calculate the "distance" between two colors. Since it's possible to think of colors as being three dimensional (red, green, and blue could map to *x*, *y*, and *z* axes, for example), let's use a little basic math:

```
import math

def color_distance(color1, color2):
    channels = zip(color1, color2)
    sum_distance_squared = 0
    for c1, c2 in channels:
        sum_distance_squared += (c1 - c2) ** 2
    return math.sqrt(sum_distance_squared)
```

This is a pretty basic-looking function; it doesn't look like it's even using the iterator protocol. There's no `yield` function, no comprehensions. However, there is a `for` loop, and that call to the `zip` function is doing some real iteration as well (remember that `zip` yields tuples containing one element from each input iterator).

Note, however, that this function is going to be called a lot of times inside our k-nearest neighbors algorithm. If our code ran too slow and we were able to identify this function as a bottleneck, we might want to replace it with a less readable, but more optimized, generator expression:

```
def color_distance(color1, color2):
    return math.sqrt(sum((x[0] - x[1]) ** 2 for x in zip(
    color1, color2)))
```

However, I strongly recommend not making such optimizations until you have proven that the readable version is too slow.

Now that we have some plumbing in place, let's do the actual k-nearest neighbor implementation. This seems like a good place to use a coroutine. Here it is with some test code to ensure it's yielding sensible values:

```
def nearest_neighbors(model_colors, num_neighbors):
    model = list(model_colors)
    target = yield
    while True:
        distances = sorted(
            ((color_distance(c[0], target), c) for c in model),
        )
        target = yield [
            d[1] for d in distances[0:num_neighbors]
        ]


model_colors = load_colors(dataset_filename)
target_colors = generate_colors(3)
get_neighbors = nearest_neighbors(model_colors, 5)
next(get_neighbors)

for color in target_colors:
    distances = get_neighbors.send(color)
    print(color)
    for d in distances:
        print(color_distance(color, d[0]), d[1])
```

The coroutine accepts two arguments, the list of colors to be used as a model, and the number of neighbors to query. It converts the model to a list because it's going to be iterated over multiple times. In the body of the coroutine, it accepts a tuple of RGB color values using the `yield` syntax. Then it combines a call to `sorted` with an odd generator expression. See if you can figure out what that generator expression is doing.

It returns a tuple of `(distance, color_data)` for each color in the model. Remember, the model itself contains tuples of `(color, name)`, where `color` is a tuple of three RGB values. Therefore, the generator is returning an iterator over a weird data structure that looks like this:

```
(distance, (r, g, b), color_name)
```

The `sorted` call then sorts the results by their first element, which is distance. This is a complicated piece of code and isn't object-oriented at all. You may want to break it down into a normal `for` loop to ensure you understand what the generator expression is doing. It might also be a good exercise to imagine how this code would look if you were to pass a key argument into the sorted function instead of constructing a tuple.

The `yield` statement is a bit less complicated; it pulls the second value from each of the first k `(distance, color_data)` tuples. In more concrete terms, it yields the `((r, g, b), color_name)` tuple for the k values with the lowest distance. Or, if you prefer more abstract terms, it yields the target's k-nearest neighbors in the given model.

The remaining code is just boilerplate to test this method; it constructs the model and a color generator, primes the coroutine, and prints the results in a `for` loop.

The two remaining tasks are to choose a color based on the nearest neighbors, and to output the results to a CSV file. Let's make two more coroutines to take care of these tasks. Let's do the output first because it can be tested independently:

```
def write_results(filename="output.csv"):
    with open(filename, "w") as file:
        writer = csv.writer(file)
        while True:
            color, name = yield
            writer.writerow(list(color) + [name])

results = write_results()
next(results)
for i in range(3):
    print(i)
    results.send(((i, i, i), i * 10))
```

This coroutine maintains an open file as state and writes lines of code to it as they are sent in using `send()`. The test code ensures the coroutine is working correctly, so now we can connect the two coroutines with a third one.

The second coroutine uses a bit of an odd trick:

```
from collections import Counter
def name_colors(get_neighbors):
    color = yield
    while True:
        near = get_neighbors.send(color)
        name_guess = Counter(
            n[1] for n in near).most_common(1)[0][0]
        color = yield name_guess
```

This coroutine accepts, *as its argument*, an existing coroutine. In this case, it's an instance of `nearest_neighbors`. This code basically proxies all the values sent into it through that `nearest_neighbors` instance. Then it does some processing on the result to get the most common color out of the values that were returned. In this case, it would probably make just as much sense to adapt the original coroutine to return a name, since it isn't being used for anything else. However, there are many cases where it is useful to pass coroutines around; this is how we do it.

Now all we have to do is connect these various coroutines and pipelines together, and kick off the process with a single function call:

```
def process_colors(dataset_filename="colors.csv"):
    model_colors = load_colors(dataset_filename)
    get_neighbors = nearest_neighbors(model_colors, 5)
    get_color_name = name_colors(get_neighbors)
    output = write_results()
    next(output)
    next(get_neighbors)
    next(get_color_name)

    for color in generate_colors():
        name = get_color_name.send(color)
        output.send((color, name))

process_colors()
```

So, this function, unlike almost every other function we've defined, is a perfectly normal function without any `yield` statements. It doesn't get turned into a coroutine or generator object. It does, however, construct a generator and three coroutines. Notice how the `get_neighbors` coroutine is passed into the constructor for `name_colors`? Pay attention to how all three coroutines are advanced to their first `yield` statements by calls to `next`.

Once all the pipes are created, we use a `for` loop to send each of the generated colors into the `get_color_name` coroutine, and then we pipe each of the values yielded by that coroutine to the output coroutine, which writes it to a file.

And that's it! I created a second Kivy app that loads the resulting CSV file and presents the colors to the user. The user can select either *Yes* or *No* depending on whether they think the choice made by the machine-learning algorithm matches the choice they would have made. This is not scientifically accurate (it's ripe for observation bias), but it's good enough for playing around. Using my eyes, it succeeded about 84 percent of the time, which is better than my grade 12 average. Not bad for our first ever machine learning experience, eh?

You might be wondering, "what does this have to do with object-oriented programming? There isn't even one class in this code!". In some ways, you'd be right; neither coroutines nor generators are commonly considered object-oriented. However, the functions that create them return objects; in fact, you could think of those functions as constructors. The constructed object has appropriate `send()` and `__next__()` methods. Basically, the coroutine/generator syntax is a syntax shortcut for a particular kind of object that would be quite verbose to create without it.

This case study has been an exercise in bottom-up design. We created various low-level objects that did specific tasks and hooked them all together at the end. I find this to be a common practice when developing with coroutines. The alternative, top-down design sometimes results in more monolithic pieces of code instead of unique individual pieces. In general, we want to find a happy medium between methods that are too large and methods that are too small and it's hard to see how they fit together. This is true, of course, regardless of whether the iterator protocol is being used as we did here.

Exercises

If you don't use comprehensions in your daily coding very often, the first thing you should do is search through some existing code and find some `for` loops. See if any of them can be trivially converted to a generator expression or a list, set, or dictionary comprehension.

Test the claim that list comprehensions are faster than `for` loops. This can be done with the built-in `timeit` module. Use the help documentation for the `timeit.timeit` function to find out how to use it. Basically, write two functions that do the same thing, one using a list comprehension, and one using a `for` loop. Pass each function into `timeit.timeit`, and compare the results. If you're feeling adventurous, compare generators and generator expressions as well. Testing code using `timeit` can become addictive, so bear in mind that code does not need to be hyperfast unless it's being executed an immense number of times, such as on a huge input list or file.

Play around with generator functions. Start with basic iterators that require multiple values (mathematical sequences are canonical examples; the Fibonacci sequence is overused if you can't think of anything better). Try some more advanced generators that do things like take multiple input lists and somehow yield values that merge them. Generators can also be used on files; can you write a simple generator that shows those lines that are identical in two files?

Coroutines abuse the iterator protocol but don't actually fulfill the iterator pattern. Can you build a non-coroutine version of the code that gets a serial number from a log file? Take an object-oriented approach so that you can store an additional state on a class. You'll learn a lot about coroutines if you can create an object that is a drop-in replacement for the existing coroutine.

See if you can abstract the coroutines used in the case study so that the k-nearest-neighbor algorithm can be used on a variety of datasets. You'll likely want to construct a coroutine that accepts other coroutines or functions that do the distance and recombination calculations as parameters, and then calls into those functions to find the actual nearest neighbors.

Summary

In this chapter, we learned that design patterns are useful abstractions that provide "best practice" solutions for common programming problems. We covered our first design pattern, the iterator, as well as numerous ways that Python uses and abuses this pattern for its own nefarious purposes. The original iterator pattern is extremely object-oriented, but it is also rather ugly and verbose to code around. However, Python's built-in syntax abstracts the ugliness away, leaving us with a clean interface to these object-oriented constructs.

Comprehensions and generator expressions can combine container construction with iteration in a single line. Generator objects can be constructed using the `yield` syntax. Coroutines look like generators on the outside but serve a much different purpose.

We'll cover several more design patterns in the next two chapters.

10
Python Design Patterns I

In the last chapter, we were briefly introduced to design patterns, and covered the iterator pattern, a pattern so useful and common that it has been abstracted into the core of the programming language itself. In this chapter, we'll be reviewing other common patterns, and how they are implemented in Python. As with iteration, Python often provides an alternative syntax to make working with such problems simpler. We will cover both the "traditional" design, and the Python version for these patterns. In summary, we'll see:

- Numerous specific patterns
- A canonical implementation of each pattern in Python
- Python syntax to replace certain patterns

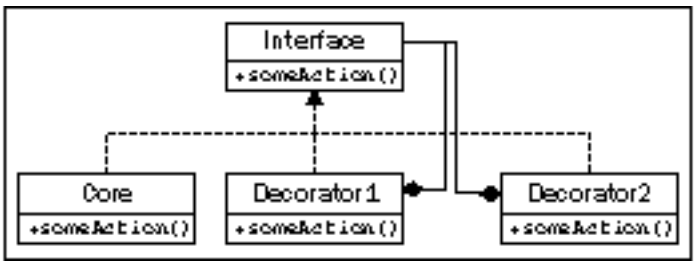
The decorator pattern

The decorator pattern allows us to "wrap" an object that provides core functionality with other objects that alter this functionality. Any object that uses the decorated object will interact with it in exactly the same way as if it were undecorated (that is, the interface of the decorated object is identical to that of the core object).

There are two primary uses of the decorator pattern:

- Enhancing the response of a component as it sends data to a second component
- Supporting multiple optional behaviors

The second option is often a suitable alternative to multiple inheritance. We can construct a core object, and then create a decorator around that core. Since the decorator object has the same interface as the core object, we can even wrap the new object in other decorators. Here's how it looks in UML:



Here, **Core** and all the decorators implement a specific **Interface**. The decorators maintain a reference to another instance of that **Interface** via composition. When called, the decorator does some added processing before or after calling its wrapped interface. The wrapped object may be another decorator, or the core functionality. While multiple decorators may wrap each other, the object in the "center" of all those decorators provides the core functionality.

A decorator example

Let's look at an example from network programming. We'll be using a TCP socket. The `socket.send()` method takes a string of input bytes and outputs them to the receiving socket at the other end. There are plenty of libraries that accept sockets and access this function to send data on the stream. Let's create such an object; it will be an interactive shell that waits for a connection from a client and then prompts the user for a string response:

```
import socket

def respond(client):
    response = input("Enter a value: ")
    client.send(bytes(response, 'utf8'))
    client.close()

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(('localhost',2401))
server.listen(1)
try:
    while True:
```

```
            client, addr = server.accept()
            respond(client)
    finally:
        server.close()
```

The `respond` function accepts a socket parameter and prompts for data to be sent as a reply, then sends it. To use it, we construct a server socket and tell it to listen on port `2401` (I picked the port randomly) on the local computer. When a client connects, it calls the `respond` function, which requests data interactively and responds appropriately. The important thing to notice is that the `respond` function only cares about two methods of the socket interface: `send` and `close`. To test this, we can write a very simple client that connects to the same port and outputs the response before exiting:

```
import socket

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(('localhost', 2401))
print("Received: {0}".format(client.recv(1024)))
client.close()
```

To use these programs:

1. Start the server in one terminal.
2. Open a second terminal window and run the client.
3. At the **Enter a value:** prompt in the server window, type a value and press enter.
4. The client will receive what you typed, print it to the console, and exit. Run the client a second time; the server will prompt for a second value.

Now, looking again at our server code, we see two sections. The `respond` function sends data into a socket object. The remaining script is responsible for creating that socket object. We'll create a pair of decorators that customize the socket behavior without having to extend or modify the socket itself.

Let's start with a "logging" decorator. This object outputs any data being sent to the server's console before it sends it to the client:

```
class LogSocket:
    def __init__(self, socket):
        self.socket = socket

    def send(self, data):
```

```
            print("Sending {0} to {1}".format(
                data, self.socket.getpeername()[0]))
            self.socket.send(data)

        def close(self):
            self.socket.close()
```

This class decorates a socket object and presents the `send` and `close` interface to client sockets. A better decorator would also implement (and possibly customize) all of the remaining socket methods. It should properly implement all of the arguments to `send`, (which actually accepts an optional flags argument) as well, but let's keep our example simple! Whenever `send` is called on this object, it logs the output to the screen before sending data to the client using the original socket.

We only have to change one line in our original code to use this decorator. Instead of calling `respond` with the socket, we call it with a decorated socket:

```
respond(LogSocket(client))
```

While that's quite simple, we have to ask ourselves why we didn't just extend the socket class and override the `send` method. We could call `super().send` to do the actual sending, after we logged it. There is nothing wrong with this design either.

When faced with a choice between decorators and inheritance, we should only use decorators if we need to modify the object dynamically, according to some condition. For example, we may only want to enable the logging decorator if the server is currently in debugging mode. Decorators also beat multiple inheritance when we have more than one optional behavior. As an example, we can write a second decorator that compresses data using `gzip` compression whenever `send` is called:

```
import gzip
from io import BytesIO

class GzipSocket:
    def __init__(self, socket):
        self.socket = socket

    def send(self, data):
        buf = BytesIO()
        zipfile = gzip.GzipFile(fileobj=buf, mode="w")
        zipfile.write(data)
        zipfile.close()
        self.socket.send(buf.getvalue())

    def close(self):
        self.socket.close()
```

The `send` method in this version compresses the incoming data before sending it on to the client.

Now that we have these two decorators, we can write code that dynamically switches between them when responding. This example is not complete, but it illustrates the logic we might follow to mix and match decorators:

```
client, addr = server.accept()
if log_send:
    client = LoggingSocket(client)
if client.getpeername()[0] in compress_hosts:
    client = GzipSocket(client)
respond(client)
```

This code checks a hypothetical configuration variable named `log_send`. If it's enabled, it wraps the socket in a `LoggingSocket` decorator. Similarly, it checks whether the client that has connected is in a list of addresses known to accept compressed content. If so, it wraps the client in a `GzipSocket` decorator. Notice that none, either, or both of the decorators may be enabled, depending on the configuration and connecting client. Try writing this using multiple inheritance and see how confused you get!

Decorators in Python

The decorator pattern is useful in Python, but there are other options. For example, we may be able to use monkey-patching, which we discussed in *Chapter 7*, *Python Object-oriented Shortcuts*, to get a similar effect. Single inheritance, where the "optional" calculations are done in one large method can be an option, and multiple inheritance should not be written off just because it's not suitable for the specific example seen previously!

In Python, it is very common to use this pattern on functions. As we saw in a previous chapter, functions are objects too. In fact, function decoration is so common that Python provides a special syntax to make it easy to apply such decorators to functions.

For example, we can look at the logging example in a more general way. Instead of logging, only send calls on sockets, we may find it helpful to log all calls to certain functions or methods. The following example implements a decorator that does just this:

```
import time

def log_calls(func):
    def wrapper(*args, **kwargs):
        now = time.time()
```

```
            print("Calling {0} with {1} and {2}".format(
                func.__name__, args, kwargs))
            return_value = func(*args, **kwargs)
            print("Executed {0} in {1}ms".format(
                func.__name__, time.time() - now))
            return return_value
        return wrapper

def test1(a,b,c):
    print("\ttest1 called")

def test2(a,b):
    print("\ttest2 called")

def test3(a,b):
    print("\ttest3 called")
    time.sleep(1)

test1 = log_calls(test1)
test2 = log_calls(test2)
test3 = log_calls(test3)

test1(1,2,3)
test2(4,b=5)
test3(6,7)
```

This decorator function is very similar to the example we explored earlier; in those cases, the decorator took a socket-like object and created a socket-like object. This time, our decorator takes a function object and returns a new function object. This code is comprised of three separate tasks:

- A function, `log_calls`, that accepts another function
- This function defines (internally) a new function, named `wrapper`, that does some extra work before calling the original function
- This new function is returned

Three sample functions demonstrate the decorator in use. The third one includes a sleep call to demonstrate the timing test. We pass each function into the decorator, which returns a new function. We assign this new function to the original variable name, effectively replacing the original function with a decorated one.

This syntax allows us to build up decorated function objects dynamically, just as we did with the socket example; if we don't replace the name, we can even keep decorated and non-decorated versions for different situations.

Often these decorators are general modifications that are applied permanently to different functions. In this situation, Python supports a special syntax to apply the decorator at the time the function is defined. We've already seen this syntax when we discussed the `property` decorator; now, let's understand how it works.

Instead of applying the decorator function after the method definition, we can use the `@decorator` syntax to do it all at once:

```
@log_calls
def test1(a,b,c):
    print("\ttest1 called")
```

The primary benefit of this syntax is that we can easily see that the function has been decorated at the time it is defined. If the decorator is applied later, someone reading the code may miss that the function has been altered at all. Answering a question like, "Why is my program logging function calls to the console?" can become much more difficult! However, the syntax can only be applied to functions we define, since we don't have access to the source code of other modules. If we need to decorate functions that are part of somebody else's third-party library, we have to use the earlier syntax.

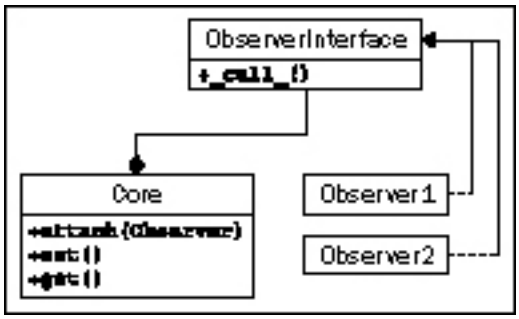
There is more to the decorator syntax than we've seen here. We don't have room to cover the advanced topics here, so check the Python reference manual or other tutorials for more information. Decorators can be created as callable objects, not just functions that return functions. Classes can also be decorated; in that case, the decorator returns a new class instead of a new function. Finally, decorators can take arguments to customize them on a per-function basis.

The observer pattern

The observer pattern is useful for state monitoring and event handling situations. This pattern allows a given object to be monitored by an unknown and dynamic group of "observer" objects.

Whenever a value on the core object changes, it lets all the observer objects know that a change has occurred, by calling an `update()` method. Each observer may be responsible for different tasks whenever the core object changes; the core object doesn't know or care what those tasks are, and the observers don't typically know or care what other observers are doing.

Here, it is in UML:



An observer example

The observer pattern might be useful in a redundant backup system. We can write a core object that maintains certain values, and then have one or more observers create serialized copies of that object. These copies might be stored in a database, on a remote host, or in a local file, for example. Let's implement the core object using properties:

```
class Inventory:
    def __init__(self):
        self.observers = []
        self._product = None
        self._quantity = 0

    def attach(self, observer):
        self.observers.append(observer)

    @property
    def product(self):
        return self._product
    @product.setter
    def product(self, value):
        self._product = value
        self._update_observers()

    @property
    def quantity(self):
        return self._quantity
    @quantity.setter
    def quantity(self, value):
        self._quantity = value
```

```
            self._update_observers()

        def _update_observers(self):
            for observer in self.observers:
                observer()
```

This object has two properties that, when set, call the `_update_observers` method on itself. All this method does is loop over the available observers and let each one know that something has changed. In this case, we call the observer object directly; the object will have to implement `__call__` to process the update. This would not be possible in many object-oriented programming languages, but it's a useful shortcut in Python that can help make our code more readable.

Now let's implement a simple observer object; this one will just print out some state to the console:

```
class ConsoleObserver:
    def __init__(self, inventory):
        self.inventory = inventory

    def __call__(self):
        print(self.inventory.product)
        print(self.inventory.quantity)
```

There's nothing terribly exciting here; the observed object is set up in the initializer, and when the observer is called, we do "something." We can test the observer in an interactive console:

```
>>> i = Inventory()
>>> c = ConsoleObserver(i)
>>> i.attach(c)
>>> i.product = "Widget"
Widget
0
>>> i.quantity = 5
Widget
5
```

After attaching the observer to the inventory object, whenever we change one of the two observed properties, the observer is called and its action is invoked. We can even add two different observer instances:

```
>>> i = Inventory()
>>> c1 = ConsoleObserver(i)
```

```
>>> c2 = ConsoleObserver(i)
>>> i.attach(c1)
>>> i.attach(c2)
>>> i.product = "Gadget"
Gadget
0
Gadget
0
```

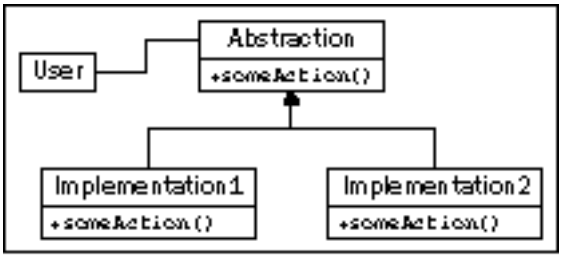
This time when we change the product, there are two sets of output, one for each observer. The key idea here is that we can easily add totally different types of observers that back up the data in a file, database, or Internet application at the same time.

The observer pattern detaches the code being observed from the code doing the observing. If we were not using this pattern, we would have had to put code in each of the properties to handle the different cases that might come up; logging to the console, updating a database or file, and so on. The code for each of these tasks would all be mixed in with the observed object. Maintaining it would be a nightmare, and adding new monitoring functionality at a later date would be painful.

The strategy pattern

The strategy pattern is a common demonstration of abstraction in object-oriented programming. The pattern implements different solutions to a single problem, each in a different object. The client code can then choose the most appropriate implementation dynamically at runtime.

Typically, different algorithms have different trade-offs; one might be faster than another, but uses a lot more memory, while a third algorithm may be most suitable when multiple CPUs are present or a distributed system is provided. Here is the strategy pattern in UML:



The **User** code connecting to the strategy pattern simply needs to know that it is dealing with the **Abstraction** interface. The actual implementation chosen performs the same task, but in different ways; either way, the interface is identical.

A strategy example

The canonical example of the strategy pattern is sort routines; over the years, numerous algorithms have been invented for sorting a collection of objects; quick sort, merge sort, and heap sort are all fast sort algorithms with different features, each useful in its own right, depending on the size and type of inputs, how out of order they are, and the requirements of the system.

If we have client code that needs to sort a collection, we could pass it to an object with a `sort()` method. This object may be a `QuickSorter` or `MergeSorter` object, but the result will be the same in either case: a sorted list. The strategy used to do the sorting is abstracted from the calling code, making it modular and replaceable.

Of course, in Python, we typically just call the `sorted` function or `list.sort` method and trust that it will do the sorting in a near-optimal fashion. So, we really need to look at a better example.

Let's consider a desktop wallpaper manager. When an image is displayed on a desktop background, it can be adjusted to the screen size in different ways. For example, assuming the image is smaller than the screen, it can be tiled across the screen, centered on it, or scaled to fit. There are other, more complicated, strategies that can be used as well, such as scaling to the maximum height or width, combining it with a solid, semi-transparent, or gradient background color, or other manipulations. While we may want to add these strategies later, let's start with the basic ones.

Our strategy objects takes two inputs; the image to be displayed, and a tuple of the width and height of the screen. They each return a new image the size of the screen, with the image manipulated to fit according to the given strategy. You'll need to install the `pillow` module with `pip3 install pillow` for this example to work:

```
from PIL import Image


class TiledStrategy:
    def make_background(self, img_file, desktop_size):
        in_img = Image.open(img_file)
        out_img = Image.new('RGB', desktop_size)
        num_tiles = [
```

```
            o // i + 1 for o, i in
            zip(out_img.size, in_img.size)
        ]
        for x in range(num_tiles[0]):
            for y in range(num_tiles[1]):
                out_img.paste(
                    in_img,
                    (
                        in_img.size[0] * x,
                        in_img.size[1] * y,
                        in_img.size[0] * (x+1),
                        in_img.size[1] * (y+1)
                    )
                )
        return out_img


class CenteredStrategy:
    def make_background(self, img_file, desktop_size):
        in_img = Image.open(img_file)
        out_img = Image.new('RGB', desktop_size)
        left = (out_img.size[0] - in_img.size[0]) // 2
        top = (out_img.size[1] - in_img.size[1]) // 2
        out_img.paste(
            in_img,
            (
                left,
                top,
                left+in_img.size[0],
                top + in_img.size[1]
            )
        )
        return out_img


class ScaledStrategy:
    def make_background(self, img_file, desktop_size):
        in_img = Image.open(img_file)
        out_img = in_img.resize(desktop_size)
        return out_img
```

Here we have three strategies, each using `PIL` to perform their task. Individual strategies have a `make_background` method that accepts the same set of parameters. Once selected, the appropriate strategy can be called to create a correctly sized version of the desktop image. `TiledStrategy` loops over the number of input images that would fit in the width and height of the image and copies it into each location, repeatedly. `CenteredStrategy` figures out how much space needs to be left on the four edges of the image to center it. `ScaledStrategy` forces the image to the output size (ignoring aspect ratio).

Consider how switching between these options would be implemented without the strategy pattern. We'd need to put all the code inside one great big method and use an awkward `if` statement to select the expected one. Every time we wanted to add a new strategy, we'd have to make the method even more ungainly.

Strategy in Python

The preceding canonical implementation of the strategy pattern, while very common in most object-oriented libraries, is rarely seen in Python programming.

These classes each represent objects that do nothing but provide a single function. We could just as easily call that function `__call__` and make the object callable directly. Since there is no other data associated with the object, we need do no more than create a set of top-level functions and pass them around as our strategies instead.

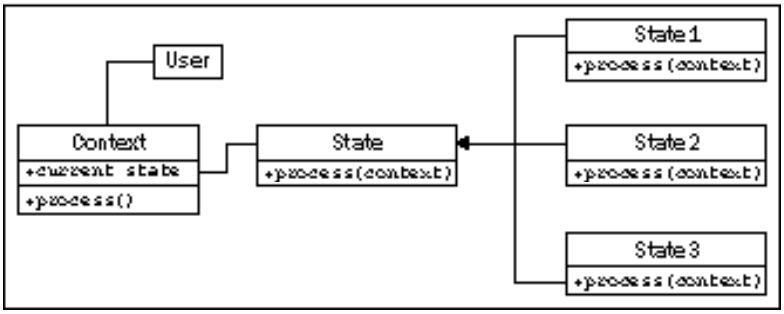
Opponents of design pattern philosophy will therefore say, "because Python has first-class functions, the strategy pattern is unnecessary". In truth, Python's first-class functions allow us to implement the strategy pattern in a more straightforward way. Knowing the pattern exists can still help us choose a correct design for our program, but implement it using a more readable syntax. The strategy pattern, or a top-level function implementation of it, should be used when we need to allow client code or the end user to select from multiple implementations of the same interface.

The state pattern

The state pattern is structurally similar to the strategy pattern, but its intent and purpose are very different. The goal of the state pattern is to represent state-transition systems: systems where it is obvious that an object can be in a specific state, and that certain activities may drive it to a different state.

To make this work, we need a manager, or context class that provides an interface for switching states. Internally, this class contains a pointer to the current state; each state knows what other states it is allowed to be in and will transition to those states depending on actions invoked upon it.

So we have two types of classes, the context class and multiple state classes. The context class maintains the current state, and forwards actions to the state classes. The state classes are typically hidden from any other objects that are calling the context; it acts like a black box that happens to perform state management internally. Here's how it looks in UML:



A state example

To illustrate the state pattern, let's build an XML parsing tool. The context class will be the parser itself. It will take a string as input and place the tool in an initial parsing state. The various parsing states will eat characters, looking for a specific value, and when that value is found, change to a different state. The goal is to create a tree of node objects for each tag and its contents. To keep things manageable, we'll parse only a subset of XML - tags and tag names. We won't be able to handle attributes on tags. It will parse text content of tags, but won't attempt to parse "mixed" content, which has tags inside of text. Here is an example "simplified XML" file that we'll be able to parse:

```
<book>
    <author>Dusty Phillips</author>
    <publisher>Packt Publishing</publisher>
    <title>Python 3 Object Oriented Programming</title>
    <content>
        <chapter>
            <number>1</number>
            <title>Object Oriented Design</title>
        </chapter>
        <chapter>
            <number>2</number>
            <title>Objects In Python</title>
        </chapter>
    </content>
</book>
```

Before we look at the states and the parser, let's consider the output of this program. We know we want a tree of `Node` objects, but what does a `Node` look like? Well, clearly it'll need to know the name of the tag it is parsing, and since it's a tree, it should probably maintain a pointer to the parent node and a list of the node's children in order. Some nodes have a text value, but not all of them. Let's look at this `Node` class first:

```
class Node:
    def __init__(self, tag_name, parent=None):
        self.parent = parent
        self.tag_name = tag_name
        self.children = []
        self.text=""

    def __str__(self):
        if self.text:
            return self.tag_name + ": " + self.text
        else:
            return self.tag_name
```

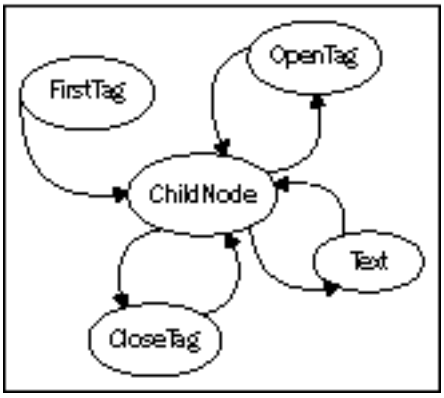
This class sets default attribute values upon initialization. The `__str__` method is supplied to help visualize the tree structure when we're finished.

Now, looking at the example document, we need to consider what states our parser can be in. Clearly it's going to start in a state where no nodes have yet been processed. We'll need a state for processing opening tags and closing tags. And when we're inside a tag with text contents, we'll have to process that as a separate state, too.

Switching states can be tricky; how do we know if the next node is an opening tag, a closing tag, or a text node? We could put a little logic in each state to work this out, but it actually makes more sense to create a new state whose sole purpose is figuring out which state we'll be switching to next. If we call this transition state **ChildNode**, we end up with the following states:

- **FirstTag**
- **ChildNode**
- **OpenTag**
- **CloseTag**
- **Text**

The **FirstTag** state will switch to **ChildNode**, which is responsible for deciding which of the other three states to switch to; when those states are finished, they'll switch back to **ChildNode**. The following state-transition diagram shows the available state changes:



The states are responsible for taking "what's left of the string", processing as much of it as they know what to do with, and then telling the parser to take care of the rest of it. Let's construct the `Parser` class first:

```
class Parser:
    def __init__(self, parse_string):
        self.parse_string = parse_string
        self.root = None
        self.current_node = None

        self.state = FirstTag()

    def process(self, remaining_string):
        remaining = self.state.process(remaining_string, self)
        if remaining:
            self.process(remaining)

    def start(self):
        self.process(self.parse_string)
```

The initializer sets up a few variables on the class that the individual states will access. The `parse_string` instance variable is the text that we are trying to parse. The `root` node is the "top" node in the XML structure. The `current_node` instance variable is the one that we are currently adding children to.

The important feature of this parser is the `process` method, which accepts the remaining string, and passes it off to the current state. The parser (the `self` argument) is also passed into the state's process method so that the state can manipulate it. The state is expected to return the remainder of the unparsed string when it is finished processing. The parser then recursively calls the `process` method on this remaining string to construct the rest of the tree.

Now, let's have a look at the `FirstTag` state:

```
class FirstTag:
    def process(self, remaining_string, parser):
        i_start_tag = remaining_string.find('<')
        i_end_tag = remaining_string.find('>')
        tag_name = remaining_string[i_start_tag+1:i_end_tag]
        root = Node(tag_name)
        parser.root = parser.current_node = root
        parser.state = ChildNode()
        return remaining_string[i_end_tag+1:]
```

This state finds the index (the `i_` stands for index) of the opening and closing angle brackets on the first tag. You may think this state is unnecessary, since XML requires that there be no text before an opening tag. However, there may be whitespace that needs to be consumed; this is why we search for the opening angle bracket instead of assuming it is the first character in the document. Note that this code is assuming a valid input file. A proper implementation would be rigorously testing for invalid input, and would attempt to recover or display an extremely descriptive error message.

The method extracts the name of the tag and assigns it to the root node of the parser. It also assigns it to `current_node`, since that's the one we'll be adding children to next.

Then comes the important part: the method changes the current state on the parser object to a `ChildNode` state. It then returns the remainder of the string (after the opening tag) to allow it to be processed.

The `ChildNode` state, which seems quite complicated, turns out to require nothing but a simple conditional:

```
class ChildNode:
    def process(self, remaining_string, parser):
        stripped = remaining_string.strip()
        if stripped.startswith("</"):
            parser.state = CloseTag()
```

```
            elif stripped.startswith("<"):
                parser.state = OpenTag()
            else:
                parser.state = TextNode()
            return stripped
```

The `strip()` call removes whitespace from the string. Then the parser determines if the next item is an opening or closing tag, or a string of text. Depending on which possibility occurs, it sets the parser to a particular state, and then tells it to parse the remainder of the string.

The `OpenTag` state is similar to the `FirstTag` state, except that it adds the newly created node to the previous `current_node` object's `children` and sets it as the new `current_node`. It places the processor back in the `ChildNode` state before continuing:

```
class OpenTag:
    def process(self, remaining_string, parser):
        i_start_tag = remaining_string.find('<')
        i_end_tag = remaining_string.find('>')
        tag_name = remaining_string[i_start_tag+1:i_end_tag]
        node = Node(tag_name, parser.current_node)
        parser.current_node.children.append(node)
        parser.current_node = node
        parser.state = ChildNode()
        return remaining_string[i_end_tag+1:]
```

The `CloseTag` state basically does the opposite; it sets the parser's `current_node` back to the parent node so any further children in the outside tag can be added to it:

```
class CloseTag:
    def process(self, remaining_string, parser):
        i_start_tag = remaining_string.find('<')
        i_end_tag = remaining_string.find('>')
        assert remaining_string[i_start_tag+1] == "/"
        tag_name = remaining_string[i_start_tag+2:i_end_tag]
        assert tag_name == parser.current_node.tag_name
        parser.current_node = parser.current_node.parent
        parser.state = ChildNode()
        return remaining_string[i_end_tag+1:].strip()
```

The two `assert` statements help ensure that the parse strings are consistent. The `if` statement at the end of the method ensures that the processor terminates when it is finished. If the parent of a node is `None`, it means that we are working on the root node.

Finally, the `TextNode` state very simply extracts the text before the next close tag and sets it as a value on the current node:

```
class TextNode:
    def process(self, remaining_string, parser):
        i_start_tag = remaining_string.find('<')
        text = remaining_string[:i_start_tag]
        parser.current_node.text = text
        parser.state = ChildNode()
        return remaining_string[i_start_tag:]
```

Now we just have to set up the initial state on the parser object we created. The initial state is a `FirstTag` object, so just add the following to the `__init__` method:

```
        self.state = FirstTag()
```

To test the class, let's add a main script that opens an file from the command line, parses it, and prints the nodes:

```
if __name__ == "__main__":
    import sys
    with open(sys.argv[1]) as file:
        contents = file.read()
        p = Parser(contents)
        p.start()

        nodes = [p.root]
        while nodes:
            node = nodes.pop(0)
            print(node)
            nodes = node.children + nodes
```

This code opens the file, loads the contents, and parses the result. Then it prints each node and its children in order. The `__str__` method we originally added on the node class takes care of formatting the nodes for printing. If we run the script on the earlier example, it outputs the tree as follows:

```
book
author: Dusty Phillips
publisher: Packt Publishing
title: Python 3 Object Oriented Programming
content
chapter
number: 1
```

```
title: Object Oriented Design
chapter
number: 2
title: Objects In Python
```

Comparing this to the original simplified XML document tells us the parser is working.

State versus strategy

The state pattern looks very similar to the strategy pattern; indeed, the UML diagrams for the two are identical. The implementation, too, is identical; we could even have written our states as first-class functions instead of wrapping them in objects, as was suggested for strategy.

While the two patterns have identical structures, they solve completely different problems. The strategy pattern is used to choose an algorithm at runtime; generally, only one of those algorithms is going to be chosen for a particular use case. The state pattern, on the other hand is designed to allow switching between different states dynamically, as some process evolves. In code, the primary difference is that the strategy pattern is not typically aware of other strategy objects. In the state pattern, either the state or the context needs to know which other states that it can switch to.

State transition as coroutines

The state pattern is the canonical object-oriented solution to state-transition problems. However, the syntax for this pattern is rather verbose. You can get a similar effect by constructing your objects as coroutines. Remember the regular expression log file parser we built in *Chapter 9*, *The Iterator Pattern*? That was a state-transition problem in disguise. The main difference between that implementation and one that defines all the objects (or functions) used in the state pattern is that the coroutine solution allows us to encode more of the boilerplate in language constructs. There are two implementations, but neither one is inherently better than the other, but you may find that coroutines are more readable, for a given definition of "readable" (you have to understand the syntax of coroutines, first!).

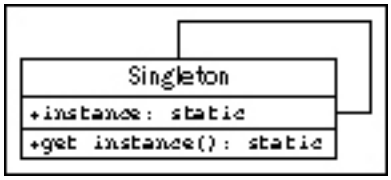
The singleton pattern

The singleton pattern is one of the most controversial patterns; many have accused it of being an "anti-pattern", a pattern that should be avoided, not promoted. In Python, if someone is using the singleton pattern, they're almost certainly doing something wrong, probably because they're coming from a more restrictive programming language.

So why discuss it at all? Singleton is one of the most famous of all design patterns. It is useful in overly object-oriented languages, and is a vital part of traditional object-oriented programming. More relevantly, the idea behind singleton is useful, even if we implement that idea in a totally different way in Python.

The basic idea behind the singleton pattern is to allow exactly one instance of a certain object to exist. Typically, this object is a sort of manager class like those we discussed in *Chapter 5*, *When to Use Object-oriented Programming*. Such objects often need to be referenced by a wide variety of other objects, and passing references to the manager object around to the methods and constructors that need them can make code hard to read.

Instead, when a singleton is used, the separate objects request the single instance of the manager object from the class, so a reference to it need not to be passed around. The UML diagram doesn't fully describe it, but here it is for completeness:



In most programming environments, singletons are enforced by making the constructor private (so no one can create additional instances of it), and then providing a static method to retrieve the single instance. This method creates a new instance the first time it is called, and then returns that same instance each time it is called again.

Singleton implementation

Python doesn't have private constructors, but for this purpose, it has something even better. We can use the `__new__` class method to ensure that only one instance is ever created:

```
class OneOnly:
    _singleton = None
    def __new__(cls, *args, **kwargs):
        if not cls._singleton:
            cls._singleton = super(OneOnly, cls
                ).__new__(cls, *args, **kwargs)
        return cls._singleton
```

When `__new__` is called, it normally constructs a new instance of that class. When we override it, we first check if our singleton instance has been created; if not, we create it using a `super` call. Thus, whenever we call the constructor on `OneOnly`, we always get the exact same instance:

```
>>> o1 = OneOnly()
>>> o2 = OneOnly()
>>> o1 == o2
True
>>> o1
<__main__.OneOnly object at 0xb71c008c>
>>> o2
<__main__.OneOnly object at 0xb71c008c>
```

The two objects are equal and located at the same address; thus, they are the same object. This particular implementation isn't very transparent, since it's not obvious that a singleton object has been created. Whenever we call a constructor, we expect a new instance of that object; in this case, that contract is violated. Perhaps, good docstrings on the class could alleviate this problem if we really think we need a singleton.

But we don't need it. Python coders frown on forcing the users of their code into a specific mindset. We may think only one instance of a class will ever be required, but other programmers may have different ideas. Singletons can interfere with distributed computing, parallel programming, and automated testing, for example. In all those cases, it can be very useful to have multiple or alternative instances of a specific object, even though a "normal' operation may never require one.

Module variables can mimic singletons

Normally, in Python, the singleton pattern can be sufficiently mimicked using module-level variables. It's not as "safe" as a singleton in that people could reassign those variables at any time, but as with the private variables we discussed in *Chapter 2*, *Objects in Python*, this is acceptable in Python. If someone has a valid reason to change those variables, why should we stop them? It also doesn't stop people from instantiating multiple instances of the object, but again, if they have a valid reason to do so, why interfere?

Ideally, we should give them a mechanism to get access to the "default singleton" value, while also allowing them to create other instances if they need them. While technically not a singleton at all, it provides the most Pythonic mechanism for singleton-like behavior.

To use module-level variables instead of a singleton, we instantiate an instance of the class after we've defined it. We can improve our state pattern to use singletons. Instead of creating a new object every time we change states, we can create a module-level variable that is always accessible:

```
class FirstTag:
    def process(self, remaining_string, parser):
        i_start_tag = remaining_string.find('<')
        i_end_tag = remaining_string.find('>')
        tag_name = remaining_string[i_start_tag+1:i_end_tag]
        root = Node(tag_name)
        parser.root = parser.current_node = root
        parser.state = child_node
        return remaining_string[i_end_tag+1:]

class ChildNode:
    def process(self, remaining_string, parser):
        stripped = remaining_string.strip()
        if stripped.startswith("</"):
            parser.state = close_tag
        elif stripped.startswith("<"):
            parser.state = open_tag
        else:
            parser.state = text_node
        return stripped

class OpenTag:
    def process(self, remaining_string, parser):
        i_start_tag = remaining_string.find('<')
        i_end_tag = remaining_string.find('>')
        tag_name = remaining_string[i_start_tag+1:i_end_tag]
        node = Node(tag_name, parser.current_node)
        parser.current_node.children.append(node)
        parser.current_node = node
        parser.state = child_node
        return remaining_string[i_end_tag+1:]
class TextNode:
    def process(self, remaining_string, parser):
        i_start_tag = remaining_string.find('<')
        text = remaining_string[:i_start_tag]
        parser.current_node.text = text
        parser.state = child_node
```

```
            return remaining_string[i_start_tag:]

    class CloseTag:
        def process(self, remaining_string, parser):
            i_start_tag = remaining_string.find('<')
            i_end_tag = remaining_string.find('>')
            assert remaining_string[i_start_tag+1] == "/"
            tag_name = remaining_string[i_start_tag+2:i_end_tag]
            assert tag_name == parser.current_node.tag_name
            parser.current_node = parser.current_node.parent
            parser.state = child_node
            return remaining_string[i_end_tag+1:].strip()

    first_tag = FirstTag()
    child_node = ChildNode()
    text_node = TextNode()
    open_tag = OpenTag()
    close_tag = CloseTag()
```

All we've done is create instances of the various state classes that can be reused. Notice how we can access these module variables inside the classes, even before the variables have been defined? This is because the code inside the classes is not executed until the method is called, and by this point, the entire module will have been defined.

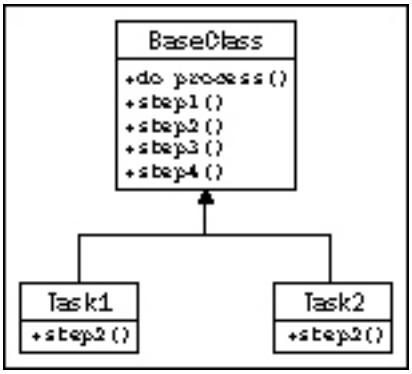
The difference in this example is that instead of wasting memory creating a bunch of new instances that must be garbage collected, we are reusing a single state object for each state. Even if multiple parsers are running at once, only these state classes need to be used.

When we originally created the state-based parser, you may have wondered why we didn't pass the parser object to `__init__` on each individual state, instead of passing it into the process method as we did. The state could then have been referenced as `self.parser`. This is a perfectly valid implementation of the state pattern, but it would not have allowed leveraging the singleton pattern. If the state objects maintain a reference to the parser, then they cannot be used simultaneously to reference other parsers.

Remember, these are two different patterns with different purposes; the fact that singleton's purpose may be useful for implementing the state pattern does not mean the two patterns are related.

The template pattern

The template pattern is useful for removing duplicate code; it's an implementation to support the **Don't Repeat Yourself** principle we discussed in *Chapter 5*, *When to Use Object-oriented Programming*. It is designed for situations where we have several different tasks to accomplish that have some, but not all, steps in common. The common steps are implemented in a base class, and the distinct steps are overridden in subclasses to provide custom behavior. In some ways, it's like a generalized strategy pattern, except similar sections of the algorithms are shared using a base class. Here it is in the UML format:



A template example

Let's create a car sales reporter as an example. We can store records of sales in an SQLite database table. SQLite is a simple file-based database engine that allows us to store records using SQL syntax. Python 3 includes SQLite in its standard library, so there are no extra modules required.

We have two common tasks we need to perform:

- Select all sales of new vehicles and output them to the screen in a comma-delimited format
- Output a comma-delimited list of all salespeople with their gross sales and save it to a file that can be imported to a spreadsheet

These seem like quite different tasks, but they have some common features. In both cases, we need to perform the following steps:

1. Connect to the database.
2. Construct a query for new vehicles or gross sales.

3. Issue the query.
4. Format the results into a comma-delimited string.
5. Output the data to a file or e-mail.

The query construction and output steps are different for the two tasks, but the remaining steps are identical. We can use the template pattern to put the common steps in a base class, and the varying steps in two subclasses.

Before we start, let's create a database and put some sample data in it, using a few lines of SQL:

```
import sqlite3

conn = sqlite3.connect("sales.db")

conn.execute("CREATE TABLE Sales (salesperson text, "
        "amt currency, year integer, model text, new boolean)")
conn.execute("INSERT INTO Sales values"
        " ('Tim', 16000, 2010, 'Honda Fit', 'true')")
conn.execute("INSERT INTO Sales values"
        " ('Tim', 9000, 2006, 'Ford Focus', 'false')")
conn.execute("INSERT INTO Sales values"
        " ('Gayle', 8000, 2004, 'Dodge Neon', 'false')")
conn.execute("INSERT INTO Sales values"
        " ('Gayle', 28000, 2009, 'Ford Mustang', 'true')")
conn.execute("INSERT INTO Sales values"
        " ('Gayle', 50000, 2010, 'Lincoln Navigator', 'true')")
conn.execute("INSERT INTO Sales values"
        " ('Don', 20000, 2008, 'Toyota Prius', 'false')")
conn.commit()
conn.close()
```

Hopefully you can see what's going on here even if you don't know SQL; we've created a table to hold the data, and used six insert statements to add sales records. The data is stored in a file named `sales.db`. Now we have a sample we can work with in developing our template pattern.

Since we've already outlined the steps that the template has to perform, we can start by defining the base class that contains the steps. Each step gets its own method (to make it easy to selectively override any one step), and we have one more managerial method that calls the steps in turn. Without any method content, here's how it might look:

```
class QueryTemplate:
    def connect(self):
        pass
```

```
    def construct_query(self):
        pass
    def do_query(self):
        pass
    def format_results(self):
        pass
    def output_results(self):
        pass

    def process_format(self):
        self.connect()
        self.construct_query()
        self.do_query()
        self.format_results()
        self.output_results()
```

The `process_format` method is the primary method to be called by an outside client. It ensures each step is executed in order, but it does not care if that step is implemented in this class or in a subclass. For our examples, we know that three methods are going to be identical between our two classes:

```
import sqlite3

class QueryTemplate:
    def connect(self):
        self.conn = sqlite3.connect("sales.db")

    def construct_query(self):
        raise NotImplementedError()

    def do_query(self):
        results = self.conn.execute(self.query)
        self.results = results.fetchall()

    def format_results(self):
        output = []
        for row in self.results:
            row =[str(i) for i in row]
            output.append(", ".join(row))
        self.formatted_results = "\n".join(output)

    def output_results(self):
        raise NotImplementedError()
```

To help with implementing subclasses, the two methods that are not specified raise `NotImplementedError`. This is a common way to specify abstract interfaces in Python when abstract base classes seem too heavyweight. The methods could have empty implementations (with `pass`), or could be fully unspecified. Raising `NotImplementedError`, however, helps the programmer understand that the class is meant to be subclassed and these methods overridden; empty methods or methods that do not exist are harder to identify as needing to be implemented and to debug if we forget to implement them.

Now we have a template class that takes care of the boring details, but is flexible enough to allow the execution and formatting of a wide variety of queries. The best part is, if we ever want to change our database engine from SQLite to another database engine (such as py-postgresql), we only have to do it here, in this template class, and we don't have to touch the two (or two hundred) subclasses we might have written.

Let's have a look at the concrete classes now:

```
import datetime
class NewVehiclesQuery(QueryTemplate):
    def construct_query(self):
        self.query = "select * from Sales where new='true'"

    def output_results(self):
        print(self.formatted_results)

class UserGrossQuery(QueryTemplate):
    def construct_query(self):
        self.query = ("select salesperson, sum(amt) " +
        " from Sales group by salesperson")

    def output_results(self):
        filename = "gross_sales_{0}".format(
                datetime.date.today().strftime("%Y%m%d")
                )
        with open(filename, 'w') as outfile:
            outfile.write(self.formatted_results)
```

These two classes are actually pretty short, considering what they're doing: connecting to a database, executing a query, formatting the results, and outputting them. The superclass takes care of the repetitive work, but lets us easily specify those steps that vary between tasks. Further, we can also easily change steps that are provided in the base class. For example, if we wanted to output something other than a comma-delimited string (for example: an HTML report to be uploaded to a website), we can still override `format_results`.

Exercises

While writing this chapter, I discovered that it can be very difficult, and extremely educational, to come up with good examples where specific design patterns should be used. Instead of going over current or old projects to see where you can apply these patterns, as I've suggested in previous chapters, think about the patterns and different situations where they might come up. Try to think outside your own experiences. If your current projects are in the banking business, consider how you'd apply these design patterns in a retail or point-of-sale application. If you normally write web applications, think about using design patterns while writing a compiler.

Look at the decorator pattern and come up with some good examples of when to apply it. Focus on the pattern itself, not the Python syntax we discussed; it's a bit more general than the actual pattern. The special syntax for decorators is, however, something you may want to look for places to apply in existing projects too.

What are some good areas to use the observer pattern? Why? Think about not only how you'd apply the pattern, but how you would implement the same task without using observer? What do you gain, or lose, by choosing to use it?

Consider the difference between the strategy and state patterns. Implementation-wise, they look very similar, yet they have different purposes. Can you think of cases where the patterns could be interchanged? Would it be reasonable to redesign a state-based system to use strategy instead, or vice versa? How different would the design actually be?

The template pattern is such an obvious application of inheritance to reduce duplicate code that you may have used it before, without knowing its name. Try to think of at least half a dozen different scenarios where it would be useful. If you can do this, you'll be finding places for it in your daily coding all the time.

Summary

This chapter discussed several common design patterns in detail, with examples, UML diagrams, and a discussion of the differences between Python and statically typed object-oriented languages. The decorator pattern is often implemented using Python's more generic decorator syntax. The observer pattern is a useful way to decouple events from actions taken on those events. The strategy pattern allows different algorithms to be chosen to accomplish the same task. The state pattern looks similar, but is used instead to represent systems can move between different states using well-defined actions. The singleton pattern, popular in some statically typed languages, is almost always an anti-pattern in Python.

In the next chapter, we'll wrap up our discussion of design patterns.

11
Python Design Patterns II

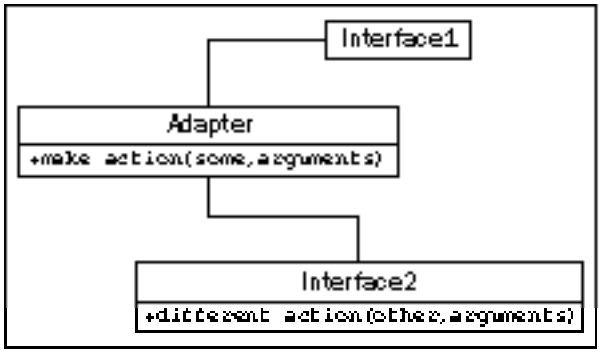
In this chapter we will be introduced to several more design patterns. Once again, we'll cover the canonical examples as well as any common alternative implementations in Python. We'll be discussing:

- The adapter pattern
- The facade pattern
- Lazy initialization and the flyweight pattern
- The command pattern
- The abstract factory pattern
- The composition pattern

The adapter pattern

Unlike most of the patterns we reviewed in *Chapter 8*, *Strings and Serialization*, the adapter pattern is designed to interact with existing code. We would not design a brand new set of objects that implement the adapter pattern. Adapters are used to allow two pre-existing objects to work together, even if their interfaces are not compatible. Like the display adapters that allow VGA projectors to be plugged into HDMI ports, an adapter object sits between two different interfaces, translating between them on the fly. The adapter object's sole purpose is to perform this translation job. Adapting may entail a variety of tasks, such as converting arguments to a different format, rearranging the order of arguments, calling a differently named method, or supplying default arguments.

In structure, the adapter pattern is similar to a simplified decorator pattern. Decorators typically provide the same interface that they replace, whereas adapters map between two different interfaces. Here it is in UML form:



Here, **Interface1** is expecting to call a method called **make_action(some, arguments)**. We already have this perfect **Interface2** class that does everything we want (and to avoid duplication, we don't want to rewrite it!), but it provides a method called **different_action(other, arguments)** instead. The **Adapter** class implements the **make_action** interface and maps the arguments to the existing interface.

The advantage here is that the code that maps from one interface to another is all in one place. The alternative would be really ugly; we'd have to perform the translation in multiple places whenever we need to access this code.

For example, imagine we have the following preexisting class, which takes a string date in the format "YYYY-MM-DD" and calculates a person's age on that day:

```
class AgeCalculator:
    def __init__(self, birthday):
        self.year, self.month, self.day = (
                int(x) for x in birthday.split('-'))

    def calculate_age(self, date):
        year, month, day = (
                int(x) for x in date.split('-'))
        age = year - self.year
        if (month,day) < (self.month,self.day):
            age -= 1
        return age
```

This is a pretty simple class that does what it's supposed to do. But we have to wonder what the programmer was thinking, using a specifically formatted string instead of using Python's incredibly useful built-in `datetime` library. As conscientious programmers who reuse code whenever possible, most of the programs we write will interact with `datetime` objects, not strings.

We have several options to address this scenario; we could rewrite the class to accept `datetime` objects, which would probably be more accurate anyway. But if this class had been provided by a third party and we don't know or can't change its internal structure, we need to try something else. We could use the class as it is, and whenever we want to calculate the age on a `datetime.date` object, we could call `datetime.date.strftime('%Y-%m-%d')` to convert it to the proper format. But that conversion would be happening in a lot of places, and worse, if we mistyped the `%m` as `%M`, it would give us the current minute instead of the entered month! Imagine if you wrote that in a dozen different places only to have to go back and change it when you realized your mistake. It's not maintainable code, and it breaks the DRY principle.

Instead, we can write an adapter that allows a normal date to be plugged into a normal `AgeCalculator` class:

```
import datetime
class DateAgeAdapter:
    def _str_date(self, date):
        return date.strftime("%Y-%m-%d")

    def __init__(self, birthday):
        birthday = self._str_date(birthday)
        self.calculator = AgeCalculator(birthday)

    def get_age(self, date):
        date = self._str_date(date)
        return self.calculator.calculate_age(date)
```

This adapter converts `datetime.date` and `datetime.time` (they have the same interface to `strftime`) into a string that our original `AgeCalculator` can use. Now we can use the original code with our new interface. I changed the method signature to `get_age` to demonstrate that the calling interface may also be looking for a different method name, not just a different type of argument.

Creating a class as an adapter is the usual way to implement this pattern, but, as usual, there are other ways to do it in Python. Inheritance and multiple inheritance can be used to add functionality to a class. For example, we could add an adapter on the `date` class so that it works with the original `AgeCalculator` class:

```
import datetime
class AgeableDate(datetime.date):
    def split(self, char):
        return self.year, self.month, self.day
```

It's code like this that makes one wonder if Python should even be legal. We have added a `split` method to our subclass that takes a single argument (which we ignore) and returns a tuple of year, month, and day. This works flawlessly with the original `AgeCalculator` class because the code calls `strip` on a specially formatted string, and `strip`, in that case, returns a tuple of year, month, and day. The `AgeCalculator` code only cares if `strip` exists and returns acceptable values; it doesn't care if we really passed in a string. It really works:

```
>>> bd = AgeableDate(1975, 6, 14)
>>> today = AgeableDate.today()
>>> today
AgeableDate(2015, 8, 4)
>>> a = AgeCalculator(bd)
>>> a.calculate_age(today)
40
```

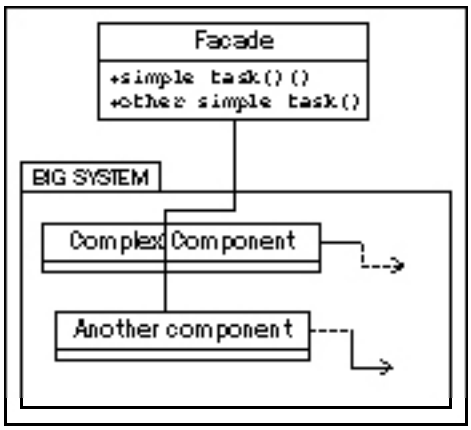
It works but it's a stupid idea. In this particular instance, such an adapter would be hard to maintain. We'd soon forget why we needed to add a `strip` method to a `date` class. The method name is ambiguous. That can be the nature of adapters, but creating an adapter explicitly instead of using inheritance usually clarifies its purpose.

Instead of inheritance, we can sometimes also use monkey-patching to add a method to an existing class. It won't work with the `datetime` object, as it doesn't allow attributes to be added at runtime, but in normal classes, we can just add a new method that provides the adapted interface that is required by calling code. Alternatively, we could extend or monkey-patch the `AgeCalculator` itself to replace the `calculate_age` method with something more amenable to our needs.

Finally, it is often possible to use a function as an adapter; this doesn't obviously fit the actual design of the adapter pattern, but if we recall that functions are essentially objects with a `__call__` method, it becomes an obvious adapter adaptation.

The facade pattern

The facade pattern is designed to provide a simple interface to a complex system of components. For complex tasks, we may need to interact with these objects directly, but there is often a "typical" usage for the system for which these complicated interactions aren't necessary. The facade pattern allows us to define a new object that encapsulates this typical usage of the system. Any time we want access to common functionality, we can use the single object's simplified interface. If another part of the project needs access to more complicated functionality, it is still able to interact with the system directly. The UML diagram for the facade pattern is really dependent on the subsystem, but in a cloudy way, it looks like this:



A facade is, in many ways, like an adapter. The primary difference is that the facade is trying to abstract a simpler interface out of a complex one, while the adapter is only trying to map one existing interface to another.

Let's write a simple facade for an e-mail application. The low-level library for sending e-mail in Python, as we saw in *Chapter 7*, *Python Object-oriented Shortcuts*, is quite complicated. The two libraries for receiving messages are even worse.

It would be nice to have a simple class that allows us to send a single e-mail, and list the e-mails currently in the inbox on an IMAP or POP3 connection. To keep our example short, we'll stick with IMAP and SMTP: two totally different subsystems that happen to deal with e-mail. Our facade performs only two tasks: sending an e-mail to a specific address, and checking the inbox on an IMAP connection. It makes some common assumptions about the connection, such as the host for both SMTP and IMAP is at the same address, that the username and password for both is the same, and that they use standard ports. This covers the case for many e-mail servers, but if a programmer needs more flexibility, they can always bypass the facade and access the two subsystems directly.

The class is initialized with the hostname of the e-mail server, a username, and a password to log in:

```
import smtplib
import imaplib

class EmailFacade:
    def __init__(self, host, username, password):
        self.host = host
        self.username = username
        self.password = password
```

The `send_email` method formats the e-mail address and message, and sends it using `smtplib`. This isn't a complicated task, but it requires quite a bit of fiddling to massage the "natural" input parameters that are passed into the facade to the correct format to enable `smtplib` to send the message:

```
    def send_email(self, to_email, subject, message):
        if not "@" in self.username:
            from_email = "{0}@{1}".format(
                    self.username, self.host)
        else:
            from_email = self.username
        message = ("From: {0}\r\n"
                "To: {1}\r\n"
                "Subject: {2}\r\n\r\n{3}").format(
                    from_email,
                    to_email,
                    subject,
                    message)

        smtp = smtplib.SMTP(self.host)
        smtp.login(self.username, self.password)
        smtp.sendmail(from_email, [to_email], message)
```

The `if` statement at the beginning of the method is catching whether or not the `username` is the entire "from" e-mail address or just the part on the left side of the `@` symbol; different hosts treat the login details differently.

Finally, the code to get the messages currently in the inbox is a ruddy mess; the IMAP protocol is painfully over-engineered, and the `imaplib` standard library is only a thin layer over the protocol:

```
    def get_inbox(self):
        mailbox = imaplib.IMAP4(self.host)
```

```
        mailbox.login(bytes(self.username, 'utf8'),
            bytes(self.password, 'utf8'))
        mailbox.select()
        x, data = mailbox.search(None, 'ALL')
        messages = []
        for num in data[0].split():
            x, message = mailbox.fetch(num, '(RFC822)')
            messages.append(message[0][1])
        return messages
```

Now, if we add all this together, we have a simple facade class that can send and receive messages in a fairly straightforward manner, much simpler than if we had to interact with these complex libraries directly.

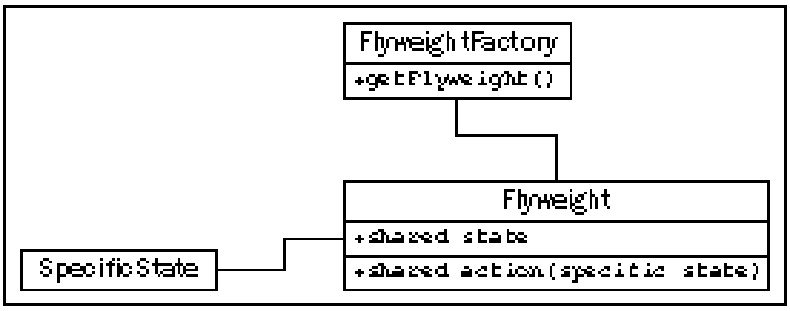
Although it is rarely named in the Python community, the facade pattern is an integral part of the Python ecosystem. Because Python emphasizes language readability, both the language and its libraries tend to provide easy-to-comprehend interfaces to complicated tasks. For example, `for` loops, `list` comprehensions, and generators are all facades into a more complicated iterator protocol. The `defaultdict` implementation is a facade that abstracts away annoying corner cases when a key doesn't exist in a dictionary. The third-party requests library is a powerful facade over less readable libraries for HTTP requests.

The flyweight pattern

The flyweight pattern is a memory optimization pattern. Novice Python programmers tend to ignore memory optimization, assuming the built-in garbage collector will take care of them. This is often perfectly acceptable, but when developing larger applications with many related objects, paying attention to memory concerns can have a huge payoff.

The flyweight pattern basically ensures that objects that share a state can use the same memory for that shared state. It is often implemented only after a program has demonstrated memory problems. It may make sense to design an optimal configuration from the beginning in some situations, but bear in mind that premature optimization is the most effective way to create a program that is too complicated to maintain.

Let's have a look at the UML diagram for the flyweight pattern:



Each **Flyweight** has no specific state; any time it needs to perform an operation on **SpecificState**, that state needs to be passed into the **Flyweight** by the calling code. Traditionally, the factory that returns a flyweight is a separate object; its purpose is to return a flyweight for a given key identifying that flyweight. It works like the singleton pattern we discussed in *Chapter 10*, *Python Design Patterns I*; if the flyweight exists, we return it; otherwise, we create a new one. In many languages, the factory is implemented, not as a separate object, but as a static method on the `Flyweight` class itself.

Think of an inventory system for car sales. Each individual car has a specific serial number and is a specific color. But most of the details about that car are the same for all cars of a particular model. For example, the Honda Fit DX model is a bare-bones car with few features. The LX model has A/C, tilt, cruise, and power windows and locks. The Sport model has fancy wheels, a USB charger, and a spoiler. Without the flyweight pattern, each individual car object would have to store a long list of which features it did and did not have. Considering the number of cars Honda sells in a year, this would add up to a huge amount of wasted memory. Using the flyweight pattern, we can instead have shared objects for the list of features associated with a model, and then simply reference that model, along with a serial number and color, for individual vehicles. In Python, the flyweight factory is often implemented using that funky `__new__` constructor, similar to what we did with the singleton pattern. Unlike singleton, which only needs to return one instance of the class, we need to be able to return different instances depending on the keys. We could store the items in a dictionary and look them up based on the key. This solution is problematic, however, because the item will remain in memory as long as it is in the dictionary. If we sold out of LX model Fits, the Fit flyweight is no longer necessary, yet it will still be in the dictionary. We could, of course, clean this up whenever we sell a car, but isn't that what a garbage collector is for?

We can solve this by taking advantage of Python's `weakref` module. This module provides a `WeakValueDictionary` object, which basically allows us to store items in a dictionary without the garbage collector caring about them. If a value is in a weak referenced dictionary and there are no other references to that object stored anywhere in the application (that is, we sold out of LX models), the garbage collector will eventually clean up for us.

Let's build the factory for our car flyweights first:

```
import weakref

class CarModel:
    _models = weakref.WeakValueDictionary()

    def __new__(cls, model_name, *args, **kwargs):
        model = cls._models.get(model_name)
        if not model:
            model = super().__new__(cls)
            cls._models[model_name] = model

        return model
```

Basically, whenever we construct a new flyweight with a given name, we first look up that name in the weak referenced dictionary; if it exists, we return that model; if not, we create a new one. Either way, we know the `__init__` method on the flyweight will be called every time, regardless of whether it is a new or existing object. Our `__init__` method can therefore look like this:

```
    def __init__(self, model_name, air=False, tilt=False,
            cruise_control=False, power_locks=False,
            alloy_wheels=False, usb_charger=False):
        if not hasattr(self, "initted"):
            self.model_name = model_name
            self.air = air
            self.tilt = tilt
            self.cruise_control = cruise_control
            self.power_locks = power_locks
            self.alloy_wheels = alloy_wheels
            self.usb_charger = usb_charger
            self.initted=True
```

The `if` statement ensures that we only initialize the object the first time `__init__` is called. This means we can call the factory later with just the model name and get the same flyweight object back. However, because the flyweight will be garbage-collected if no external references to it exist, we have to be careful not to accidentally create a new flyweight with null values.

Let's add a method to our flyweight that hypothetically looks up a serial number on a specific model of vehicle, and determines if it has been involved in any accidents. This method needs access to the car's serial number, which varies from car to car; it cannot be stored with the flyweight. Therefore, this data must be passed into the method by the calling code:

```
        def check_serial(self, serial_number):
            print("Sorry, we are unable to check "
                    "the serial number {0} on the {1} "
                    "at this time".format(
                        serial_number, self.model_name))
```

We can define a class that stores the additional information, as well as a reference to the flyweight:

```
    class Car:
        def __init__(self, model, color, serial):
            self.model = model
            self.color = color
            self.serial = serial

        def check_serial(self):
            return self.model.check_serial(self.serial)
```

We can also keep track of the available models as well as the individual cars on the lot:

```
>>> dx = CarModel("FIT DX")
>>> lx = CarModel("FIT LX", air=True, cruise_control=True,
... power_locks=True, tilt=True)
>>> car1 = Car(dx, "blue", "12345")
>>> car2 = Car(dx, "black", "12346")
>>> car3 = Car(lx, "red", "12347")
```

Now, let's demonstrate the weak referencing at work:

```
>>> id(lx)
3071620300
>>> del lx
>>> del car3
>>> import gc
>>> gc.collect()
0
```

```
>>> lx = CarModel("FIT LX", air=True, cruise_control=True,
... power_locks=True, tilt=True)
>>> id(lx)
3071576140
>>> lx = CarModel("FIT LX")
>>> id(lx)
3071576140
>>> lx.air
True
```

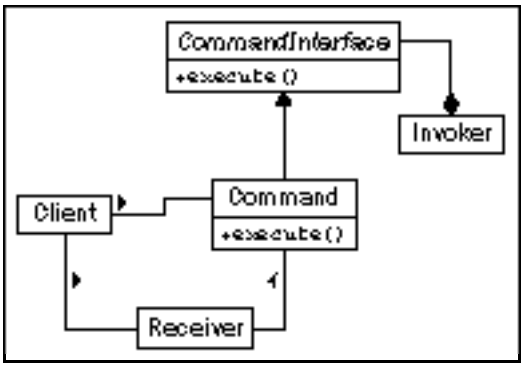
The `id` function tells us the unique identifier for an object. When we call it a second time, after deleting all references to the LX model and forcing garbage collection, we see that the ID has changed. The value in the `CarModel __new__` factory dictionary was deleted and a fresh one created. If we then try to construct a second `CarModel` instance, however, it returns the same object (the IDs are the same), and, even though we did not supply any arguments in the second call, the `air` variable is still set to `True`. This means the object was not initialized the second time, just as we designed.

Obviously, using the flyweight pattern can be more complicated than just storing features on a single car class. When should we choose to use it? The flyweight pattern is designed for conserving memory; if we have hundreds of thousands of similar objects, combining similar properties into a flyweight can have an enormous impact on memory consumption. It is common for programming solutions that optimize CPU, memory, or disk space result in more complicated code than their unoptimized brethren. It is therefore important to weigh up the tradeoffs when deciding between code maintainability and optimization. When choosing optimization, try to use patterns such as flyweight to ensure that the complexity introduced by optimization is confined to a single (well documented) section of the code.

The command pattern

The command pattern adds a level of abstraction between actions that must be done, and the object that invokes those actions, normally at a later time. In the command pattern, client code creates a `Command` object that can be executed at a later date. This object knows about a receiver object that manages its own internal state when the command is executed on it. The `Command` object implements a specific interface (typically it has an `execute` or `do_action` method, and also keeps track of any arguments required to perform the action. Finally, one or more `Invoker` objects execute the command at the correct time.

Here's the UML diagram:



A common example of the command pattern is actions on a graphical window. Often, an action can be invoked by a menu item on the menu bar, a keyboard shortcut, a toolbar icon, or a context menu. These are all examples of `Invoker` objects. The actions that actually occur, such as `Exit`, `Save`, or `Copy`, are implementations of `CommandInterface`. A GUI window to receive exit, a document to receive save, and `ClipboardManager` to receive copy commands, are all examples of possible `Receivers`.

Let's implement a simple command pattern that provides commands for `Save` and `Exit` actions. We'll start with some modest receiver classes:

```
import sys

class Window:
    def exit(self):
        sys.exit(0)

class Document:
    def __init__(self, filename):
        self.filename = filename
        self.contents = "This file cannot be modified"

    def save(self):
        with open(self.filename, 'w') as file:
            file.write(self.contents)
```

These mock classes model objects that would likely be doing a lot more in a working environment. The window would need to handle mouse movement and keyboard events, and the document would need to handle character insertion, deletion, and selection. But for our example these two classes will do what we need.

Now let's define some invoker classes. These will model toolbar, menu, and keyboard events that can happen; again, they aren't actually hooked up to anything, but we can see how they are decoupled from the command, receiver, and client code:

```
class ToolbarButton:
    def __init__(self, name, iconname):
        self.name = name
        self.iconname = iconname

    def click(self):
        self.command.execute()

class MenuItem:
    def __init__(self, menu_name, menuitem_name):
        self.menu = menu_name
        self.item = menuitem_name

    def click(self):
        self.command.execute()

class KeyboardShortcut:
    def __init__(self, key, modifier):
        self.key = key
        self.modifier = modifier

    def keypress(self):
        self.command.execute()
```

Notice how the various action methods each call the `execute` method on their respective commands? This code doesn't show the `command` attribute being set on each object. They could be passed into the `__init__` function, but because they may be changed (for example, with a customizable keybinding editor), it makes more sense to set the attributes on the objects afterwards.

Now, let's hook up the commands themselves:

```
class SaveCommand:
    def __init__(self, document):
        self.document = document

    def execute(self):
        self.document.save()

class ExitCommand:
```

```
    def __init__(self, window):
        self.window = window

    def execute(self):
        self.window.exit()
```

These commands are straightforward; they demonstrate the basic pattern, but it is important to note that we can store state and other information with the command if necessary. For example, if we had a command to insert a character, we could maintain state for the character currently being inserted.

Now all we have to do is hook up some client and test code to make the commands work. For basic testing, we can just include this at the end of the script:

```
window = Window()
document = Document("a_document.txt")
save = SaveCommand(document)
exit = ExitCommand(window)

save_button = ToolbarButton('save', 'save.png')
save_button.command = save
save_keystroke = KeyboardShortcut("s", "ctrl")
save_keystroke.command = save
exit_menu = MenuItem("File", "Exit")
exit_menu.command = exit
```

First we create two receivers and two commands. Then we create several of the available invokers and set the correct command on each of them. To test, we can use `python3 -i filename.py` and run code like `exit_menu.click()`, which will end the program, or `save_keystroke.keystroke()`, which will save the fake file.

Unfortunately, the preceding examples do not feel terribly Pythonic. They have a lot of "boilerplate code" (code that does not accomplish anything, but only provides structure to the pattern), and the `Command` classes are all eerily similar to each other. Perhaps we could create a generic command object that takes a function as a callback?

In fact, why bother? Can we just use a function or method object for each command? Instead of an object with an `execute()` method, we can write a function and use that as the command directly. This is a common paradigm for the command pattern in Python:

```
import sys

class Window:
```

```
        def exit(self):
            sys.exit(0)

    class MenuItem:
        def click(self):
            self.command()

    window = Window()
    menu_item = MenuItem()
    menu_item.command = window.exit
```

Now that looks a lot more like Python. At first glance, it looks like we've removed the command pattern altogether, and we've tightly connected the `menu_item` and `Window` classes. But if we look closer, we find there is no tight coupling at all. Any callable can be set up as the command on the `MenuItem`, just as before. And the `Window.exit` method can be attached to any invoker. Most of the flexibility of the command pattern has been maintained. We have sacrificed complete decoupling for readability, but this code is, in my opinion, and that of many Python programmers, more maintainable than the fully abstracted version.

Of course, since we can add a `__call__` method to any object, we aren't restricted to functions. The previous example is a useful shortcut when the method being called doesn't have to maintain state, but in more advanced usage, we can use this code as well:

```
class Document:
    def __init__(self, filename):
        self.filename = filename
        self.contents = "This file cannot be modified"

    def save(self):
        with open(self.filename, 'w') as file:
            file.write(self.contents)

class KeyboardShortcut:
    def keypress(self):
        self.command()
class SaveCommand:
    def __init__(self, document):
        self.document = document

    def __call__(self):
```

```
            self.document.save()

document = Document("a_file.txt")
shortcut = KeyboardShortcut()
save_command = SaveCommand(document)
shortcut.command = save_command
```

Here we have something that looks like the first command pattern, but a bit more idiomatic. As you can see, making the invoker call a callable instead of a command object with an execute method has not restricted us in any way. In fact, it's given us more flexibility. We can link to functions directly when that works, yet we can build a complete callable command object when the situation calls for it.

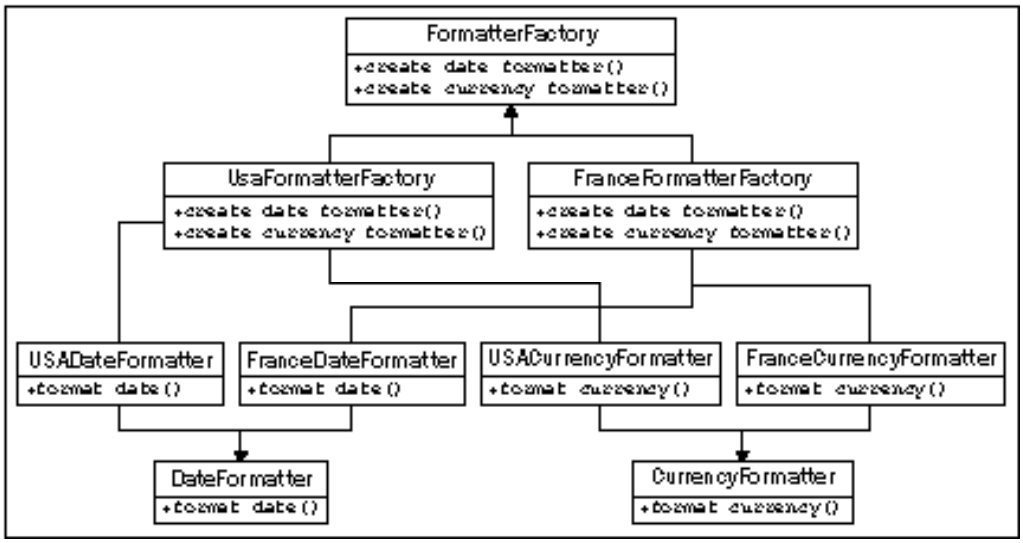
The command pattern is often extended to support undoable commands. For example, a text program may wrap each insertion in a separate command with not only an `execute` method, but also an `undo` method that will delete that insertion. A graphics program may wrap each drawing action (rectangle, line, freehand pixels, and so on) in a command that has an `undo` method that resets the pixels to their original state. In such cases, the decoupling of the command pattern is much more obviously useful, because each action has to maintain enough of its state to undo that action at a later date.

The abstract factory pattern

The abstract factory pattern is normally used when we have multiple possible implementations of a system that depend on some configuration or platform issue. The calling code requests an object from the abstract factory, not knowing exactly what class of object will be returned. The underlying implementation returned may depend on a variety of factors, such as current locale, operating system, or local configuration.

Common examples of the abstract factory pattern include code for operating-system independent toolkits, database backends, and country-specific formatters or calculators. An operating-system-independent GUI toolkit might use an abstract factory pattern that returns a set of WinForm widgets under Windows, Cocoa widgets under Mac, GTK widgets under Gnome, and QT widgets under KDE. Django provides an abstract factory that returns a set of object relational classes for interacting with a specific database backend (MySQL, PostgreSQL, SQLite, and others) depending on a configuration setting for the current site. If the application needs to be deployed in multiple places, each one can use a different database backend by changing only one configuration variable. Different countries have different systems for calculating taxes, subtotals, and totals on retail merchandise; an abstract factory can return a particular tax calculation object.

The UML class diagram for an abstract factory pattern is hard to understand without a specific example, so let's turn things around and create a concrete example first. We'll create a set of formatters that depend on a specific locale and help us format dates and currencies. There will be an abstract factory class that picks the specific factory, as well as a couple example concrete factories, one for France and one for the USA. Each of these will create formatter objects for dates and times, which can be queried to format a specific value. Here's the diagram:



Comparing that image to the earlier simpler text shows that a picture is not always worth a thousand words, especially considering we haven't even allowed for factory selection code here.

Of course, in Python, we don't have to implement any interface classes, so we can discard `DateFormatter`, `CurrencyFormatter`, and `FormatterFactory`. The formatting classes themselves are pretty straightforward, if verbose:

```
class FranceDateFormatter:
    def format_date(self, y, m, d):
        y, m, d = (str(x) for x in (y,m,d))
        y = '20' + y if len(y) == 2 else y
        m = '0' + m if len(m) == 1 else m
        d = '0' + d if len(d) == 1 else d
        return("{0}/{1}/{2}".format(d,m,y))

class USADateFormatter:
```

```
        def format_date(self, y, m, d):
            y, m, d = (str(x) for x in (y,m,d))
            y = '20' + y if len(y) == 2 else y
            m = '0' + m if len(m) == 1 else m
            d = '0' + d if len(d) == 1 else d
            return("{0}-{1}-{2}".format(m,d,y))

    class FranceCurrencyFormatter:
        def format_currency(self, base, cents):
            base, cents = (str(x) for x in (base, cents))
            if len(cents) == 0:
                cents = '00'
            elif len(cents) == 1:
                cents = '0' + cents

            digits = []
            for i,c in enumerate(reversed(base)):
                if i and not i % 3:
                    digits.append(' ')
                digits.append(c)
            base = ''.join(reversed(digits))
            return "{0}€{1}".format(base, cents)

    class USACurrencyFormatter:
        def format_currency(self, base, cents):
            base, cents = (str(x) for x in (base, cents))
            if len(cents) == 0:
                cents = '00'
            elif len(cents) == 1:
                cents = '0' + cents
            digits = []
            for i,c in enumerate(reversed(base)):
                if i and not i % 3:
                    digits.append(',')
                digits.append(c)
            base = ''.join(reversed(digits))
            return "${0}.{1}".format(base, cents)
```

These classes use some basic string manipulation to try to turn a variety of possible inputs (integers, strings of different lengths, and others) into the following formats:

	USA	**France**
Date	mm-dd-yyyy	dd/mm/yyyy
Currency	$14,500.50	14 500€50

There could obviously be more validation on the input in this code, but let's keep it simple and dumb for this example.

Now that we have the formatters set up, we just need to create the formatter factories:

```
class USAFormatterFactory:
    def create_date_formatter(self):
        return USADateFormatter()
    def create_currency_formatter(self):
        return USACurrencyFormatter()

class FranceFormatterFactory:
    def create_date_formatter(self):
        return FranceDateFormatter()
    def create_currency_formatter(self):
        return FranceCurrencyFormatter()
```

Now we set up the code that picks the appropriate formatter. Since this is the kind of thing that only needs to be set up once, we could make it a singleton—except singletons aren't very useful in Python. Let's just make the current formatter a module-level variable instead:

```
country_code = "US"
factory_map = {
        "US": USAFormatterFactory,
        "FR": FranceFormatterFactory}
formatter_factory = factory_map.get(country_code)()
```

In this example, we hardcode the current country code; in practice, it would likely introspect the locale, the operating system, or a configuration file to choose the code. This example uses a dictionary to associate the country codes with factory classes. Then we grab the correct class from the dictionary and instantiate it.

It is easy to see what needs to be done when we want to add support for more countries: create the new formatter classes and the abstract factory itself. Bear in mind that `Formatter` classes might be reused; for example, Canada formats its currency the same way as the USA, but its date format is more sensible than its Southern neighbor.

Abstract factories often return a singleton object, but this is not required; in our code, it's returning a new instance of each formatter every time it's called. There's no reason the formatters couldn't be stored as instance variables and the same instance returned for each factory.

Looking back at these examples, we see that, once again, there appears to be a lot of boilerplate code for factories that just doesn't feel necessary in Python. Often, the requirements that might call for an abstract factory can be more easily fulfilled by using a separate module for each factory type (for example: the USA and France), and then ensuring that the correct module is being accessed in a factory module. The package structure for such modules might look like this:

```
localize/
    __init__.py
    backends/
        __init__.py
        USA.py
        France.py
        …
```

The trick is that `__init__.py` in the `localize` package can contain logic that redirects all requests to the correct backend. There is a variety of ways this could be done.

If we know that the backend is never going to change dynamically (that is, without a restart), we can just put some `if` statements in `__init__.py` that check the current country code, and use the usually unacceptable `from .backends.USA import *` syntax to import all variables from the appropriate backend. Or, we could import each of the backends and set a `current_backend` variable to point at a specific module:

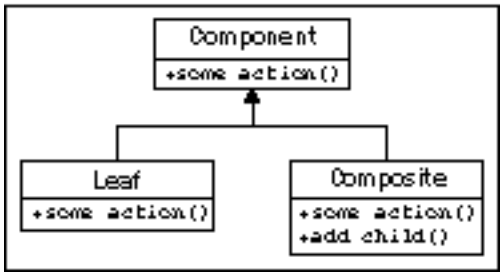
```
from .backends import USA, France

if country_code == "US":
    current_backend = USA
```

Depending on which solution we choose, our client code would have to call either `localize.format_date` or `localize.current_backend.format_date` to get a date formatted in the current country's locale. The end result is much more Pythonic than the original abstract factory pattern, and, in typical usage, just as flexible.

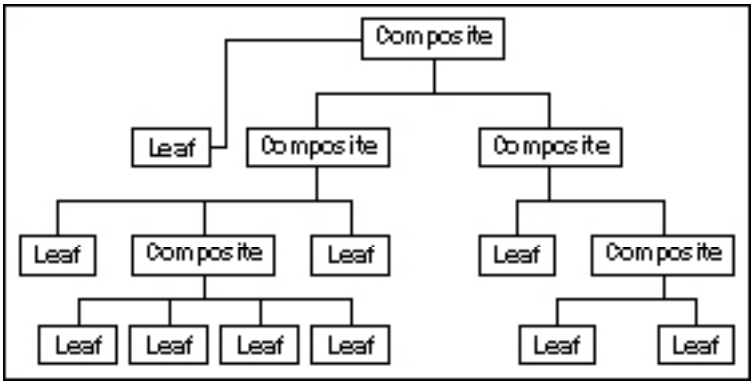
The composite pattern

The composite pattern allows complex tree-like structures to be built from simple components. These components, called composite objects, are able to behave sort of like a container and sort of like a variable depending on whether they have child components. Composite objects are container objects, where the content may actually be another composite object.

Traditionally, each component in a composite object must be either a leaf node (that cannot contain other objects) or a composite node. The key is that both composite and leaf nodes can have the same interface. The UML diagram is very simple:



This simple pattern, however, allows us to create complex arrangements of elements, all of which satisfy the interface of the component object. Here is a concrete instance of such a complicated arrangement:



The composite pattern is commonly useful in file/folder-like trees. Regardless of whether a node in the tree is a normal file or a folder, it is still subject to operations such as moving, copying, or deleting the node. We can create a component interface that supports these operations, and then use a composite object to represent folders, and leaf nodes to represent normal files.

Of course, in Python, once again, we can take advantage of duck typing to implicitly provide the interface, so we only need to write two classes. Let's define these interfaces first:

```
class Folder:
    def __init__(self, name):
        self.name = name
        self.children = {}

    def add_child(self, child):
        pass

    def move(self, new_path):
        pass

    def copy(self, new_path):
        pass

    def delete(self):
        pass

class File:
    def __init__(self, name, contents):
        self.name = name
        self.contents = contents

    def move(self, new_path):
        pass

    def copy(self, new_path):
        pass

    def delete(self):
        pass
```

For each folder (composite) object, we maintain a dictionary of children. Often, a list is sufficient, but in this case, a dictionary will be useful for looking up children by name. Our paths will be specified as node names separated by the `/` character, similar to paths in a Unix shell.

Thinking about the methods involved, we can see that moving or deleting a node behaves in a similar way, regardless of whether or not it is a file or folder node. Copying, however, has to do a recursive copy for folder nodes, while copying a file node is a trivial operation.

To take advantage of the similar operations, we can extract some of the common methods into a parent class. Let's take that discarded `Component` interface and change it to a base class:

```
class Component:
    def __init__(self, name):
        self.name = name

    def move(self, new_path):
        new_folder =get_path(new_path)
        del self.parent.children[self.name]
        new_folder.children[self.name] = self
        self.parent = new_folder

    def delete(self):
        del self.parent.children[self.name]

class Folder(Component):
    def __init__(self, name):
        super().__init__(name)
        self.children = {}

    def add_child(self, child):
        pass

    def copy(self, new_path):
        pass

class File(Component):
    def __init__(self, name, contents):
        super().__init__(name)
        self.contents = contents

    def copy(self, new_path):
        pass

root = Folder('')
def get_path(path):
```

```
    names = path.split('/')[1:]
    node = root
    for name in names:
        node = node.children[name]
    return node
```

We've created the `move` and `delete` methods on the `Component` class. Both of them access a mysterious `parent` variable that we haven't set yet. The `move` method uses a module-level `get_path` function that finds a node from a predefined root node, given a path. All files will be added to this root node or a child of that node. For the `move` method, the target should be a currently existing folder, or we'll get an error. As with many of the examples in technical books, error handling is woefully absent, to help focus on the principles under consideration.

Let's set up that mysterious `parent` variable first; this happens, in the folder's `add_child` method:

```
    def add_child(self, child):
        child.parent = self
        self.children[child.name] = child
```

Well, that was easy enough. Let's see if our composite file hierarchy is working properly:

```
$ python3 -i 1261_09_18_add_child.py


>>> folder1 = Folder('folder1')
>>> folder2 = Folder('folder2')
>>> root.add_child(folder1)
>>> root.add_child(folder2)
>>> folder11 = Folder('folder11')
>>> folder1.add_child(folder11)
>>> file111 = File('file111', 'contents')
>>> folder11.add_child(file111)
>>> file21 = File('file21', 'other contents')
>>> folder2.add_child(file21)
>>> folder2.children
{'file21': <__main__.File object at 0xb7220a4c>}
>>> folder2.move('/folder1/folder11')
>>> folder11.children
{'folder2': <__main__.Folder object at 0xb722080c>, 'file111': <__main__.
File object at 0xb72209ec>}
```

```
>>> file21.move('/folder1')
>>> folder1.children
{'file21': <__main__.File object at 0xb7220a4c>, 'folder11': <__main__.
Folder object at 0xb722084c>}
```

Yes, we can create folders, add folders to other folders, add files to folders, and move them around! What more could we ask for in a file hierarchy?

Well, we could ask for copying to be implemented, but to conserve trees, let's leave that as an exercise.

The composite pattern is extremely useful for a variety of tree-like structures, including GUI widget hierarchies, file hierarchies, tree sets, graphs, and HTML DOM. It can be a useful pattern in Python when implemented according to the traditional implementation, as the example earlier demonstrated. Sometimes, if only a shallow tree is being created, we can get away with a list of lists or a dictionary of dictionaries, and do not need to implement custom component, leaf, and composite classes. Other times, we can get away with implementing only one composite class, and treating leaf and composite objects as a single class. Alternatively, Python's duck typing can make it easy to add other objects to a composite hierarchy, as long as they have the correct interface.

Exercises

Before diving into exercises for each design pattern, take a moment to implement the `copy` method for the `File` and `Folder` objects in the previous section. The `File` method should be quite trivial; just create a new node with the same name and contents, and add it to the new parent folder. The `copy` method on `Folder` is quite a bit more complicated, as you first have to duplicate the folder, and then recursively copy each of its children to the new location. You can call the `copy()` method on the children indiscriminately, regardless of whether each is a file or a folder object. This will drive home just how powerful the composite pattern can be.

Now, as with the previous chapter, look at the patterns we've discussed, and consider ideal places where you might implement them. You may want to apply the adapter pattern to existing code, as it is usually applicable when interfacing with existing libraries, rather than new code. How can you use an adapter to force two interfaces to interact with each other correctly?

Can you think of a system complex enough to justify using the facade pattern? Consider how facades are used in real-life situations, such as the driver-facing interface of a car, or the control panel in a factory. It is similar in software, except the users of the facade interface are other programmers, rather than people trained to use them. Are there complex systems in your latest project that could benefit from the facade pattern?

It's possible you don't have any huge, memory-consuming code that would benefit from the flyweight pattern, but can you think of situations where it might be useful? Anywhere that large amounts of overlapping data need to be processed, a flyweight is waiting to be used. Would it be useful in the banking industry? In web applications? At what point does the flyweight pattern make sense? When is it overkill?

What about the command pattern? Can you think of any common (or better yet, uncommon) examples of places where the decoupling of action from invocation would be useful? Look at the programs you use on a daily basis, and imagine how they are implemented internally. It's likely that many of them use the command pattern for one purpose or another.

The abstract factory pattern, or the somewhat more Pythonic derivatives we discussed, can be very useful for creating one-touch-configurable systems. Can you think of places where such systems are useful?

Finally, consider the composite pattern. There are tree-like structures all around us in programming; some of them, like our file hierarchy example, are blatant; others are fairly subtle. What situations might arise where the composite pattern would be useful? Can you think of places where you can use it in your own code? What if you adapted the pattern slightly; for example, to contain different types of leaf or composite nodes for different types of objects?

Summary

In this chapter, we went into detail on several more design patterns, covering their canonical descriptions as well as alternatives for implementing them in Python, which is often more flexible and versatile than traditional object-oriented languages. The adapter pattern is useful for matching interfaces, while the facade pattern is suited to simplifying them. Flyweight is a complicated pattern and only useful if memory optimization is required. In Python, the command pattern is often more aptly implemented using first class functions as callbacks. Abstract factories allow run-time separation of implementations depending on configuration or system information. The composite pattern is used universally for tree-like structures.

In the next chapter, we'll discuss how important it is to test Python programs, and how to do it.

12
Testing Object-oriented Programs

Skilled Python programmers agree that testing is one of the most important aspects of software development. Even though this chapter is placed near the end of the book, it is not an afterthought; everything we have studied so far will help us when writing tests. We'll be studying:

- The importance of unit testing and test-driven development
- The standard `unittest` module
- The `py.test` automated testing suite
- The `mock` module
- Code coverage
- Cross-platform testing with `tox`

Why test?

A large collection of programmers already know how important it is to test their code. If you're among them, feel free to skim this section. You'll find the next section—where we actually see how to do the tests in Python—much more scintillating. If you're not convinced of the importance of testing, I promise that your code is broken, you just don't know it. Read on!

Some people argue that testing is more important in Python code because of its dynamic nature; compiled languages such as Java and C++ are occasionally thought to be somehow "safer" because they enforce type checking at compile time. However, Python tests rarely check types. They're checking values. They're making sure that the right attributes have been set at the right time or that the sequence has the right length, order, and values. These higher-level things need to be tested in any language. The real reason Python programmers test more than programmers of other languages is that it is so easy to test in Python!

But why test? Do we really need to test? What if we didn't test? To answer those questions, write a tic-tac-toe game from scratch without any testing at all. Don't run it until it is completely written, start to finish. Tic-tac-toe is fairly simple to implement if you make both players human players (no artificial intelligence). You don't even have to try to calculate who the winner is. Now run your program. And fix all the errors. How many were there? I recorded eight on my tic-tac-toe implementation, and I'm not sure I caught them all. Did you?

We need to test our code to make sure it works. Running the program, as we just did, and fixing the errors is one crude form of testing. Python programmers are able to write a few lines of code and run the program to make sure those lines are doing what they expect. But changing a few lines of code can affect parts of the program that the developer hadn't realized will be influenced by the changes, and therefore won't test it. Furthermore, as a program grows, the various paths that the interpreter can take through that code also grow, and it quickly becomes impossible to manually test all of them.

To handle this, we write automated tests. These are programs that automatically run certain inputs through other programs or parts of programs. We can run these test programs in seconds and cover more possible input situations than one programmer would think to test every time they change something.

There are four main reasons to write tests:

- To ensure that code is working the way the developer thinks it should
- To ensure that code continues working when we make changes
- To ensure that the developer understood the requirements
- To ensure that the code we are writing has a maintainable interface

The first point really doesn't justify the time it takes to write a test; we can simply test the code directly in the interactive interpreter. But when we have to perform the same sequence of test actions multiple times, it takes less time to automate those steps once and then run them whenever necessary. It is a good idea to run tests whenever we change code, whether it is during initial development or maintenance releases. When we have a comprehensive set of automated tests, we can run them after code changes and know that we didn't inadvertently break anything that was tested.

The last two points are more interesting. When we write tests for code, it helps us design the API, interface, or pattern that code takes. Thus, if we misunderstood the requirements, writing a test can help highlight that misunderstanding. On the other side, if we're not certain how we want to design a class, we can write a test that interacts with that class so we have an idea what the most natural way to test it would be. In fact, it is often beneficial to write the tests before we write the code we are testing.

Test-driven development

"Write tests first" is the mantra of test-driven development. Test-driven development takes the "untested code is broken code" concept one step further and suggests that only unwritten code should be untested. Do not write any code until you have written the tests for this code. So the first step is to write a test that proves the code would work. Obviously, the test is going to fail, since the code hasn't been written. Then write the code that ensures the test passes. Then write another test for the next segment of code.

Test-driven development is fun. It allows us to build little puzzles to solve. Then we implement the code to solve the puzzles. Then we make a more complicated puzzle, and we write code that solves the new puzzle without unsolving the previous one.

There are two goals to the test-driven methodology. The first is to ensure that tests really get written. It's so very easy, after we have written code, to say: "Hmm, it seems to work. I don't have to write any tests for this. It was just a small change, nothing could have broken." If the test is already written before we write the code, we will know exactly when it works (because the test will pass), and we'll know in the future if it is ever broken by a change we, or someone else has made.

Secondly, writing tests first forces us to consider exactly how the code will be interacted with. It tells us what methods objects need to have and how attributes will be accessed. It helps us break up the initial problem into smaller, testable problems, and then to recombine the tested solutions into larger, also tested, solutions. Writing tests can thus become a part of the design process. Often, if we're writing a test for a new object, we discover anomalies in the design that force us to consider new aspects of the software.

As a concrete example, imagine writing code that uses an object-relational mapper to store object properties in a database. It is common to use an automatically assigned database ID in such objects. Our code might use this ID for various purposes. If we are writing a test for such code, before we write it, we may realize that our design is faulty because objects do not have these IDs until they have been saved to the database. If we want to manipulate an object without saving it in our test, it will highlight this problem before we have written code based on the faulty premise.

Testing makes software better. Writing tests before we release the software makes it better before the end user sees or purchases the buggy version (I have worked for companies that thrive on the "the users can test it" philosophy. It's not a healthy business model!). Writing tests before we write software makes it better the first time it is written.

Unit testing

Let's start our exploration with Python's built-in test library. This library provides a common interface for **unit tests**. Unit tests focus on testing the least amount of code possible in any one test. Each one tests a single unit of the total amount of available code.

The Python library for this is called, unsurprisingly, `unittest`. It provides several tools for creating and running unit tests, the most important being the `TestCase` class. This class provides a set of methods that allow us to compare values, set up tests, and clean up when they have finished.

When we want to write a set of unit tests for a specific task, we create a subclass of `TestCase`, and write individual methods to do the actual testing. These methods must all start with the name `test`. When this convention is followed, the tests automatically run as part of the test process. Normally, the tests set some values on an object and then run a method, and use the built-in comparison methods to ensure that the right results were calculated. Here's a very simple example:

```
import unittest

class CheckNumbers(unittest.TestCase):
    def test_int_float(self):
        self.assertEqual(1, 1.0)

if __name__ == "__main__":
    unittest.main()
```

This code simply subclasses the `TestCase` class and adds a method that calls the `TestCase.assertEqual` method. This method will either succeed or raise an exception, depending on whether the two parameters are equal. If we run this code, the `main` function from `unittest` will give us the following output:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Did you know that floats and integers can compare as equal? Let's add a failing test:

```
        def test_str_float(self):
            self.assertEqual(1, "1")
```

The output of this code is more sinister, as integers and strings are not considered equal:

```
.F
======================================================================
FAIL: test_str_float (__main__.CheckNumbers)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "simplest_unittest.py", line 8, in test_str_float
    self.assertEqual(1, "1")
AssertionError: 1 != '1'

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

The dot on the first line indicates that the first test (the one we wrote before) passed successfully; the letter `F` after it shows that the second test failed. Then, at the end, it gives us some informative output telling us how and where the test failed, along with a summary of the number of failures.

We can have as many test methods on one `TestCase` class as we like; as long as the method name begins with `test`, the test runner will execute each one as a separate test. Each test should be completely independent of other tests. Results or calculations from a previous test should have no impact on the current test. The key to writing good unit tests is to keep each test method as short as possible, testing a small unit of code with each test case. If your code does not seem to naturally break up into such testable units, it's probably a sign that your design needs rethinking.

Assertion methods

The general layout of a test case is to set certain variables to known values, run one or more functions, methods, or processes, and then "prove" that correct expected results were returned or calculated by using `TestCase` assertion methods.

There are a few different assertion methods available to confirm that specific results have been achieved. We just saw `assertEqual`, which will cause a test failure if the two parameters do not pass an equality check. The inverse, `assertNotEqual`, will fail if the two parameters do compare as equal. The `assertTrue` and `assertFalse` methods each accept a single expression, and fail if the expression does not pass an `if` test. These tests are not checking for the Boolean values `True` or `False`. Rather, they test the same condition as though an `if` statement were used: `False`, `None`, `0`, or an empty list, dictionary, string, set, or tuple would pass a call to the `assertFalse` method, while nonzero numbers, containers with values in them, or the value `True` would succeed when calling the `assertTrue` method.

There is an `assertRaises` method that can be used to ensure a specific function call raises a specific exception or, optionally, it can be used as a context manager to wrap inline code. The test passes if the code inside the `with` statement raises the proper exception; otherwise, it fails. Here's an example of both versions:

```
import unittest

def average(seq):
    return sum(seq) / len(seq)

class TestAverage(unittest.TestCase):
    def test_zero(self):
        self.assertRaises(ZeroDivisionError,
                average,
```

```
                []):

        def test_with_zero(self):
            with self.assertRaises(ZeroDivisionError):
                average([])

if __name__ == "__main__":
    unittest.main()
```

The context manager allows us to write the code the way we would normally write it (by calling functions or executing code directly), rather than having to wrap the function call in another function call.

There are also several other assertion methods, summarized in the following table:

Methods	Description

`assertGreater`
`assertGreaterEqual`
`assertLess`
`assertLessEqual`
	Accept two comparable objects and ensure the named inequality holds.

`assertIn` `assertNotIn`	Ensure an element is (or is not) an element in a container object.
`assertIsNone` `assertIsNotNone`	Ensure an element is (or is not) the exact value `None` (but not another falsey value).
`assertSameElements`	Ensure two container objects have the same elements, ignoring the order.

`assertSequenceEqualassertDictEqual`
`assertSetEqual`
`assertListEqual`
`assertTupleEqual`
	Ensure two containers have the same elements in the same order. If there's a failure, show a code diff comparing the two lists to see where they differ. The last four methods also test the type of the list.

Each of the assertion methods accepts an optional argument named `msg`. If supplied, it is included in the error message if the assertion fails. This is useful for clarifying what was expected or explaining where a bug may have occurred to cause the assertion to fail.

Reducing boilerplate and cleaning up

After writing a few small tests, we often find that we have to do the same setup code for several related tests. For example, the following `list` subclass has three methods for statistical calculations:

```
from collections import defaultdict

class StatsList(list):
    def mean(self):
        return sum(self) / len(self)

    def median(self):
        if len(self) % 2:
            return self[int(len(self) / 2)]
        else:
            idx = int(len(self) / 2)
            return (self[idx] + self[idx-1]) / 2

    def mode(self):
        freqs = defaultdict(int)
        for item in self:
            freqs[item] += 1
        mode_freq = max(freqs.values())
        modes = []
        for item, value in freqs.items():
            if value == mode_freq:
                modes.append(item)
        return modes
```

Clearly, we're going to want to test situations with each of these three methods that have very similar inputs; we'll want to see what happens with empty lists or with lists containing non-numeric values or with lists containing a normal dataset. We can use the `setUp` method on the `TestCase` class to do initialization for each test. This method accepts no arguments, and allows us to do arbitrary setup before each test is run. For example, we can test all three methods on identical lists of integers as follows:

```
from stats import StatsList
import unittest

class TestValidInputs(unittest.TestCase):
    def setUp(self):
        self.stats = StatsList([1,2,2,3,3,4])

    def test_mean(self):
```

```
            self.assertEqual(self.stats.mean(), 2.5)

        def test_median(self):
            self.assertEqual(self.stats.median(), 2.5)
            self.stats.append(4)
            self.assertEqual(self.stats.median(), 3)

        def test_mode(self):
            self.assertEqual(self.stats.mode(), [2,3])
            self.stats.remove(2)
            self.assertEqual(self.stats.mode(), [3])

    if __name__ == "__main__":
        unittest.main()
```

If we run this example, it indicates that all tests pass. Notice first that the `setUp` method is never explicitly called inside the three `test_*` methods. The test suite does this on our behalf. More importantly notice how `test_median` alters the list, by adding an additional `4` to it, yet when `test_mode` is called, the list has returned to the values specified in `setUp` (if it had not, there would be two fours in the list, and the `mode` method would have returned three values). This shows that `setUp` is called individually before each test, to ensure the test class starts with a clean slate. Tests can be executed in any order, and the results of one test should not depend on any other tests.

In addition to the `setUp` method, `TestCase` offers a no-argument `tearDown` method, which can be used for cleaning up after each and every test on the class has run. This is useful if cleanup requires anything other than letting an object be garbage collected. For example, if we are testing code that does file I/O, our tests may create new files as a side effect of testing; the `tearDown` method can remove these files and ensure the system is in the same state it was before the tests ran. Test cases should never have side effects. In general, we group test methods into separate `TestCase` subclasses depending on what setup code they have in common. Several tests that require the same or similar setup will be placed in one class, while tests that require unrelated setup go in another class.

Organizing and running tests

It doesn't take long for a collection of unit tests to grow very large and unwieldy. It quickly becomes complicated to load and run all the tests at once. This is a primary goal of unit testing; it should be trivial to run all tests on our program and get a quick "yes or no" answer to the question, "Did my recent changes break any existing tests?".

Python's `discover` module basically looks for any modules in the current folder or subfolders with names that start with the characters `test`. If it finds any `TestCase` objects in these modules, the tests are executed. It's a painless way to ensure we don't miss running any tests. To use it, ensure your test modules are named `test_<something>.py` and then run the command `python3 -m unittest discover`.

Ignoring broken tests

Sometimes, a test is known to fail, but we don't want the test suite to report the failure. This may be because a broken or unfinished feature has had tests written, but we aren't currently focusing on improving it. More often, it happens because a feature is only available on a certain platform, Python version, or for advanced versions of a specific library. Python provides us with a few decorators to mark tests as expected to fail or to be skipped under known conditions.

The decorators are:

- `expectedFailure()`
- `skip(reason)`
- `skipIf(condition, reason)`
- `skipUnless(condition, reason)`

These are applied using the Python decorator syntax. The first one accepts no arguments, and simply tells the test runner not to record the test as a failure when it fails. The `skip` method goes one step further and doesn't even bother to run the test. It expects a single string argument describing why the test was skipped. The other two decorators accept two arguments, one a Boolean expression that indicates whether or not the test should be run, and a similar description. In use, these three decorators might be applied like this:

```
import unittest
import sys

class SkipTests(unittest.TestCase):
    @unittest.expectedFailure
    def test_fails(self):
        self.assertEqual(False, True)

    @unittest.skip("Test is useless")
```

```
        def test_skip(self):
            self.assertEqual(False, True)

        @unittest.skipIf(sys.version_info.minor == 4,
                "broken on 3.4")
        def test_skipif(self):
            self.assertEqual(False, True)

        @unittest.skipUnless(sys.platform.startswith('linux'),
                "broken unless on linux")
        def test_skipunless(self):
            self.assertEqual(False, True)

    if __name__ == "__main__":
        unittest.main()
```

The first test fails, but it is reported as an expected failure; the second test is never run. The other two tests may or may not be run depending on the current Python version and operating system. On my Linux system running Python 3.4, the output looks like this:

```
xssF
======================================================================
FAIL: test_skipunless (__main__.SkipTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "skipping_tests.py", line 21, in test_skipunless
    self.assertEqual(False, True)
AssertionError: False != True


----------------------------------------------------------------------
Ran 4 tests in 0.001s


FAILED (failures=1, skipped=2, expected failures=1)
```

The `x` on the first line indicates an expected failure; the two `s` characters represent skipped tests, and the `F` indicates a real failure, since the conditional to `skipUnless` was `True` on my system.

Testing with py.test

The Python `unittest` module requires a lot of boilerplate code to set up and initialize tests. It is based on the very popular JUnit testing framework for Java. It even uses the same method names (you may have noticed they don't conform to the PEP-8 naming standard, which suggests underscores rather than CamelCase to separate words in a method name) and test layout. While this is effective for testing in Java, it's not necessarily the best design for Python testing.

Because Python programmers like their code to be elegant and simple, other test frameworks have been developed, outside the standard library. Two of the more popular ones are `py.test` and `nose`. The former is more robust and has had Python 3 support for much longer, so we'll discuss it here.

Since `py.test` is not part of the standard library, you'll need to download and install it yourself; you can get it from the `py.test` home page at `http://pytest.org/`. The website has comprehensive installation instructions for a variety of interpreters and platforms, but you can usually get away with the more common python package installer, pip. Just type `pip install pytest` on your command line and you'll be good to go.

`py.test` has a substantially different layout from the `unittest` module. It doesn't require test cases to be classes. Instead, it takes advantage of the fact that Python functions are objects, and allows any properly named function to behave like a test. Rather than providing a bunch of custom methods for asserting equality, it uses the `assert` statement to verify results. This makes tests more readable and maintainable. When we run `py.test`, it will start in the current folder and search for any modules in that folder or subpackages whose names start with the characters `test_`. If any functions in this module also start with `test`, they will be executed as individual tests. Furthermore, if there are any classes in the module whose name starts with `Test`, any methods on that class that start with `test_` will also be executed in the test environment.

Let's port the simplest possible `unittest` example we wrote earlier to `py.test`:

```
def test_int_float():
    assert 1 == 1.0
```

For the exact same test, we've written two lines of more readable code, in comparison to the six lines required in our first `unittest` example.

However, we are not forbidden from writing class-based tests. Classes can be useful for grouping related tests together or for tests that need to access related attributes or methods on the class. This example shows an extended class with a passing and a failing test; we'll see that the error output is more comprehensive than that provided by the `unittest` module:

```
class TestNumbers:
    def test_int_float(self):
        assert 1 == 1.0

    def test_int_str(self):
        assert 1 == "1"
```

Notice that the class doesn't have to extend any special objects to be picked up as a test (although `py.test` will run standard `unittest TestCases` just fine). If we run `py.test <filename>`, the output looks like this:

```
============== test session starts ==============
python: platform linux2 -- Python 3.4.1 -- pytest-2.6.4
test object 1: class_pytest.py

class_pytest.py .F

==================== FAILURES====================
___________ TestNumbers.test_int_str _____________

self = <class_pytest.TestNumbers object at 0x85b4fac>

    def test_int_str(self):
>       assert 1 == "1"
E       assert 1 == '1'

class_pytest.py:7: AssertionError
====== 1 failed, 1 passed in 0.10 seconds =======
```

The output starts with some useful information about the platform and interpreter. This can be useful for sharing bugs across disparate systems. The third line tells us the name of the file being tested (if there are multiple test modules picked up, they will all be displayed), followed by the familiar `.F` we saw in the `unittest` module; the `.` character indicates a passing test, while the letter `F` demonstrates a failure.

After all tests have run, the error output for each of them is displayed. It presents a summary of local variables (there is only one in this example: the `self` parameter passed into the function), the source code where the error occurred, and a summary of the error message. In addition, if an exception other than an `AssertionError` is raised, `py.test` will present us with a complete traceback, including source code references.

By default, `py.test` suppresses output from `print` statements if the test is successful. This is useful for test debugging; when a test is failing, we can add `print` statements to the test to check the values of specific variables and attributes as the test runs. If the test fails, these values are output to help with diagnosis. However, once the test is successful, the `print` statement output is not displayed, and they can be easily ignored. We don't have to "clean up" the output by removing `print` statements. If the tests ever fail again, due to future changes, the debugging output will be immediately available.

One way to do setup and cleanup

`py.test` supports setup and teardown methods similar to those used in `unittest`, but it provides even more flexibility. We'll discuss these briefly, since they are familiar, but they are not used as extensively as in the `unittest` module, as `py.test` provides us with a powerful funcargs facility, which we'll discuss in the next section.

If we are writing class-based tests, we can use two methods called `setup_method` and `teardown_method` in basically the same way that `setUp` and `tearDown` are called in `unittest`. They are called before and after each test method in the class to perform setup and cleanup duties. There is one difference from the `unittest` methods though. Both methods accept an argument: the function object representing the method being called.

In addition, `py.test` provides other setup and teardown functions to give us more control over when setup and cleanup code is executed. The `setup_class` and `teardown_class` methods are expected to be class methods; they accept a single argument (there is no `self` argument) representing the class in question.

Finally, we have the `setup_module` and `teardown_module` functions, which are run immediately before and after all tests (in functions or classes) in that module. These can be useful for "one time" setup, such as creating a socket or database connection that will be used by all tests in the module. Be careful with this one, as it can accidentally introduce dependencies between tests if the object being set up stores the state.

That short description doesn't do a great job of explaining exactly when these methods are called, so let's look at an example that illustrates exactly when it happens:

```
def setup_module(module):
    print("setting up MODULE {0}".format(
        module.__name__))

def teardown_module(module):
    print("tearing down MODULE {0}".format(
        module.__name__))

def test_a_function():
    print("RUNNING TEST FUNCTION")

class BaseTest:
    def setup_class(cls):
        print("setting up CLASS {0}".format(
            cls.__name__))

    def teardown_class(cls):
        print("tearing down CLASS {0}\n".format(
            cls.__name__))

    def setup_method(self, method):
        print("setting up METHOD {0}".format(
            method.__name__))

    def teardown_method(self, method):
        print("tearing down  METHOD {0}".format(
            method.__name__))

class TestClass1(BaseTest):
    def test_method_1(self):
        print("RUNNING METHOD 1-1")

    def test_method_2(self):
```

```
            print("RUNNING METHOD 1-2")

    class TestClass2(BaseTest):
        def test_method_1(self):
            print("RUNNING METHOD 2-1")

        def test_method_2(self):
            print("RUNNING METHOD 2-2")
```

The sole purpose of the `BaseTest` class is to extract four methods that would be otherwise identical to the test classes, and use inheritance to reduce the amount of duplicate code. So, from the point of view of `py.test`, the two subclasses have not only two test methods each, but also two setup and two teardown methods (one at the class level, one at the method level).

If we run these tests using `py.test` with the `print` function output suppression disabled (by passing the `-s` or `--capture=no` flag), they show us when the various functions are called in relation to the tests themselves:

```
py.test setup_teardown.py -s
setup_teardown.py
setting up MODULE setup_teardown
RUNNING TEST FUNCTION
.setting up CLASS TestClass1
setting up METHOD test_method_1
RUNNING METHOD 1-1
.tearing down  METHOD test_method_1
setting up METHOD test_method_2
RUNNING METHOD 1-2
.tearing down  METHOD test_method_2
tearing down CLASS TestClass1
setting up CLASS TestClass2
setting up METHOD test_method_1
RUNNING METHOD 2-1
.tearing down  METHOD test_method_1
setting up METHOD test_method_2
RUNNING METHOD 2-2
.tearing down  METHOD test_method_2
tearing down CLASS TestClass2

tearing down MODULE setup_teardown
```

The setup and teardown methods for the module are executed at the beginning and end of the session. Then the lone module-level test function is run. Next, the setup method for the first class is executed, followed by the two tests for that class. These tests are each individually wrapped in separate `setup_method` and `teardown_method` calls. After the tests have executed, the class teardown method is called. The same sequence happens for the second class, before the `teardown_module` method is finally called, exactly once.

A completely different way to set up variables

One of the most common uses for the various setup and teardown functions is to ensure certain class or module variables are available with a known value before each test method is run.

`py.test` offers a completely different way to do this using what are known as **funcargs**, short for function arguments. Funcargs are basically named variables that are predefined in a test configuration file. This allows us to separate configuration from execution of tests, and allows the funcargs to be used across multiple classes and modules.

To use them, we add parameters to our test function. The names of the parameters are used to look up specific arguments in specially named functions. For example, if we wanted to test the `StatsList` class we used while demonstrating `unittest`, we would again want to repeatedly test a list of valid integers. But we can write our tests like so instead of using a setup method:

```
from stats import StatsList

def pytest_funcarg__valid_stats(request):
    return StatsList([1,2,2,3,3,4])


def test_mean(valid_stats):
    assert valid_stats.mean() == 2.5

def test_median(valid_stats):
    assert valid_stats.median() == 2.5
    valid_stats.append(4)
    assert valid_stats.median() == 3

def test_mode(valid_stats):
    assert valid_stats.mode() == [2,3]
    valid_stats.remove(2)
    assert valid_stats.mode() == [3]
```

Each of the three test methods accepts a parameter named `valid_stats`; this parameter is created by calling the `pytest_funcarg__valid_stats` function defined at the top of the file. It can also be defined in a file called `conftest.py` if the funcarg is needed by multiple modules. The `conftest.py` file is parsed by `py.test` to load any "global" test configuration; it is a sort of catch-all for customizing the `py.test` experience.

As with other `py.test` features, the name of the factory for returning a funcarg is important; funcargs are functions that are named `pytest_funcarg__<identifier>`, where `<identifier>` is a valid variable name that can be used as a parameter in a test function. This function accepts a mysterious `request` parameter, and returns the object to be passed as an argument into the individual test functions. The funcarg is created afresh for each call to an individual test function; this allows us, for example, to change the list in one test and know that it will be reset to its original values in the next test.

Funcargs can do a lot more than return basic variables. That `request` object passed into the funcarg factory provides some extremely useful methods and attributes to modify the funcarg's behavior. The `module`, `cls`, and `function` attributes allow us to see exactly which test is requesting the funcarg. The `config` attribute allows us to check command-line arguments and other configuration data.

More interestingly, the request object provides methods that allow us to do additional cleanup on the funcarg, or to reuse it across tests, activities that would otherwise be relegated to setup and teardown methods of a specific scope.

The `request.addfinalizer` method accepts a callback function that performs cleanup after each test function that uses the funcarg has been called. This provides the equivalent of a teardown method, allowing us to clean up files, close connections, empty lists, or reset queues. For example, the following code tests the `os.mkdir` functionality by creating a temporary directory `funcarg`:

```
import tempfile
import shutil
import os.path

def pytest_funcarg__temp_dir(request):
    dir = tempfile.mkdtemp()
    print(dir)

    def cleanup():
        shutil.rmtree(dir)
    request.addfinalizer(cleanup)
```

```
    return dir

def test_osfiles(temp_dir):
    os.mkdir(os.path.join(temp_dir, 'a'))
    os.mkdir(os.path.join(temp_dir, 'b'))
    dir_contents = os.listdir(temp_dir)
    assert len(dir_contents) == 2
    assert 'a' in dir_contents
    assert 'b' in dir_contents
```

The funcarg creates a new empty temporary directory for files to be created in. Then it adds a finalizer call to remove that directory (using `shutil.rmtree`, which recursively removes a directory and anything inside it) after the test has completed. The filesystem is then left in the same state in which it started.

We can use the `request.cached_setup` method to create function argument variables that last longer than one test. This is useful when setting up an expensive operation that can be reused by multiple tests as long as the resource reuse doesn't break the atomic or unit nature of the tests (so that one test does not rely on and is not impacted by a previous one). For example, if we were to test the following echo server, we may want to run only one instance of the server in a separate process, and then have multiple tests connect to that instance:

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind(('localhost',1028))
s.listen(1)

    while True:
        client, address = s.accept()
        data = client.recv(1024)
        client.send(data)
        client.close()
```

All this code does is listen on a specific port and wait for input from a client socket. When it receives input, it sends the same value back. To test this, we can start the server in a separate process and cache the result for use in multiple tests. Here's how the test code might look:

```
import subprocess
import socket
```

```
import time

def pytest_funcarg__echoserver(request):
    def setup():
        p = subprocess.Popen(
                ['python3', 'echo_server.py'])
        time.sleep(1)
        return p

    def cleanup(p):
        p.terminate()

    return request.cached_setup(
            setup=setup,
            teardown=cleanup,
            scope="session")

def pytest_funcarg__clientsocket(request):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(('localhost', 1028))
    request.addfinalizer(lambda: s.close())
    return s

def test_echo(echoserver, clientsocket):
    clientsocket.send(b"abc")
    assert clientsocket.recv(3) == b'abc'

def test_echo2(echoserver, clientsocket):
    clientsocket.send(b"def")
    assert clientsocket.recv(3) == b'def'
```

We've created two funcargs here. The first runs the echo server in a separate process, and returns the process object. The second instantiates a new socket object for each test, and closes it when the test has completed, using `addfinalizer`. The first funcarg is the one we're currently interested in. It looks much like a traditional unit test setup and teardown. We create a `setup` function that accepts no parameters and returns the correct argument; in this case, a process object that is actually ignored by the tests, since they only care that the server is running. Then, we create a `cleanup` function (the name of the function is arbitrary since it's just an object we pass into another function), which accepts a single argument: the argument returned by `setup`. This cleanup code terminates the process.

Instead of returning a funcarg directly, the parent function returns the results of a call to `request.cached_setup`. It accepts two arguments for the `setup` and `teardown` functions (which we just created), and a `scope` argument. This last argument should be one of the three strings "function", "module", or "session"; it determines just how long the argument will be cached. We set it to "session" in this example, so it is cached for the duration of the entire `py.test` run. The process will not be terminated or restarted until all tests have run. The "module" scope, of course, caches it only for tests in that module, and the "function" scope treats the object more like a normal funcarg, in that it is reset after each test function is run.

Skipping tests with py.test

As with the `unittest` module, it is frequently necessary to skip tests in `py.test`, for a variety of reasons: the code being tested hasn't been written yet, the test only runs on certain interpreters or operating systems, or the test is time consuming and should only be run under certain circumstances.

We can skip tests at any point in our code using the `py.test.skip` function. It accepts a single argument: a string describing why it has been skipped. This function can be called anywhere; if we call it inside a test function, the test will be skipped. If we call it at the module level, all the tests in that module will be skipped. If we call it inside a funcarg function, all tests that call that funcarg will be skipped.

Of course, in all these locations, it is often desirable to skip tests only if certain conditions are or are not met. Since we can execute the `skip` function at any place in Python code, we can execute it inside an `if` statement. So we may write a test that looks like this:

```
import sys
import py.test

def test_simple_skip():
    if sys.platform != "fakeos":
        py.test.skip("Test works only on fakeOS")

    fakeos.do_something_fake()
    assert fakeos.did_not_happen
```

That's some pretty silly code, really. There is no Python platform named `fakeos`, so this test will skip on all operating systems. It shows how we can skip conditionally, and since the `if` statement can check any valid conditional, we have a lot of power over when tests are skipped. Often, we check `sys.version_info` to check the Python interpreter version, `sys.platform` to check the operating system, or `some_library.__version__` to check whether we have a recent enough version of a given API.

Since skipping an individual test method or function based on a certain conditional is one of the most common uses of test skipping, `py.test` provides a convenience decorator that allows us to do this in one line. The decorator accepts a single string, which can contain any executable Python code that evaluates to a Boolean value. For example, the following test will only run on Python 3 or higher:

```
import py.test

@py.test.mark.skipif("sys.version_info <= (3,0)")
def test_python3():
    assert b"hello".decode() == "hello"
```

The `py.test.mark.xfail` decorator behaves similarly, except that it marks a test as expected to fail, similar to `unittest.expectedFailure()`. If the test is successful, it will be recorded as a failure; if it fails, it will be reported as expected behavior. In the case of `xfail`, the conditional argument is optional; if it is not supplied, the test will be marked as expected to fail under all conditions.

Imitating expensive objects

Sometimes, we want to test code that requires an object be supplied that is either expensive or difficult to construct. While this may mean your API needs rethinking to have a more testable interface (which typically means a more usable interface), we sometimes find ourselves writing test code that has a ton of boilerplate to set up objects that are only incidentally related to the code under test.

For example, imagine we have some code that keeps track of flight statuses in a key-value store (such as `redis` or `memcache`) such that we can store the timestamp and the most recent status. A basic version of such code might look like this:

```
import datetime
import redis

class FlightStatusTracker:
    ALLOWED_STATUSES = {'CANCELLED', 'DELAYED', 'ON TIME'}

    def __init__(self):
        self.redis = redis.StrictRedis()

    def change_status(self, flight, status):
```

```
        status = status.upper()
        if status not in self.ALLOWED_STATUSES:
            raise ValueError(
                   "{} is not a valid status".format(status))

        key = "flightno:{}".format(flight)
        value = "{}|{}".format(
            datetime.datetime.now().isoformat(), status)
        self.redis.set(key, value)
```

There are a lot of things we ought to test in that `change_status` method. We should check that it raises the appropriate error if a bad status is passed in. We need to ensure that it converts statuses to uppercase. We can see that the key and value have the correct formatting when the `set()` method is called on the `redis` object.

One thing we don't have to check in our unit tests, however, is that the `redis` object is properly storing the data. This is something that absolutely should be tested in integration or application testing, but at the unit test level, we can assume that the py-redis developers have tested their code and that this method does what we want it to. As a rule, unit tests should be self-contained and not rely on the existence of outside resources, such as a running Redis instance.

Instead, we only need to test that the `set()` method was called the appropriate number of times and with the appropriate arguments. We can use `Mock()` objects in our tests to replace the troublesome method with an object we can introspect. The following example illustrates the use of mock:

```
from unittest.mock import Mock
import py.test
def pytest_funcarg__tracker():
    return FlightStatusTracker()

def test_mock_method(tracker):
    tracker.redis.set = Mock()
    with py.test.raises(ValueError) as ex:
        tracker.change_status("AC101", "lost")
    assert ex.value.args[0] == "LOST is not a valid status"
    assert tracker.redis.set.call_count == 0
```

This test, written using `py.test` syntax, asserts that the correct exception is raised when an inappropriate argument is passed in. In addition, it creates a mock object for the `set` method and makes sure that it is never called. If it was, it would mean there was a bug in our exception handling code.

Simply replacing the method worked fine in this case, since the object being replaced was destroyed in the end. However, we often want to replace a function or method only for the duration of a test. For example, if we want to test the timestamp formatting in the mock method, we need to know exactly what `datetime.datetime.now()` is going to return. However, this value changes from run to run. We need some way to pin it to a specific value so we can test it deterministically.

Remember monkey-patching? Temporarily setting a library function to a specific value is an excellent use of it. The mock library provides a patch context manager that allows us to replace attributes on existing libraries with mock objects. When the context manager exits, the original attribute is automatically restored so as not to impact other test cases. Here's an example:

```
from unittest.mock import patch
def test_patch(tracker):
    tracker.redis.set = Mock()
    fake_now = datetime.datetime(2015, 4, 1)
    with patch('datetime.datetime') as dt:
        dt.now.return_value = fake_now
        tracker.change_status("AC102", "on time")
    dt.now.assert_called_once_with()
    tracker.redis.set.assert_called_once_with(
        "flightno:AC102",
        "2015-04-01T00:00:00|ON TIME")
```

In this example, we first construct a value called `fake_now`, which we will set as the return value of the `datetime.datetime.now` function. We have to construct this object before we patch `datetime.datetime` because otherwise we'd be calling the patched `now` function before we constructed it!

The `with` statement invites the patch to replace the `datetime.datetime` module with a mock object, which is returned as the value `dt`. The neat thing about mock objects is that any time you access an attribute or method on that object, it returns another mock object. Thus when we access `dt.now`, it gives us a new mock object. We set the `return_value` of that object to our `fake_now` object; that way, whenever the `datetime.datetime.now` function is called, it will return our object instead of a new mock object.

Then, after calling our `change_status` method with known values, we use the mock class's `assert_called_once_with` function to ensure that the `now` function was indeed called exactly once with no arguments. We then call it a second time to prove that the `redis.set` method was called with arguments that were formatted as we expected them to be.

The previous example is a good indication of how writing tests can guide our API design. The `FlightStatusTracker` object looks sensible at first glance; we construct a `redis` connection when the object is constructed, and we call into it when we need it. When we write tests for this code, however, we discover that even if we mock out that `self.redis` variable on a `FlightStatusTracker`, the `redis` connection still has to be constructed. This call actually fails if there is no Redis server running, and our tests also fail.

We could solve this problem by mocking out the `redis.StrictRedis` class to return a mock in a `setUp` method. A better idea, however, might be to rethink our example. Instead of constructing the `redis` instance inside`__init__`, perhaps we should allow the user to pass one in, as in the following example:

```
        def __init__(self, redis_instance=None):
            self.redis = redis_instance if redis_instance else redis.
    StrictRedis()
```

This allows us to pass a mock in when we are testing, so the `StrictRedis` method never gets constructed. However, it also allows any client code that talks to `FlightStatusTracker` to pass in their own `redis` instance. There are a variety of reasons they might want to do this. They may have already constructed one for other parts of their code. They may have created an optimized implementation of the `redis` API. Perhaps they have one that logs metrics to their internal monitoring systems. By writing a unit test, we've uncovered a use case that makes our API more flexible from the start, rather than waiting for clients to demand we support their exotic needs.

This has been a brief introduction to the wonders of mocking code. Mocks are part of the standard `unittest` library since Python 3.3, but as you see from these examples, they can also be used with `py.test` and other libraries. Mocks have other more advanced features that you may need to take advantage of as your code gets more complicated. For example, you can use the `spec` argument to invite a mock to imitate an existing class so that it raises an error if code tries to access an attribute that does not exist on the imitated class. You can also construct mock methods that return different arguments each time they are called by passing a list as the `side_effect` argument. The `side_effect` parameter is quite versatile; you can also use it to execute arbitrary functions when the mock is called or to raise an exception.

In general, we should be quite stingy with mocks. If we find ourselves mocking out multiple elements in a given unit test, we may end up testing the mock framework rather than our real code. This serves no useful purpose whatsoever; after all, mocks are well-tested already! If our code is doing a lot of this, it's probably another sign that the API we are testing is poorly designed. Mocks should exist at the boundaries between the code under test and the libraries they interface with. If this isn't happening, we may need to change the API so that the boundaries are redrawn in a different place.

How much testing is enough?

We've already established that untested code is broken code. But how can we tell how well our code is tested? How do we know how much of our code is actually being tested and how much is broken? The first question is the more important one, but it's hard to answer. Even if we know we have tested every line of code in our application, we do not know that we have tested it properly. For example, if we write a stats test that only checks what happens when we provide a list of integers, it may still fail spectacularly if used on a list of floats or strings or self-made objects. The onus of designing complete test suites still lies with the programmer.

The second question—how much of our code is actually being tested—is easy to verify. Code coverage is essentially an estimate of the number of lines of code that are executed by a program. If we know that number and the number of lines that are in the program, we can get an estimate of what percentage of the code was really tested, or covered. If we additionally have an indicator as to which lines were not tested, we can more easily write new tests to ensure those lines are less broken.

The most popular tool for testing code coverage is called, memorably enough, `coverage.py`. It can be installed like most other third-party libraries using the command `pip install coverage`.

We don't have space to cover all the details of the coverage API, so we'll just look at a few typical examples. If we have a Python script that runs all our unit tests for us (for example, using `unittest.main`, a custom test runner or `discover`), we can use the following command to perform a coverage analysis:

```
coverage run coverage_unittest.py
```

This command will exit normally, but it creates a file named `.coverage` that holds the data from the run. We can now use the `coverage report` command to get an analysis of code coverage:

```
>>> coverage report
```

The output is as follows:

```
Name                              Stmts   Exec  Cover
-------------------------------------------------
coverage_unittest                     7      7   100%
stats                                19      6    31%
-------------------------------------------------
TOTAL                                26     13    50%
```

This basic report lists the files that were executed (our unit test and a module it imported). The number of lines of code in each file, and the number that were executed by the test are also listed. The two numbers are then combined to estimate the amount of code coverage. If we pass the `-m` option to the report command, it will additionally add a column that looks like this:

```
Missing
-----------
8-12, 15-23
```

The ranges of lines listed here identify lines in the `stats` module that were not executed during the test run.

The example we just ran the code coverage tool on uses the same stats module we created earlier in the chapter. However, it deliberately uses a single test that fails to test a lot of code in the file. Here's the test:

```
from stats import StatsList
import unittest

class TestMean(unittest.TestCase):
    def test_mean(self):
        self.assertEqual(StatsList([1,2,2,3,3,4]).mean(), 2.5)

if __name__ == "__main__":

    unittest.main()
```

This code doesn't test the median or mode functions, which correspond to the line numbers that the coverage output told us were missing.

The textual report is sufficient, but if we use the command `coverage html`, we can get an even fancier interactive HTML report that we can view in a web browser. The web page even highlights which lines in the source code were and were not tested. Here's how it looks:

We can use the `coverage.py` module with `py.test` as well. We'll need to install the `py.test` plugin for code coverage, using `pip install pytest-coverage`. The plugin adds several command-line options to `py.test`, the most useful being `--cover-report`, which can be set to `html`, `report`, or `annotate` (the latter actually modifies the source code to highlight any lines that were not covered).

Unfortunately, if we could somehow run a coverage report on this section of the chapter, we'd find that we have not covered most of what there is to know about code coverage! It is possible to use the coverage API to manage code coverage from within our own programs (or test suites), and `coverage.py` accepts numerous configuration options that we haven't touched on. We also haven't discussed the difference between statement coverage and branch coverage (the latter is much more useful, and the default in recent versions of `coverage.py`) or other styles of code coverage.

Bear in mind that while 100 percent code coverage is a lofty goal that we should all strive for, 100 percent coverage is not enough! Just because a statement was tested does not mean that it was tested properly for all possible inputs.

Case study

Let's walk through test-driven development by writing a small, tested, cryptography application. Don't worry, you won't need to understand the mathematics behind complicated modern encryption algorithms such as Threefish or RSA. Instead, we'll be implementing a sixteenth-century algorithm known as the Vigenère cipher. The application simply needs to be able to encode and decode a message, given an encoding keyword, using this cipher.

First, we need to understand how the cipher works if we apply it manually (without a computer). We start with a table like this:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
B C D E F G H I J K L M N O P Q R S T U V W X Y Z A
C D E F G H I J K L M N O P Q R S T U V W X Y Z A B
D E F G H I J K L M N O P Q R S T U V W X Y Z A B C
E F G H I J K L M N O P Q R S T U V W X Y Z A B C D
F G H I J K L M N O P Q R S T U V W X Y Z A B C D E
G H I J K L M N O P Q R S T U V W X Y Z A B C D E F
H I J K L M N O P Q R S T U V W X Y Z A B C D E F G
I J K L M N O P Q R S T U V W X Y Z A B C D E F G H
J K L M N O P Q R S T U V W X Y Z A B C D E F G H I
K L M N O P Q R S T U V W X Y Z A B C D E F G H I J
L M N O P Q R S T U V W X Y Z A B C D E F G H I J K
M N O P Q R S T U V W X Y Z A B C D E F G H I J K L
N O P Q R S T U V W X Y Z A B C D E F G H I J K L M
O P Q R S T U V W X Y Z A B C D E F G H I J K L M N
P Q R S T U V W X Y Z A B C D E F G H I J K L M N O
Q R S T U V W X Y Z A B C D E F G H I J K L M N O P
R S T U V W X Y Z A B C D E F G H I J K L M N O P Q
S T U V W X Y Z A B C D E F G H I J K L M N O P Q R
T U V W X Y Z A B C D E F G H I J K L M N O P Q R S
U V W X Y Z A B C D E F G H I J K L M N O P Q R S T
V W X Y Z A B C D E F G H I J K L M N O P Q R S T U
W X Y Z A B C D E F G H I J K L M N O P Q R S T U V
X Y Z A B C D E F G H I J K L M N O P Q R S T U V W
Y Z A B C D E F G H I J K L M N O P Q R S T U V W X
Z A B C D E F G H I J K L M N O P Q R S T U V W X Y
```

Given a keyword, TRAIN, we can encode the message ENCODED IN PYTHON as follows:

1. Repeat the keyword and message together such that it is easy to map letters from one to the other:

   ```
   E N C O D E D I N P Y T H O N   T R A I N T R A I N T R A I N
   ```

2. For each letter in the plain text, find the row that begins with that letter in the table.
3. Find the column with the letter associated with the keyword letter for the chosen plaintext letter.
4. The encoded character is at the intersection of this row and column.

For example, the row starting with E intersects the column starting with T at the character X. So, the first letter in the ciphertext is X. The row starting with N intersects the column starting with R at the character E, leading to the ciphertext XE. C intersects A at C, and O intersects I at W. D and N map to Q while E and T map to X. The full encoded message is XECWQXUIVCRKHWA.

Decoding basically follows the opposite procedure. First, find the row with the character for the shared keyword (the T row), then find the location in that row where the encoded character (the X) is located. The plaintext character is at the top of the column for that row (the E).

Implementing it

Our program will need an `encode` method that takes a keyword and plaintext and returns the ciphertext, and a `decode` method that accepts a keyword and ciphertext and returns the original message.

But rather than just writing those methods, let's follow a test-driven development strategy. We'll be using `py.test` for our unit testing. We need an `encode` method, and we know what it has to do; let's write a test for that method first:

```
def test_encode():
    cipher = VigenereCipher("TRAIN")
    encoded = cipher.encode("ENCODEDINPYTHON")
    assert encoded == "XECWQXUIVCRKHWA"
```

This test fails, naturally, because we aren't importing a `VigenereCipher` class anywhere. Let's create a new module to hold that class.

Let's start with the following `VigenereCipher` class:

```
class VigenereCipher:
    def __init__(self, keyword):
        self.keyword = keyword

    def encode(self, plaintext):
        return "XECWQXUIVCRKHWA"
```

If we add a `from vigenere_cipher import VigenereCipher` line to the top of our test class and run `py.test`, the preceding test will pass! We've finished our first test-driven development cycle.

Obviously, returning a hardcoded string is not the most sensible implementation of a cipher class, so let's add a second test:

```
def test_encode_character():
    cipher = VigenereCipher("TRAIN")
    encoded = cipher.encode("E")
    assert encoded == "X"
```

Ah, now that test will fail. It looks like we're going to have to work harder. But I just thought of something: what if someone tries to encode a string with spaces or lowercase characters? Before we start implementing the encoding, let's add some tests for these cases, so we don't we forget them. The expected behavior will be to remove spaces, and to convert lowercase letters to capitals:

```
def test_encode_spaces():
    cipher = VigenereCipher("TRAIN")
    encoded = cipher.encode("ENCODED IN PYTHON")
    assert encoded == "XECWQXUIVCRKHWA"

def test_encode_lowercase():
    cipher = VigenereCipher("TRain")
    encoded = cipher.encode("encoded in Python")
    assert encoded == "XECWQXUIVCRKHWA"
```

If we run the new test suite, we find that the new tests pass (they expect the same hardcoded string). But they ought to fail later if we forget to account for these cases.

Now that we have some test cases, let's think about how to implement our encoding algorithm. Writing code to use a table like we used in the earlier manual algorithm is possible, but seems complicated, considering that each row is just an alphabet rotated by an offset number of characters. It turns out (I asked Wikipedia) that we can use modulo arithmetic to combine the characters instead of doing a table lookup. Given plaintext and keyword characters, if we convert the two letters to their numerical values (with A being 0 and Z being 25), add them together, and take the remainder mod 26, we get the ciphertext character! This is a straightforward calculation, but since it happens on a character-by-character basis, we should probably put it in its own function. And before we do that, we should write a test for the new function:

```
from vigenere_cipher import combine_character
def test_combine_character():
    assert combine_character("E", "T") == "X"
    assert combine_character("N", "R") == "E"
```

Now we can write the code to make this function work. In all honesty, I had to run the test several times before I got this function completely correct; first I returned an integer, and then I forgot to shift the character back up to the normal ASCII scale from the zero-based scale. Having the test available made it easy to test and debug these errors. This is another bonus of test-driven development.

```
def combine_character(plain, keyword):
    plain = plain.upper()
    keyword = keyword.upper()
    plain_num = ord(plain) - ord('A')
    keyword_num = ord(keyword) - ord('A')
    return chr(ord('A') + (plain_num + keyword_num) % 26)
```

Now that `combine_characters` is tested, I thought we'd be ready to implement our `encode` function. However, the first thing we want inside that function is a repeating version of the keyword string that is as long as the plaintext. Let's implement a function for that first. Oops, I mean let's implement the test first!

```
def test_extend_keyword():
    cipher = VigenereCipher("TRAIN")
    extended = cipher.extend_keyword(16)
    assert extended == "TRAINTRAINTRAINT"
```

Before writing this test, I expected to write `extend_keyword` as a standalone function that accepted a keyword and an integer. But as I started drafting the test, I realized it made more sense to use it as a helper method on the `VigenereCipher` class. This shows how test-driven development can help design more sensible APIs. Here's the method implementation:

```
def extend_keyword(self, number):
    repeats = number // len(self.keyword) + 1
    return (self.keyword * repeats)[:number]
```

Once again, this took a few runs of the test to get right. I ended up adding a second versions of the test, one with fifteen and one with sixteen letters, to make sure it works if the integer division has an even number.

Now we're finally ready to write our `encode` method:

```
def encode(self, plaintext):
    cipher = []
    keyword = self.extend_keyword(len(plaintext))
    for p,k in zip(plaintext, keyword):
        cipher.append(combine_character(p,k))
    return "".join(cipher)
```

That looks correct. Our test suite should pass now, right?

Actually, if we run it, we'll find that two tests are still failing. We totally forgot about the spaces and lowercase characters! It is a good thing we wrote those tests to remind us. We'll have to add this line at the beginning of the method:

```
plaintext = plaintext.replace(" ", "").upper()
```

If we have an idea about a corner case in the middle of implementing something, we can create a test describing that idea. We don't even have to implement the test; we can just run `assert False` to remind us to implement it later. The failing test will never let us forget the corner case and it can't be ignored like filing a task can. If it takes a while to get around to fixing the implementation, we can mark the test as an expected failure.

Now all the tests pass successfully. This chapter is pretty long, so we'll condense the examples for decoding. Here are a couple tests:

```
def test_separate_character():
    assert separate_character("X", "T") == "E"
```

```
        assert separate_character("E", "R") == "N"

    def test_decode():
        cipher = VigenereCipher("TRAIN")
        decoded = cipher.decode("XECWQXUIVCRKHWA")
        assert decoded == "ENCODEDINPYTHON"
```

Here's the `separate_character` function:

```
    def separate_character(cypher, keyword):
        cypher = cypher.upper()
        keyword = keyword.upper()
        cypher_num = ord(cypher) - ord('A')
        keyword_num = ord(keyword) - ord('A')
        return chr(ord('A') + (cypher_num - keyword_num) % 26)
```

And the `decode` method:

```
        def decode(self, ciphertext):
            plain = []
            keyword = self.extend_keyword(len(ciphertext))
            for p,k in zip(ciphertext, keyword):
                plain.append(separate_character(p,k))
            return "".join(plain)
```

These methods have a lot of similarity to those used for encoding. The great thing about having all these tests written and passing is that we can now go back and modify our code, knowing it is still safely passing the tests. For example, if we replace our existing `encode` and `decode` methods with these refactored methods, our tests still pass:

```
        def _code(self, text, combine_func):
            text = text.replace(" ", "").upper()
            combined = []
            keyword = self.extend_keyword(len(text))
            for p,k in zip(text, keyword):
                combined.append(combine_func(p,k))
            return "".join(combined)

        def encode(self, plaintext):
            return self._code(plaintext, combine_character)

        def decode(self, ciphertext):
            return self._code(ciphertext, separate_character)
```

This is the final benefit of test-driven development, and the most important. Once the tests are written, we can improve our code as much as we like and be confident that our changes didn't break anything we have been testing for. Furthermore, we know exactly when our refactor is finished: when the tests all pass.

Of course, our tests may not comprehensively test everything we need them to; maintenance or code refactoring can still cause undiagnosed bugs that don't show up in testing. Automated tests are not foolproof. If bugs do occur, however, it is still possible to follow a test-driven plan; step one is to write a test (or multiple tests) that duplicates or "proves" that the bug in question is occurring. This will, of course, fail. Then write the code to make the tests stop failing. If the tests were comprehensive, the bug will be fixed, and we will know if it ever happens again, as soon as we run the test suite.

Finally, we can try to determine how well our tests operate on this code. With the `py.test` coverage plugin installed, `py.test --coverage-report=report` tells us that our test suite has 100 percent code coverage. This is a great statistic, but we shouldn't get too cocky about it. Our code hasn't been tested when encoding messages that have numbers, and its behavior with such inputs is thus undefined.

Exercises

Practice test-driven development. That is your first exercise. It's easier to do this if you're starting a new project, but if you have existing code you need to work on, you can start by writing tests for each new feature you implement. This can become frustrating as you become more enamored with automated tests. The old, untested code will start to feel rigid and tightly coupled, and will become uncomfortable to maintain; you'll start feeling like changes you make are breaking the code and you have no way of knowing, for lack of tests. But if you start small, adding tests will improve, the codebase improves over time.

So to get your feet wet with test-driven development, start a fresh project. Once you've started to appreciate the benefits (you will) and realize that the time spent writing tests is quickly regained in terms of more maintainable code, you'll want to start writing tests for existing code. This is when you should start doing it, not before. Writing tests for code that we "know" works is boring. It is hard to get interested in the project until you realize just how broken the code we thought was working really is.

Try writing the same set of tests using both the built-in `unittest` module and `py.test`. Which do you prefer? `unittest` is more similar to test frameworks in other languages, while `py.test` is arguably more Pythonic. Both allow us to write object-oriented tests and to test object-oriented programs with ease.

We used `py.test` in our case study, but we didn't touch on any features that wouldn't have been easily testable using `unittest`. Try adapting the tests to use test skipping or funcargs. Try the various setup and teardown methods, and compare their use to funcargs. Which feels more natural to you?

In our case study, we have a lot of tests that use a similar `VigenereCipher` object; try reworking this code to use a funcarg. How many lines of code does it save?

Try running a coverage report on the tests you've written. Did you miss testing any lines of code? Even if you have 100 percent coverage, have you tested all the possible inputs? If you're doing test-driven development, 100 percent coverage should follow quite naturally, as you will write a test before the code that satisfies that test. However, if writing tests for existing code, it is more likely that there will be edge conditions that go untested.

Think carefully about the values that are somehow different: empty lists when you expect full ones, zero or one or infinity compared to intermediate integers, floats that don't round to an exact decimal place, strings when you expected numerals, or the ubiquitous `None` value when you expected something meaningful. If your tests cover such edge cases, your code will be in good shape.

Summary

We have finally covered the most important topic in Python programming: automated testing. Test-driven development is considered a best practice. The standard library `unittest` module provides a great out-of-the-box solution for testing, while the `py.test` framework has some more Pythonic syntaxes. Mocks can be used to emulate complex classes in our tests. Code coverage gives us an estimate of how much of our code is being run by our tests, but it does not tell us that we have tested the right things.

In the next chapter, we'll jump into a completely different topic: concurrency.

13
Concurrency

Concurrency is the art of making a computer do (or appear to do) multiple things at once. Historically, this meant inviting the processor to switch between different tasks many times per second. In modern systems, it can also literally mean doing two or more things simultaneously on separate processor cores.

Concurrency is not inherently an object-oriented topic, but Python's concurrent systems are built on top of the object-oriented constructs we've covered throughout the book. This chapter will introduce you to the following topics:

- Threads
- Multiprocessing
- Futures
- AsyncIO

Concurrency is complicated. The basic concepts are fairly simple, but the bugs that can occur are notoriously difficult to track down. However, for many projects, concurrency is the only way to get the performance we need. Imagine if a web server couldn't respond to a user's request until the previous one was completed! We won't be going into all the details of just how hard it is (another full book would be required) but we'll see how to do basic concurrency in Python, and some of the most common pitfalls to avoid.

Threads

Most often, concurrency is created so that work can continue happening while the program is waiting for I/O to happen. For example, a server can start processing a new network request while it waits for data from a previous request to arrive. An interactive program might render an animation or perform a calculation while waiting for the user to press a key. Bear in mind that while a person can type more than 500 characters per minute, a computer can perform billions of instructions per second. Thus, a ton of processing can happen between individual key presses, even when typing quickly.

It's theoretically possible to manage all this switching between activities within your program, but it would be virtually impossible to get right. Instead, we can rely on Python and the operating system to take care of the tricky switching part, while we create objects that appear to be running independently, but simultaneously. These objects are called **threads**; in Python they have a very simple API. Let's take a look at a basic example:

```
from threading import Thread

class InputReader(Thread):
    def run(self):
        self.line_of_text = input()

print("Enter some text and press enter: ")
thread = InputReader()
thread.start()

count = result = 1
while thread.is_alive():
    result = count * count
    count += 1

print("calculated squares up to {0} * {0} = {1}".format(
    count, result))
print("while you typed '{}'".format(thread.line_of_text))
```

This example runs two threads. Can you see them? Every program has one thread, called the main thread. The code that executes from the beginning is happening in this thread. The second thread, more obviously, exists as the `InputReader` class.

To construct a thread, we must extend the `Thread` class and implement the `run` method. Any code inside the `run` method (or that is called from within that method) is executed in a separate thread.

The new thread doesn't start running until we call the `start()` method on the object. In this case, the thread immediately pauses to wait for input from the keyboard. In the meantime, the original thread continues executing at the point `start` was called. It starts calculating squares inside a `while` loop. The condition in the `while` loop checks if the `InputReader` thread has exited its `run` method yet; once it does, it outputs some summary information to the screen.

If we run the example and type the string "hello world", the output looks as follows:

```
Enter some text and press enter:
hello world
calculated squares up to 1044477 * 1044477 = 1090930114576
while you typed 'hello world'
```

You will, of course, calculate more or less squares while typing the string as the numbers are related to both our relative typing speeds, and to the processor speeds of the computers we are running.

A thread only starts running in concurrent mode when we call the `start` method. If we want to take out the concurrent call to see how it compares, we can call `thread.run()` in the place that we originally called `thread.start()`. The output is telling:

```
Enter some text and press enter:
hello world
calculated squares up to 1 * 1 = 1
while you typed 'hello world'
```

In this case, the thread never becomes alive and the `while` loop never executes. We wasted a lot of CPU power sitting idle while we were typing.

There are a lot of different patterns for using threads effectively. We won't be covering all of them, but we will look at a common one so we can learn about the `join` method. Let's check the current temperature in the capital city of every province in Canada:

```
from threading import Thread
import json
from urllib.request import urlopen
import time

CITIES = [
    'Edmonton', 'Victoria', 'Winnipeg', 'Fredericton',
    "St. John's", 'Halifax', 'Toronto', 'Charlottetown',
```

```
        'Quebec City', 'Regina'
    ]

    class TempGetter(Thread):
        def __init__(self, city):
            super().__init__()
            self.city = city

        def run(self):
            url_template = (
                'http://api.openweathermap.org/data/2.5/'
                'weather?q={},CA&units=metric')
            response = urlopen(url_template.format(self.city))
            data = json.loads(response.read().decode())
            self.temperature = data['main']['temp']

    threads = [TempGetter(c) for c in CITIES]
    start = time.time()
    for thread in threads:
        thread.start()

    for thread in threads:
        thread.join()

    for thread in threads:
        print(
            "it is {0.temperature:.0f}°C in {0.city}".format(thread))
    print(
        "Got {} temps in {} seconds".format(
        len(threads), time.time() - start))
```

This code constructs 10 threads before starting them. Notice how we can override the constructor to pass them into the `Thread` object, remembering to call `super` to ensure the `Thread` is properly initialized. Pay attention to this: the new thread isn't running yet, so the `__init__` method is still executing from inside the main thread. Data we construct in one thread is accessible from other running threads.

After the 10 threads have been started, we loop over them again, calling the `join()` method on each. This method essentially says "wait for the thread to complete before doing anything". We call this ten times in sequence; the for loop won't exit until all ten threads have completed.

At this point, we can print the temperature that was stored on each thread object. Notice once again that we can access data that was constructed within the thread from the main thread. In threads, all state is shared by default.

Executing this code on my 100 mbit connection takes about two tenths of a second:

```
it is 5°C in Edmonton
it is 11°C in Victoria
it is 0°C in Winnipeg
it is -10°C in Fredericton
it is -12°C in St. John's
it is -8°C in Halifax
it is -6°C in Toronto
it is -13°C in Charlottetown
it is -12°C in Quebec City
it is 2°C in Regina
    Got 10 temps in 0.18970298767089844 seconds
```

If we run this code in a single thread (by changing the `start()` call to `run()` and commenting out the `join()` call), it takes closer to 2 seconds because each 0.2 second request has to complete before the next one begins. This speedup of 10 times shows just how useful concurrent programming can be.

The many problems with threads

Threads can be useful, especially in other programming languages, but modern Python programmers tend to avoid them for several reasons. As we'll see, there are other ways to do concurrent programming that are receiving more attention from the Python developers. Let's discuss some of these pitfalls before moving on to more salient topics.

Shared memory

The main problem with threads is also their primary advantage. Threads have access to all the memory and thus all the variables in the program. This can too easily cause inconsistencies in the program state. Have you ever encountered a room where a single light has two switches and two different people turn them on at the same time? Each person (thread) expects their action to turn the lamp (a variable) on, but the resulting value (the lamp is off) is inconsistent with those expectations. Now imagine if those two threads were transferring funds between bank accounts or managing the cruise control in a vehicle.

The solution to this problem in threaded programming is to "synchronize" access to any code that reads or writes a shared variable. There are a few different ways to do this, but we won't go into them here so we can focus on more Pythonic constructs. The synchronization solution works, but it is way too easy to forget to apply it. Worse, bugs due to inappropriate use of synchronization are really hard to track down because the order in which threads perform operations is inconsistent. We can't easily reproduce the error. Usually, it is safest to force communication between threads to happen using a lightweight data structure that already uses locks appropriately. Python offers the `queue.Queue` class to do this; it's functionality is basically the same as the `multiprocessing.Queue` that we will discuss in the next section.

In some cases, these disadvantages might be outweighed by the one advantage of allowing shared memory: it's fast. If multiple threads need access to a huge data structure, shared memory can provide that access quickly. However, this advantage is usually nullified by the fact that, in Python, it is impossible for two threads running on different CPU cores to be performing calculations at exactly the same time. This brings us to our second problem with threads.

The global interpreter lock

In order to efficiently manage memory, garbage collection, and calls to machine code in libraries, Python has a utility called the **global interpreter lock**, or **GIL**. It's impossible to turn off, and it means that threads are useless in Python for one thing that they excel at in other languages: parallel processing. The GIL's primary effect, for our purposes is to prevent any two threads from doing work at the exact same time, even if they have work to do. In this case, "doing work" means using the CPU, so it's perfectly ok for multiple threads to access the disk or network; the GIL is released as soon as the thread starts to wait for something.

The GIL is quite highly disparaged, mostly by people who don't understand what it is or all the benefits it brings to Python. It would definitely be nice if our language didn't have this restriction, but the Python reference developers have determined that, for now at least, it brings more value than it costs. It makes the reference implementation easier to maintain and develop, and during the single-core processor days when Python was originally developed, it actually made the interpreter faster. The net result of the GIL, however, is that it limits the benefits that threads bring us, without alleviating the costs.

While the GIL is a problem in the reference implementation of Python that most people use, it has been solved in some of the nonstandard implementations such as IronPython and Jython. Unfortunately, at the time of publication, none of these support Python 3.

Thread overhead

One final limitation of threads as compared to the asynchronous system we will be discussing later is the cost of maintaining the thread. Each thread takes up a certain amount of memory (both in the Python process and the operating system kernel) to record the state of that thread. Switching between the threads also uses a (small) amount of CPU time. This work happens seamlessly without any extra coding (we just have to call `start()` and the rest is taken care of), but the work still has to happen somewhere.

This can be alleviated somewhat by structuring our workload so that threads can be reused to perform multiple jobs. Python provides a `ThreadPool` feature to handle this. It is shipped as part of the multiprocessing library and behaves identically to the `ProcessPool`, that we will discuss shortly, so let's defer discussion until the next section.

Multiprocessing

The multiprocessing API was originally designed to mimic the thread API. However, it has evolved and in recent versions of Python 3, it supports more features more robustly. The multiprocessing library is designed when CPU-intensive jobs need to happen in parallel and multiple cores are available (given that a four core Raspberry Pi can currently be purchased for $35, there are usually multiple cores available). Multiprocessing is not useful when the processes spend a majority of their time waiting on I/O (for example, network, disk, database, or keyboard), but they are the way to go for parallel computation.

The multiprocessing module spins up new operating system processes to do the work. On Windows machines, this is a relatively expensive operation; on Linux, processes are implemented in the kernel the same way threads are, so the overhead is limited to the cost of running separate Python interpreters in each process.

Let's try to parallelize a compute-heavy operation using similar constructs to those provided by the `threading` API:

```
from multiprocessing import Process, cpu_count
import time
import os

class MuchCPU(Process):
    def run(self):
        print(os.getpid())
        for i in range(200000000):
            pass

if __name__ == '__main__':
    procs =  [MuchCPU() for f in range(cpu_count())]
    t = time.time()
    for p in procs:
        p.start()
    for p in procs:
        p.join()
    print('work took {} seconds'.format(time.time() - t))
```

This example just ties up the CPU for 200 million iterations. You may not consider this to be useful work, but it's a cold day and I appreciate the heat my laptop generates under such load.

The API should be familiar; we implement a subclass of `Process` (instead of `Thread`) and implement a `run` method. This method prints out the process ID (a unique number the operating system assigns to each process on the machine) before doing some intense (if misguided) work.

Pay special attention to the `if __name__ == '__main__':` guard around the module level code that prevents it to run if the module is being imported, rather than run as a program. This is good practice in general, but when using multiprocessing on some operating systems, it is essential. Behind the scenes, multiprocessing may have to import the module inside the new process in order to execute the `run()` method. If we allowed the entire module to execute at that point, it would start creating new processes recursively until the operating system ran out of resources.

We construct one process for each processor core on our machine, then start and join each of those processes. On my 2014 era quad-core laptop, the output looks like this:

```
6987
6988
```

```
6989
6990
work took 12.96659541130066 seconds
```

The first four lines are the process ID that was printed inside each `MuchCPU` instance. The last line shows that the 200 million iterations can run in about 13 seconds on my machine. During that 13 seconds, my process monitor indicated that all four of my cores were running at 100 percent.

If we subclass `threading.Thread` instead of `multiprocessing.Process` in `MuchCPU`, the output looks like this:

```
7235
7235
7235
7235
work took 28.577413082122803 seconds
```

This time, the four threads are running inside the same process and take close to three times as long to run. This is the cost of the global interpreter lock; in other languages or implementations of Python, the threaded version would run at least as fast as the multiprocessing version, We might expect it to be four times as long, but remember that many other programs are running on my laptop. In the multiprocessing version, these programs also need a share of the four CPUs. In the threading version, those programs can use the other three CPUs instead.

Multiprocessing pools

In general, there is no reason to have more processes than there are processors on the computer. There are a few reasons for this:

- Only `cpu_count()` processes can run simultaneously
- Each process consumes resources with a full copy of the Python interpreter
- Communication between processes is expensive
- Creating processes takes a nonzero amount of time

Given these constraints, it makes sense to create at most `cpu_count()` processes when the program starts and then have them execute tasks as needed. It is not difficult to implement a basic series of communicating processes that does this, but it can be tricky to debug, test, and get right. Of course, Python being Python, we don't have to do all this work because the Python developers have already done it for us in the form of multiprocessing pools.

The primary advantage of pools is that they abstract away the overhead of figuring out what code is executing in the main process and which code is running in the subprocess. As with the threading API that multiprocessing mimics, it can often be hard to remember who is executing what. The pool abstraction restricts the number of places that code in different processes interact with each other, making it much easier to keep track of.

- Pools also seamlessly hide the process of passing data between processes. Using a pool looks much like a function call; you pass data into a function, it is executed in another process or processes, and when the work is done, a value is returned. It is important to understand that under the hood, a lot of work is being done to support this: objects in one process are being pickled and passed into a pipe.
- Another process retrieves data from the pipe and unpickles it. Work is done in the subprocess and a result is produced. The result is pickled and passed into a pipe. Eventually, the original process unpickles it and returns it.

All this pickling and passing data into pipes takes time and memory. Therefore, it is ideal to keep the amount and size of data passed into and returned from the pool to a minimum, and it is only advantageous to use the pool if a lot of processing has to be done on the data in question.

Armed with this knowledge, the code to make all this machinery work is surprisingly simple. Let's look at the problem of calculating all the prime factors of a list of random numbers. This is a common and expensive part of a variety of cryptography algorithms (not to mention attacks on those algorithms!). It requires years of processing power to crack the extremely large numbers used to secure your bank accounts. The following implementation, while readable, is not at all efficient, but that's ok because we want to see it using lots of CPU time:

```
import random
from multiprocessing.pool import Pool

def prime_factor(value):
    factors = []
    for divisor in range(2, value-1):
        quotient, remainder = divmod(value, divisor)
        if not remainder:
            factors.extend(prime_factor(divisor))
            factors.extend(prime_factor(quotient))
            break
```

```
        else:
            factors = [value]
        return factors

    if __name__ == '__main__':
        pool = Pool()

        to_factor = [
            random.randint(100000, 50000000) for i in range(20)
        ]
        results = pool.map(prime_factor, to_factor)
        for value, factors in zip(to_factor, results):
            print("The factors of {} are {}".format(value, factors))
```

Let's focus on the parallel processing aspects as the brute force recursive algorithm for calculating factors is pretty clear. We first construct a multiprocessing pool instance. By default, this pool creates a separate process for each of the CPU cores in the machine it is running on.

The `map` method accepts a function and an iterable. The pool pickles each of the values in the iterable and passes it into an available process, which executes the function on it. When that process is finished doing it's work, it pickles the resulting list of factors and passes it back to the pool. Once all the pools are finished processing work (which could take some time), the results list is passed back to the original process, which has been waiting patiently for all this work to complete.

It is often more useful to use the similar `map_async` method, which returns immediately even though the processes are still working. In that case, the results variable would not be a list of values, but a promise to return a list of values later by calling `results.get()`. This promise object also has methods like `ready()`, and `wait()`, which allow us to check whether all the results are in yet.

Alternatively, if we don't know all the values we want to get results for in advance, we can use the `apply_async` method to queue up a single job. If the pool has a process that isn't already working, it will start immediately; otherwise, it will hold onto the task until there is a free process available.

Pools can also be `close`d, which refuses to take any further tasks, but processes everything currently in the queue, or `terminate`d, which goes one step further and refuses to start any jobs still on the queue, although any jobs currently running are still permitted to complete.

Queues

If we need more control over communication between processes, we can use a `Queue`. `Queue` data structures are useful for sending messages from one process into one or more other processes. Any picklable object can be sent into a `Queue`, but remember that pickling can be a costly operation, so keep such objects small. To illustrate queues, let's build a little search engine for text content that stores all relevant entries in memory.

This is not the most sensible way to build a text-based search engine, but I have used this pattern to query numerical data that needed to use CPU-intensive processes to construct a chart that was then rendered to the user.

This particular search engine scans all files in the current directory in parallel. A process is constructed for each core on the CPU. Each of these is instructed to load some of the files into memory. Let's look at the function that does the loading and searching:

```
def search(paths, query_q, results_q):
    lines = []
    for path in paths:
        lines.extend(l.strip() for l in path.open())

    query = query_q.get()
    while query:
        results_q.put([l for l in lines if query in l])
        query = query_q.get()
```

Remember, this function is run in a different process (in fact, it is run in `cpucount()` different processes) from the main thread. It is passes a list of `path.path` objects and two `multiprocessing.Queue` objects; one for incoming queries and one to send outgoing results. These queues have a similar interface to the `Queue` class we discussed in *Chapter 6*, *Python Data Structures*. However, they are doing extra work to pickle the data in the queue and pass it into the subprocess over a pipe. These two queues are set up in the main process and passed through the pipes into the search function inside the child processes.

The search code is pretty dumb, both in terms of efficiency and of capabilities; it loops over every line stored in memory and puts the matching ones in a list. The list is placed on a queue and passed back to the main process.

Let's look at the main process, which sets up these queues:

```
if __name__ == '__main__':
    from multiprocessing import Process, Queue, cpu_count
    from path import path
    cpus = cpu_count()
```

```
pathnames = [f for f in path('.').listdir() if f.isfile()]
paths = [pathnames[i::cpus] for i in range(cpus)]
query_queues = [Queue() for p in range(cpus)]
results_queue = Queue()

search_procs = [
    Process(target=search, args=(p, q, results_queue))
    for p, q in zip(paths, query_queues)
]
for proc in search_procs: proc.start()
```

For easier description, let's assume `cpu_count` is four. Notice how the import statements are placed inside the `if` guard? This is a small optimization that prevents them from being imported in each subprocess (where they aren't needed) on certain operating systems. We list all the paths in the current directory and then split the list into four approximately equal parts. We also construct a list of four `Queue` objects to send data into each subprocess. Finally, we construct a `single` results queue; this is passed into all four of the subprocesses. Each of them can put data into the queue and it will be aggregated in the main process.

Now let's look at the code that makes a search actually happen:

```
for q in query_queues:
    q.put("def")
    q.put(None)  # Signal process termination

for i in range(cpus):
    for match in results_queue.get():
        print(match)
for proc in search_procs: proc.join()
```

This code performs a single search for `"def"` (because it's a common phrase in a directory full of Python files!). In a more production ready system, we would probably hook a socket up to this search code. In that case, we'd have to change the inter-process protocol so that the message coming back on the return queue contained enough information to identify which of many queries the results were attached to.

This use of queues is actually a local version of what could become a distributed system. Imagine if the searches were being sent out to multiple computers and then recombined. We won't discuss it here, but the multiprocessing module includes a manager class that can take a lot of the boilerplate out of the preceding code. There is even a version of the `multiprocessing.Manager` that can manage subprocesses on remote systems to construct a rudimentary distributed application. Check the Python multiprocessing documentation if you are interested in pursuing this further.

The problems with multiprocessing

As threads do, multiprocessing also has problems, some of which we have already discussed. There is no best way to do concurrency; this is especially true in Python. We always need to examine the parallel problem to figure out which of the many available solutions is the best one for that problem. Sometimes, there is no best solution.

In the case of multiprocessing, the primary drawback is that sharing data between processes is very costly. As we have discussed, all communication between processes, whether by queues, pipes, or a more implicit mechanism requires pickling the objects. Excessive pickling quickly dominates processing time. Multiprocessing works best when relatively small objects are passed between processes and a tremendous amount of work needs to be done on each one. On the other hand, if no communication between processes is required, there may not be any point in using the module at all; we can spin up four separate Python processes and use them independently.

The other major problem with multiprocessing is that, like threads, it can be hard to tell which process a variable or method is being accessed in. In multiprocessing, if you access a variable from another process it will usually overwrite the variable in the currently running process while the other process keeps the old value. This is really confusing to maintain, so don't do it.

Futures

Let's start looking at a more asynchronous way of doing concurrency. Futures wrap either multiprocessing or threading depending on what kind of concurrency we need (tending towards I/O versus tending towards CPU). They don't completely solve the problem of accidentally altering shared state, but they allow us to structure our code such that it is easier to track down when we do so. Futures provide distinct boundaries between the different threads or processes. Similar to the multiprocessing pool, they are useful for "call and answer" type interactions in which processing can happen in another thread and then at some point in the future (they are aptly named, after all), you can ask it for the result. It's really just a wrapper around multiprocessing pools and thread pools, but it provides a cleaner API and encourages nicer code.

A future is an object that basically wraps a function call. That function call is run in the background in a thread or process. The future object has methods to check if the future has completed and to get the results after it has completed.

Let's do another file search example. In the last section, we implemented a version of the `unix grep` command. This time, let's do a simple version of the `find` command. The example will search the entire filesystem for paths that contain a given string of characters:

```
from concurrent.futures import ThreadPoolExecutor
from pathlib import Path
from os.path import sep as pathsep
from collections import deque

def find_files(path, query_string):
    subdirs = []
    for p in path.iterdir():
        full_path = str(p.absolute())
        if p.is_dir() and not p.is_symlink():
            subdirs.append(p)
        if query_string in full_path:
                print(full_path)

    return subdirs

query = '.py'
futures = deque()
basedir = Path(pathsep).absolute()

with ThreadPoolExecutor(max_workers=10) as executor:
    futures.append(
        executor.submit(find_files, basedir, query))
    while futures:
        future = futures.popleft()
        if future.exception():
            continue
        elif future.done():
            subdirs = future.result()
            for subdir in subdirs:
                futures.append(executor.submit(
                    find_files, subdir, query))
        else:
            futures.append(future)
```

This code consists of a function named `find_files` that is run in a separate thread (or process, if we used `ProcessPoolExecutor`). There isn't anything particularly special about this function, but note how it does not access any global variables. All interaction with the external environment is passed into the function or returned from it. This is not a technical requirement, but it is the best way to keep your brain inside your skull when programming with futures.

Accessing outside variables without proper synchronization results in something called a **race** condition. For example, imagine two concurrent writes trying to increment an integer counter. They start at the same time and both read the value as 5. Then they both increment the value and write back the result as 6. But if two processes are trying to increment a variable, the expected result would be that it gets incremented by two, so the result should be 7. Modern wisdom is that the easiest way to avoid doing this is to keep as much state as possible private and share them through known-safe constructs, such as queues.

We set up a couple variables before we get started; we'll be searching for all files that contain the characters `'.py'` for this example. We have a queue of futures that we'll discuss shortly. The `basedir` variable points to the root of the filesystem; `'/'` on Unix machines and probably `C:\` on Windows.

First, let's have a short course on search theory. This algorithm implements breadth first search in parallel. Rather than recursively searching every directory using a depth first search, it adds all the subdirectories in the current folder to the queue, then all the subdirectories of each of those folders and so on.

The meat of the program is known as an event loop. We can construct a `ThreadPoolExecutor` as a context manager so that it is automatically cleaned up and its threads closed when it is done. It requires a `max_workers` argument to indicate the number of threads running at a time; if more than this many jobs are submitted, it queues up the rest until a worker thread becomes available. When using `ProcessPoolExecutor`, this is normally constrained to the number of CPUs on the machine, but with threads, it can be much higher, depending how many are waiting on I/O at a time. Each thread takes up a certain amount of memory, so it shouldn't be too high; it doesn't take all that many threads before the speed of the disk, rather than number of parallel requests, is the bottleneck.

Once the executor has been constructed, we submit a job to it using the root directory. The `submit()` method immediately returns a `Future` object, which promises to give us a result eventually. The future is placed on the queue. The loop then repeatedly removes the first future from the queue and inspects it. If it is still running, it gets added back to the end of the queue. Otherwise, we check if the function raised an exception with a call to `future.exception()`. If it did, we just ignore it (it's usually a permission error, although a real app would need to be more careful about what the exception was). If we didn't check this exception here, it would be raised when we called `result()` and could be handled through the normal `try...except` mechanism.

Assuming no exception occurred, we can call `result()` to get the return value of the function call. Since the function returns a list of subdirectories that are not symbolic links (my lazy way of preventing an infinite loop), `result()` returns the same thing. These new subdirectories are submitted to the executor and the resulting futures are tossed onto the queue to have their contents searched in a later iteration.

So that's all that is required to develop a future-based I/O-bound application. Under the hood, it's using the same thread or process APIs we've already discussed, but it provides a more understandable interface and makes it easier to see the boundaries between concurrently running functions (just don't try to access global variables from inside the future!).

AsyncIO

AsyncIO is the current state of the art in Python concurrent programming. It combines the concept of futures and an event loop with the coroutines we discussed in *Chapter 9, The Iterator Pattern*. The result is about as elegant and easy to understand as it is possible to get when writing concurrent code, though that isn't saying a lot!

AsyncIO can be used for a few different concurrent tasks, but it was specifically designed for network I/O. Most networking applications, especially on the server side, spend a lot of time waiting for data to come in from the network. This can be solved by handling each client in a separate thread, but threads use up memory and other resources. AsyncIO uses coroutines instead of threads.

The library also provides its own event loop, obviating the need for the several lines long while loop in the previous example. However, event loops come with a cost. When we run code in an async task on the event loop, that code must return immediately, blocking neither on I/O nor on long-running calculations. This is a minor thing when writing our own code, but it means that any standard library or third-party functions that block on I/O have to have non-blocking versions created.

AsyncIO solves this by creating a set of coroutines that use the `yield from` syntax to return control to the event loop immediately. The event loop takes care of checking whether the blocking call has completed and performing any subsequent tasks, just like we did manually in the previous section.

AsyncIO in action

A canonical example of a blocking function is the `time.sleep` call. Let's use the asynchronous version of this call to illustrate the basics of an AsyncIO event loop:

```
import asyncio
import random

@asyncio.coroutine
def random_sleep(counter):
    delay = random.random() * 5
    print("{} sleeps for {:.2f} seconds".format(counter, delay))
    yield from asyncio.sleep(delay)
    print("{} awakens".format(counter))

@asyncio.coroutine
def five_sleepers():
    print("Creating five tasks")
    tasks = [
        asyncio.async(random_sleep(i)) for i in range(5)]
    print("Sleeping after starting five tasks")
    yield from asyncio.sleep(2)
    print("Waking and waiting for five tasks")
    yield from asyncio.wait(tasks)

asyncio.get_event_loop().run_until_complete(five_sleepers())
print("Done five tasks")
```

This is a fairly basic example, but it covers several features of AsyncIO programming. It is easiest to understand in the order that it executes, which is more or less bottom to top.

The second last line gets the event loop and instructs it to run a future until it is finished. The future in question is named `five_sleepers`. Once that future has done its work, the loop will exit and our code will terminate. As asynchronous programmers, we don't need to know too much about what happens inside that `run_until_complete` call, but be aware that a lot is going on. It's a souped up coroutine version of the futures loop we wrote in the previous chapter that knows how to deal with iteration, exceptions, function returns, parallel calls, and more.

Now look a little more closely at that `five_sleepers` future. Ignore the decorator for a few paragraphs; we'll get back to it. The coroutine first constructs five instances of the `random_sleep` future. The resulting futures are wrapped in an `asyncio.async` task, which adds them to the loop's task queue so they can execute concurrently when control is returned to the event loop.

That control is returned whenever we call `yield from`. In this case, we call `yield from asyncio.sleep` to pause execution of this coroutine for two seconds. During this break, the event loop executes the tasks that it has queued up; namely the five `random_sleep` futures. These coroutines each print a starting message, then send control back to the event loop for a specific amount of time. If any of the sleep calls inside `random_sleep` are shorter than two seconds, the event loop passes control back into the relevant future, which prints its awakening message before returning. When the sleep call inside `five_sleepers` wakes up, it executes up to the next yield from call, which waits for the remaining `random_sleep` tasks to complete. When all the sleep calls have finished executing, the `random_sleep` tasks return, which removes them from the event queue. Once all five of those are completed, the `asyncio.wait` call and then the `five_sleepers` method also return. Finally, since the event queue is now empty, the `run_until_complete` call is able to terminate and the program ends.

The `asyncio.coroutine` decorator mostly just documents that this coroutine is meant to be used as a future in an event loop. In this case, the program would run just fine without the decorator. However, the `asyncio.coroutine` decorator can also be used to wrap a normal function (one that doesn't yield) so that it can be treated as a future. In this case, the entire function executes before returning control to the event loop; the decorator just forces the function to fulfill the coroutine API so the event loop knows how to handle it.

Reading an AsyncIO future

An AsyncIO coroutine executes each line in order until it encounters a `yield from` statement, at which point it returns control to the event loop. The event loop then executes any other tasks that are ready to run, including the one that the original coroutine was waiting on. Whenever that child task completes, the event loop sends the result back into the coroutine so that it can pick up executing until it encounters another `yield from` statement or returns.

This allows us to write code that executes synchronously until we explicitly need to wait for something. This removes the nondeterministic behavior of threads, so we don't need to worry nearly so much about shared state.

It's still a good idea to avoid accessing shared state from inside a coroutine. It makes your code much easier to reason about. More importantly, even though an ideal world might have all asynchronous execution happen inside coroutines, the reality is that some futures are executed behind the scenes inside threads or processes. Stick to a "share nothing" philosophy to avoid a ton of difficult bugs.

In addition, AsyncIO allows us to collect logical sections of code together inside a single coroutine, even if we are waiting for other work elsewhere. As a specific instance, even though the `yield from asyncio.sleep` call in the `random_sleep` coroutine is allowing a ton of stuff to happen inside the event loop, the coroutine itself looks like it's doing everything in order. This ability to read related pieces of asynchronous code without worrying about the machinery that waits for tasks to complete is the primary benefit of the AsyncIO module.

AsyncIO for networking

AsyncIO was specifically designed for use with network sockets, so let's implement a DNS server. More accurately, let's implement one extremely basic feature of a DNS server.

The domain name system's basic purpose is to translate domain names, such as www.amazon.com into IP addresses such as 72.21.206.6. It has to be able to perform many types of queries and know how to contact other DNS servers if it doesn't have the answer required. We won't be implementing any of this, but the following example is able to respond directly to a standard DNS query to look up IPs for my three most recent employers:

```
import asyncio
from contextlib import suppress

ip_map = {
    b'facebook.com.': '173.252.120.6',
    b'yougov.com.': '213.52.133.246',
    b'wipo.int.': '193.5.93.80'
}

def lookup_dns(data):
    domain = b''
    pointer, part_length = 13, data[12]
    while part_length:
```

```
            domain += data[pointer:pointer+part_length] + b'.'
            pointer += part_length + 1
            part_length = data[pointer - 1]

        ip = ip_map.get(domain, '127.0.0.1')

        return domain, ip

    def create_response(data, ip):
        ba = bytearray
        packet = ba(data[:2]) + ba([129, 128]) + data[4:6] * 2
        packet += ba(4) + data[12:]
        packet += ba([192, 12, 0, 1, 0, 1, 0, 0, 0, 60, 0, 4])
        for x in ip.split('.'): packet.append(int(x))
        return packet

    class DNSProtocol(asyncio.DatagramProtocol):
        def connection_made(self, transport):
            self.transport = transport

        def datagram_received(self, data, addr):
            print("Received request from {}".format(addr[0]))
            domain, ip = lookup_dns(data)
            print("Sending IP {} for {} to {}".format(
                domain.decode(), ip, addr[0]))
            self.transport.sendto(
                create_response(data, ip), addr)

    loop = asyncio.get_event_loop()
    transport, protocol = loop.run_until_complete(
        loop.create_datagram_endpoint(
            DNSProtocol, local_addr=('127.0.0.1', 4343)))
    print("DNS Server running")

    with suppress(KeyboardInterrupt):
        loop.run_forever()
    transport.close()
    loop.close()
```

This example sets up a dictionary that dumbly maps a few domains to IPv4 addresses. It is followed by two functions that extract information from a binary DNS query packet and construct the response. We won't be discussing these; if you want to know more about DNS read RFC ("request for comment", the format for defining most Internet protocols) 1034 and 1035.

You can test this service by running the following command in another terminal:

```
nslookup -port=4343 facebook.com localhost
```

Let's get on with the entrée. AsyncIO networking revolves around the intimately linked concepts of transports and protocols. A protocol is a class that has specific methods that are called when relevant events happen. Since DNS runs on top of **UDP** (**User Datagram Protocol**); we build our protocol class as a subclass of `DatagramProtocol`. This class has a variety of events that it can respond to; we are specifically interested in the initial connection occurring (solely so we can store the transport for future use) and the `datagram_received` event. For DNS, each received datagram must be parsed and responded to, at which point the interaction is over.

So, when a datagram is received, we process the packet, look up the IP, and construct a response using the functions we aren't talking about (they're black sheep in the family). Then we instruct the underlying transport to send the resulting packet back to the requesting client using its `sendto` method.

The transport essentially represents a communication stream. In this case, it abstracts away all the fuss of sending and receiving data on a UDP socket on an event loop. There are similar transports for interacting with TCP sockets and subprocesses, for example.

The UDP transport is constructed by calling the loop's `create_datagram_endpoint` coroutine. This constructs the appropriate UDP socket and starts listening on it. We pass it the address that the socket needs to listen on, and importantly, the protocol class we created so that the transport knows what to call when it receives data.

Since the process of initializing a socket takes a non-trivial amount of time and would block the event loop, the `create_datagram_endpoint` function is a coroutine. In our example, we don't really need to do anything while we wait for this initialization, so we wrap the call in `loop.run_until_complete`. The event loop takes care of managing the future, and when it's complete, it returns a tuple of two values: the newly initialized transport and the protocol object that was constructed from the class we passed in.

Behind the scenes, the transport has set up a task on the event loop that is listening for incoming UDP connections. All we have to do, then, is start the event loop running with the call to `loop.run_forever()` so that task can process these packets. When the packets arrive, they are processed on the protocol and everything just works.

The only other major thing to pay attention to is that transports (and, indeed, event loops) are supposed to be closed when we are finished with them. In this case, the code runs just fine without the two calls to `close()`, but if we were constructing transports on the fly (or just doing proper error handling!), we'd need to be quite a bit more conscious of it.

You may have been dismayed to see how much boilerplate is required in setting up a protocol class and underlying transport. AsyncIO provides an abstraction on top of these two key concepts called streams. We'll see an example of streams in the TCP server in the next example.

Using executors to wrap blocking code

AsyncIO provides its own version of the futures library to allow us to run code in a separate thread or process when there isn't an appropriate non-blocking call to be made. This essentially allows us to combine threads and processes with the asynchronous model. One of the more useful applications of this feature is to get the best of both worlds when an application has bursts of I/O-bound and CPU-bound activity. The I/O-bound portions can happen in the event-loop while the CPU-intensive work can be spun off to a different process. To illustrate this, let's implement "sorting as a service" using AsyncIO:

```
import asyncio
import json
from concurrent.futures import ProcessPoolExecutor

def sort_in_process(data):
    nums = json.loads(data.decode())
    curr = 1
    while curr < len(nums):
        if nums[curr] >= nums[curr-1]:
            curr += 1
        else:
            nums[curr], nums[curr-1] = \
                nums[curr-1], nums[curr]
            if curr > 1:
```

```
                curr -= 1

    return json.dumps(nums).encode()

@asyncio.coroutine
def sort_request(reader, writer):
    print("Received connection")
    length = yield from reader.read(8)
    data = yield from reader.readexactly(
        int.from_bytes(length, 'big'))
    result = yield from asyncio.get_event_loop().run_in_executor(
        None, sort_in_process, data)
    print("Sorted list")
    writer.write(result)
    writer.close()
    print("Connection closed")

loop = asyncio.get_event_loop()
loop.set_default_executor(ProcessPoolExecutor())
server = loop.run_until_complete(
    asyncio.start_server(sort_request, '127.0.0.1', 2015))
print("Sort Service running")

loop.run_forever()
server.close()
loop.run_until_complete(server.wait_closed())
loop.close()
```

This is an example of good code implementing some really stupid ideas. The whole idea of sort as a service is pretty ridiculous. Using our own sorting algorithm instead of calling Python's `sorted` is even worse. The algorithm we used is called gnome sort, or in some cases, "stupid sort". It is a slow sort algorithm implemented in pure Python. We defined our own protocol instead of using one of the many perfectly suitable application protocols that exist in the wild. Even the idea of using multiprocessing for parallelism might be suspect here; we still end up passing all the data into and out of the subprocesses. Sometimes, it's important to take a step back from the program you are writing and ask yourself if you are trying to meet the right goals.

But let's look at some of the smart features of this design. First, we are passing bytes into and out of the subprocess. This is a lot smarter than decoding the JSON in the main process. It means the (relatively expensive) decoding can happen on a different CPU. Also, pickled JSON strings are generally smaller than pickled lists, so less data is passing between processes.

Second, the two methods are very linear; it looks like code is being executed one line after another. Of course, in AsyncIO, this is an illusion, but we don't have to worry about shared memory or concurrency primitives.

Streams

The previous example should look familiar by now as it has a similar boilerplate to other AsyncIO programs. However, there are a few differences. You'll notice we called `start_server` instead of `create_server`. This method hooks into AsyncIO's streams instead of using the underlying transport/protocol code. Instead of passing in a protocol class, we can pass in a normal coroutine, which receives reader and writer parameters. These both represent streams of bytes that can be read from and written like files or sockets. Second, because this is a TCP server instead of UDP, there is some socket cleanup required when the program finishes. This cleanup is a blocking call, so we have to run the `wait_closed` coroutine on the event loop.

Streams are fairly simple to understand. Reading is a potentially blocking call so we have to call it with `yield from`. Writing doesn't block; it just puts the data on a queue, which AsyncIO sends out in the background.

Our code inside the `sort_request` method makes two read requests. First, it reads 8 bytes from the wire and converts them to an integer using big endian notation. This integer represents the number of bytes of data the client intends to send. So in the next call, to `readexactly`, it reads that many bytes. The difference between `read` and `readexactly` is that the former will read up to the requested number of bytes, while the latter will buffer reads until it receives all of them, or until the connection closes.

Executors

Now let's look at the executor code. We import the exact same `ProcessPoolExecutor` that we used in the previous section. Notice that we don't need a special AsyncIO version of it. The event loop has a handy `run_in_executor` coroutine that we can use to run futures on. By default, the loop runs code in `ThreadPoolExecutor`, but we can pass in a different executor if we wish. Or, as we did in this example, we can set a different default when we set up the event loop by calling `loop.set_default_executor()`.

As you probably recall from the previous section, there is not a lot of boilerplate for using futures with an executor. However, when we use them with AsyncIO, there is none at all! The coroutine automatically wraps the function call in a future and submits it to the executor. Our code blocks until the future completes, while the event loop continues processing other connections, tasks, or futures. When the future is done, the coroutine wakes up and continues on to write the data back to the client.

You may be wondering if, instead of running multiple processes inside an event loop, it might be better to run multiple event loops in different processes. The answer is: "maybe". However, depending on the exact problem space, we are probably better off running independent copies of a program with a single event loop than to try to coordinate everything with a master multiprocessing process.

We've hit most of the high points of AsyncIO in this section, and the chapter has covered many other concurrency primitives. Concurrency is a hard problem to solve, and no one solution fits all use cases. The most important part of designing a concurrent system is deciding which of the available tools is the correct one to use for the problem. We have seen advantages and disadvantages of several concurrent systems, and now have some insights into which are the better choices for different types of requirements.

Case study

To wrap up this chapter, and the book, let's build a basic image compression tool. It will take black and white images (with 1 bit per pixel, either on or off) and attempt to compress it using a very basic form of compression known as run-length encoding. You may find black and white images a bit far-fetched. If so, you haven't enjoyed enough hours at `http://xkcd.com`!

I've included some sample black and white BMP images (which are easy to read data into and leave a lot of opportunity to improve on file size) with the example code for this chapter.

We'll be compressing the images using a simple technique called run-length encoding. This technique basically takes a sequence of bits and replaces any strings of repeated bits with the number of bits that are repeated. For example, the string 000011000 might be replaced with 04 12 03 to indicate that 4 zeros are followed by 2 ones and then 3 more zeroes. To make things a little more interesting, we will break each row into 127 bit chunks.

I didn't pick 127 bits arbitrarily. 127 different values can be encoded into 7 bits, which means that if a row contains all ones or all zeros, we can store it in a single byte; the first bit indicating whether it is a row of 0s or a row of 1s, and the remaining 7 bits indicating how many of that bit exists.

Breaking up the image into blocks has another advantage; we can process individual blocks in parallel without them depending on each other. However, there's a major disadvantage as well; if a run has just a few ones or zeros in it, then it will take up `more` space in the compressed file. When we break up long runs into blocks, we may end up creating more of these small runs and bloat the size of the file.

When dealing with files, we have to think about the exact layout of the bytes in the compressed file. Our file will store two byte little-endian integers at the beginning of the file representing the width and height of the completed file. Then it will write bytes representing the 127 bit chunks of each row.

Now before we start designing a concurrent system to build such compressed images, we should ask a fundamental question: Is this application I/O-bound or CPU-bound?

My answer, honestly, is "I don't know". I'm not sure whether the app will spend more time loading data from disk and writing it back or doing the compression in memory. I suspect that it is a CPU bound app in principle, but once we start passing image strings into subprocesses, we may lose any benefit of parallelism. The optimal solution to this problem is probably to write a C or Cython extension, but let's see how far we can get in pure Python.

We'll build this application using bottom-up design. That way we'll have some building blocks that we can combine into different concurrency patterns to see how they compare. Let's start with the code that compresses a 127-bit chunk using run-length encoding:

```
from bitarray import bitarray
def compress_chunk(chunk):
    compressed = bytearray()
    count = 1
    last = chunk[0]
    for bit in chunk[1:]:
        if bit != last:
            compressed.append(count | (128 * last))
            count = 0
            last = bit
        count += 1
    compressed.append(count | (128 * last))
    return compressed
```

This code uses the `bitarray` class for manipulating individual zeros and ones. It is distributed as a third-party module, which you can install with the command `pip install bitarray`. The chunk that is passed into `compress_chunks` is an instance of this class (although the example would work just as well with a list of Booleans). The primary benefit of the bitarray in this case is that when pickling them between processes, they take up an 8th of the space of a list of Booleans or a bytestring of 1s and 0s. Therefore, they pickle faster. They are also a bit (pun intended) easier to work with than doing a ton of bitwise operations.

The method compresses the data using run-length encoding and returns a bytearray containing the packed data. Where a bitarray is like a list of ones and zeros, a bytearray is like a list of byte objects (each byte, of course, containing 8 ones or zeros).

The algorithm that performs the compression is pretty simple (although I'd like to point out that it took me two days to implement and debug it. Simple to understand does not necessarily imply easy to write!). It first sets the `last` variable to the type of bit in the current run (either `True` or `False`). It then loops over the bits, counting each one, until it finds one that is different. When it does, it constructs a new byte by making the leftmost bit of the byte (the 128 position) either a zero or a one, depending on what the `last` variable contained. Then it resets the counter and repeats the operation. Once the loop is done, it creates one last byte for the last run, and returns the result.

While we're creating building blocks, let's make a function that compresses a row of image data:

```
def compress_row(row):
    compressed = bytearray()
    chunks = split_bits(row, 127)
    for chunk in chunks:
        compressed.extend(compress_chunk(chunk))
    return compressed
```

This function accepts a bitarray named row. It splits it into chunks that are each 127 bits wide using a function that we'll define very shortly. Then it compresses each of those chunks using the previously defined `compress_chunk`, concatenating the results into a `bytearray`, which it returns.

We define `split_bits` as a simple generator:

```
def split_bits(bits, width):
    for i in range(0, len(bits), width):
        yield bits[i:i+width]
```

Now, since we aren't certain yet whether this will run more effectively in threads or processes, let's wrap these functions in a method that runs everything in a provided executor:

```
def compress_in_executor(executor, bits, width):
    row_compressors = []
    for row in split_bits(bits, width):
        compressor = executor.submit(compress_row, row)
```

```
            row_compressors.append(compressor)

        compressed = bytearray()
        for compressor in row_compressors:
            compressed.extend(compressor.result())
        return compressed
```

This example barely needs explaining; it splits the incoming bits into rows based on the width of the image using the same `split_bits` function we have already defined (hooray for bottom-up design!).

Note that this code will compress any sequence of bits, although it would bloat, rather than compress binary data that has frequent changes in bit values. Black and white images are definitely good candidates for the compression algorithm in question. Let's now create a function that loads an image file using the third-party pillow module, converts it to bits, and compresses it. We can easily switch between executors using the venerable comment statement:

```
from PIL import Image
def compress_image(in_filename, out_filename, executor=None):
    executor = executor if executor else ProcessPoolExecutor()
    with Image.open(in_filename) as image:
        bits = bitarray(image.convert('1').getdata())
        width, height = image.size

    compressed = compress_in_executor(executor, bits, width)

    with open(out_filename, 'wb') as file:
        file.write(width.to_bytes(2, 'little'))
        file.write(height.to_bytes(2, 'little'))
        file.write(compressed)

def single_image_main():
    in_filename, out_filename = sys.argv[1:3]
    #executor = ThreadPoolExecutor(4)
    executor = ProcessPoolExecutor()
    compress_image(in_filename, out_filename, executor)
```

The `image.convert()` call changes the image to black and white (one bit) mode, while `getdata()` returns an iterator over those values. We pack the results into a bitarray so they transfer across the wire more quickly. When we output the compressed file, we first write the width and height of the image followed by the compressed data, which arrives as a bytearray, which can be written directly to the binary file.

Having written all this code, we are finally able to test whether thread pools or process pools give us better performance. I created a large (7200 x 5600 pixels) black and white image and ran it through both pools. The `ProcessPool` takes about 7.5 seconds to process the image on my system, while the `ThreadPool` consistently takes about 9. Thus, as we suspected, the cost of pickling bits and bytes back and forth between processes is eating almost all the efficiency gains from running on multiple processors (though looking at my CPU monitor, it does fully utilize all four cores on my machine).

So it looks like compressing a single image is most effectively done in a separate process, but only barely because we are passing so much data back and forth between the parent and subprocesses. Multiprocessing is more effective when the amount of data passed between processes is quite low.

So let's extend the app to compress all the bitmaps in a directory in parallel. The only thing we'll have to pass into the subprocesses are filenames, so we should get a speed gain compared to using threads. Also, to be kind of crazy, we'll use the existing code to compress individual images. This means we'll be running a `ProcessPoolExecutor` inside each subprocess to create even more subprocesses. I don't recommend doing this in real life!

```
from pathlib import Path
def compress_dir(in_dir, out_dir):
    if not out_dir.exists():
        out_dir.mkdir()

    executor = ProcessPoolExecutor()
    for file in (
            f for f in in_dir.iterdir() if f.suffix == '.bmp'):
        out_file = (out_dir / file.name).with_suffix('.rle')
        executor.submit(
            compress_image, str(file), str(out_file))

def dir_images_main():
    in_dir, out_dir = (Path(p) for p in sys.argv[1:3])
    compress_dir(in_dir, out_dir)
```

This code uses the `compress_image` function we defined previously, but runs it in a separate process for each image. It doesn't pass an executor into the function, so `compress_image` creates a `ProcessPoolExecutor` once the new process has started running.

Now that we are running executors inside executors, there are four combinations of threads and process pools that we can be using to compress images. They each have quite different timing profiles:

	Process pool per image	**Thread pool per image**
Process pool per row	42 seconds	53 seconds
Thread pool per row	34 seconds	64 seconds

As we might expect, using threads for each image and again using threads for each row is the slowest, since the GIL prevents us from doing any work in parallel. Given that we were slightly faster when using separate processes for each row when we were using a single image, you may be surprised to see that it is faster to use a `ThreadPool` feature for rows if we are processing each image in a separate process. Take some time to understand why this might be.

My machine contains only four processor cores. Each row in each image is being processed in a separate pool, which means that all those rows are competing for processing power. When there is only one image, we get a (very modest) speedup by running each row in parallel. However, when we increase the number of images being processed at once, the cost of passing all that row data into and out of a subprocess is actively stealing processing time from each of the other images. So, if we can process each image on a separate processor, where the only thing that has to get pickled into the subprocess pipe is a couple filenames, we get a solid speedup.

Thus, we see that different workloads require different concurrency paradigms. Even if we are just using futures we have to make informed decisions about what kind of executor to use.

Also note that for typically-sized images, the program runs quickly enough that it really doesn't matter which concurrency structures we use. In fact, even if we didn't use any concurrency at all, we'd probably end up with about the same user experience.

This problem could also have been solved using the threading and/or multiprocessing modules directly, though there would have been quite a bit more boilerplate code to write. You may be wondering whether or not AsyncIO would be useful here. The answer is: "probably not". Most operating systems don't have a good way to do non-blocking reads from the filesystem, so the library ends up wrapping all the calls in futures anyway.

For completeness, here's the code that I used to decompress the RLE images to confirm that the algorithm was working correctly (indeed, it wasn't until I fixed bugs in both compression and decompression, and I'm still not sure if it is perfect. I should have used test-driven development!):

```
from PIL import Image
import sys

def decompress(width, height, bytes):
    image = Image.new('1', (width, height))

    col = 0
    row = 0
    for byte in bytes:
        color = (byte & 128) >> 7
        count = byte & ~128
        for i in range(count):
            image.putpixel((row, col), color)
            row += 1
        if not row % width:
            col += 1
            row = 0
    return image


with open(sys.argv[1], 'rb') as file:
    width = int.from_bytes(file.read(2), 'little')
    height = int.from_bytes(file.read(2), 'little')

    image = decompress(width, height, file.read())
    image.save(sys.argv[2], 'bmp')
```

This code is fairly straightforward. Each run is encoded in a single byte. It uses some bitwise math to extract the color of the pixel and the length of the run. Then it sets each pixel from that run in the image, incrementing the row and column of the next pixel to check at appropriate intervals.

Exercises

We've covered several different concurrency paradigms in this chapter and still don't have a clear idea of when each one is useful. As we saw in the case study, it is often a good idea to prototype a few different strategies before committing to one.

Concurrency in Python 3 is a huge topic and an entire book of this size could not cover everything there is to know about it. As your first exercise, I encourage you to check out several third-party libraries that may provide additional context:

- execnet, a library that permits local and remote share-nothing concurrency
- Parallel python, an alternative interpreter that can execute threads in parallel
- Cython, a python-compatible language that compiles to C and has primitives to release the gil and take advantage of fully parallel multi-threading.
- PyPy-STM, an experimental implementation of software transactional memory on top of the ultra-fast PyPy implementation of the Python interpreter
- Gevent

If you have used threads in a recent application, take a look at the code and see if you can make it more readable and less bug-prone by using futures. Compare thread and multiprocessing futures to see if you can gain anything by using multiple CPUs.

Try implementing an AsyncIO service for some basic HTTP requests. You may need to look up the structure of an HTTP request on the web; they are fairly simple ASCII packets to decipher. If you can get it to the point that a web browser can render a simple GET request, you'll have a good understanding of AsyncIO network transports and protocols.

Make sure you understand the race conditions that happen in threads when you access shared data. Try to come up with a program that uses multiple threads to set shared values in such a way that the data deliberately becomes corrupt or invalid.

Remember the link collector we covered for the case study in *Chapter 6, Python Data Structures*? Can you make it run faster by making requests in parallel? Is it better to use raw threads, futures, or AsyncIO for this?

Try writing the run-length encoding example using threads or multiprocessing directly. Do you get any speed gains? Is the code easier or harder to reason about? Is there any way to speed up the decompression script by using concurrency or parallelism?

Summary

This chapter ends our exploration of object-oriented programming with a topic that isn't very object-oriented. Concurrency is a difficult problem and we've only scratched the surface. While the underlying OS abstractions of processes and threads do not provide an API that is remotely object-oriented, Python offers some really good object-oriented abstractions around them. The threading and multiprocessing packages both provide an object-oriented interface to the underlying mechanics. Futures are able to encapsulate a lot of the messy details into a single object. AsyncIO uses coroutine objects to make our code read as though it runs synchronously, while hiding ugly and complicated implementation details behind a very simple loop abstraction.

Thank you for reading *Python 3 Object-oriented Programming*, *Second Edition*. I hope you've enjoyed the ride and are eager to start implementing object-oriented software in all your future projects!

Module 3

Mastering Python

Master the art of writing beautiful and powerful Python by using all of the features that Python 3.5 offers

1
Getting Started – One Environment per Project

There is one aspect of the Python philosophy that always has been, and always will be, the most important in the entire language—readability, or Pythonic code. This book will help you master writing Python the way it was meant to be: readable, beautiful, explicit, and as simple as possible. In short, it will be Pythonic code. That is not to say that complicated subjects will not be covered. Naturally, they will, but whenever the philosophy of Python is at stake, you will be warned when and where the technique is justified.

Most of the code within this book will function on both Python 2 and Python 3, but the main target is Python 3. There are three reasons for doing this:

1. Python 3 was released in 2008, which is a very long time in the rapidly changing software world. It's not a new thing anymore, it's stable, it's usable, and, most importantly, it's the future.
2. Development for Python 2 effectively stopped in 2009. Certain features have been backported from Python 3 to Python 2, but any new development will be for Python 3 first.
3. Python 3 has become mature. While I have to admit that Python 3.2 and older versions still had a few small issues that made it hard to write code that functions on both Python 2 and 3, Python 3.3 did improve greatly in that aspect, and I consider it mature. This is evidenced by the marginally modified syntax in Python 3.4 and 3.5 and a lot of very useful features, which are covered in this book.

To summarize, Python 3 is an improvement over Python 2. I have been a skeptic for a very long time myself, but I do not see any reason not to use Python 3 for new projects, and even porting existing projects to Python 3 is generally possible with only minor changes. With cool new features such as `async with` in Python 3.5, you will want to upgrade just to try it.

This first chapter will show you how to properly set up an environment, create a new isolated environment, and make sure you get similar results when running the same code on different machines. Most Python programmers are already using `virtualenv` to create virtual Python environments, but the `venv` command, introduced in Python 3.3, is a very nice alternative. It is essentially a clone of the `virtualenv` package but is slightly simpler and bundled with Python. While its usage is mostly analogous to `virtualenv`, there are a few changes that are interesting to know.

Secondly, we will discuss the `pip` command. The `pip` command is automatically installed when using `venv` through the `ensurepip` package, a package introduced in Python 3.4. This package automatically bootstraps `pip` into an existing Python library while maintaining independent versions of Python and `pip`. Before Python 3.4, `venv` came without `pip` and had to be installed manually.

Finally, we will discuss how packages created with `distutils` can be installed. While pure Python packages are generally easy to install, it can get challenging when C modules are involved.

In this chapter, the following topics are covered:

- Creating a virtual Python environment using `venv`
- Bootstrapping pip using `ensurepip`
- Installing packages based on `distutils` (C/C++) with `pip`

Creating a virtual Python environment using venv

Most Python programmers are already be familiar with `venv` or `virtualenv`, but even if you're not, it's never too late to start using it. The `venv` module is designed to isolate your Python environments so that you can install packages specific to your current project without polluting your global namespace. For example, having a filename such as `sys.py` in your current directory can seriously break your code if you expect to have the standard Python `sys` library—your local sys libraries will be imported before the global one, effectively hiding the system library. In addition, because the packages are installed locally, you don't need system (root/administrator) access to install them.

The result is that you can make sure you have exactly the same version of a package on both your local development machine and production machines without interfering with other packages. For example, there are many Django packages around that require specific versions of the Django project. Using `venv`, you can easily install Django 1.4 for project A and Django 1.8 for project B without them ever knowing that there are different versions installed in other environments. By default, the environments are even configured in such a way that the global packages are not visible. The benefit of this is that to get an exact list of all installed packages within the environment, simply a `pip freeze` will suffice. The downside is that some of the heavier packages (for example, `numpy`) will have to be installed in every separate environment. Needless to say, which choice is the best for your project depends on the project. For most projects, I would keep the default setting of not having the global packages, but when messing around with projects that have lots of C/C++ extensions, it would be convenient to simply enable the global site packages. The reason is simple; if you do not have a compiler available, installing the package locally can be difficult, while the global install has an executable for Windows or an installable package for Linux/Unix available.

The `venv` module (`https://docs.python.org/3/library/venv.html`) can be seen as a slightly simplified version of the `virtualenv` tool (`https://virtualenv.pypa.io/`), which has been bundled with Python since version 3.3 (refer to PEP 0405 -- Python Virtual Environments: `https://www.python.org/dev/peps/pep-0405/`).

The `virtualenv` package can generally be used as a drop-in replacement for `venv`, which is especially relevant for older Python versions (below 3.3) that do not come bundled with `venv`.

Creating your first venv

Creating an environment is quite easy. The basic command comes down to `pyvenv PATH_TO_THE_NEW_VIRTUAL_ENVIRONMENT`, so let's give it a try. Note that this command works on Linux, Unix, and Mac; the Windows command will follow shortly:

```
# pyvenv test_venv
# . ./test_venv/bin/activate
(test_venv) #
```

Some Ubuntu releases (notably 14.04 LTS) maim the Python installation by not including the full `pyvenv` package with `ensurepip`. The standard workaround is to call `pyvenv --without-pip test_env`, which requires a manual `pip` installation through the `get_pip.py` file available on the `pip` home page.

This creates an environment called `test_venv`, and the second line activates the environment.

On Windows, everything is slightly different but similar overall. By default, the `pyvenv` command won't be in your PATH, so running the command is slightly different. The three options are as follows:

- Add the `Python\Tools\Scripts\` directory to your PATH
- Run the module:

  ```
  python -m venv test_venv
  ```

- Run the script directly:

  ```
  python Python\Tools\Scripts\pyvenv.py test_venv
  ```

For convenience, I would recommend that you add the `Scripts` directory to your PATH anyhow, since many other applications/scripts (such as `pip`) will be installed there as well.

Here is the full example for Windows:

```
C:\envs>python -m venv test_venv
C:\envs>test_venv\Scripts\activate.bat
(test_venv) C:\envs>
```

When using Windows PowerShell, the environment can be activated by using `test_venv\Scripts\Activate.ps1` instead. Note that you really do need backslashes here.

venv arguments

So far, we have just created a plain and regular `venv`, but there are a few, really useful flags for customizing your `venv` specifically to your needs.

First, let's look at the `venv` help:

Parameter	Description
`--system-site-packages`	It gives the virtual environment access to the `system-site-packages` directory
`--symlinks`	Try to use `symlinks` rather than copies when symlinks are not the default for the platform
`--copies`	Try to use copies rather than symlinks even when symlinks are the default for the platform
`--clear`	Delete the contents of the environment directory, if it exists, before environment creation
`--upgrade`	Upgrade the environment directory to use this version of Python, assuming that Python has been upgraded in-place
`--without-pip`	This skips installing or upgrading pip in the virtual environment (`pip` is bootstrapped by default)

The most important argument to note is `--system-site-packages`, which enables the global site packages within the environment. This means that if you have a package installed in your global Python version, it will be available within your environment as well. However, if you try to update it to a different version, it will be installed locally. Whenever possible, I would recommend disabling the `--system-site-packages` flag because it gives you a simple environment without too many variables. A simple update of the system packages could break your virtual environment otherwise, but worse, there is no way to know which packages are needed locally and which ones are just installed for other purposes.

To enable this for an existing environment, you can simply run the environment creation command again, but this time adding the `--system-site-packages` flag to enable the global site packages.

To disable it again, you can simply run the environment creation command without the flag. This will keep the locally (within the environment) installed packages available but will remove the global packages from your Python scope.

When using `virtualenvwrapper`, this can also be done with the `toggleglobalsitepackages` command from within the activated environment.

The `--symlinks` and `--copies` arguments can generally be ignored, but it is important to know the difference. These arguments decide whether the files will be copied from the base python directory or whether they will be symlinked.

Symlinks are a Linux/Unix/Mac thing; instead of copying a file it creates a symbolic link that tells the system where to find the actual file.

By default, `venv` will try to symlink the files, and if that fails, it will fall back to copying. Since Windows Vista and Python 3.2, this is also supported on Windows, so unless you're using a very old system, you will most likely be using symlinks in your environment. The benefit of symlinks is that it saves disk space and stays in sync with your Python installation. The downside is that if your system's Python version undergoes an upgrade, it can break the packages installed within your environment, but that can easily be fixed by reinstalling the packages using `pip`.

Finally, the `--upgrade` argument is useful if your system Python version has been upgraded in-place. The most common use case for this argument is for repairing broken environments after upgrading the system Python with a copied (as opposed to symlinked) environment.

Differences between virtualenv and venv

Since the `venv` module is essentially a simpler version of `virtualenv`, they are mostly the same, but some things are different. Also, since `virtualenv` is a package that is distributed separately from Python, it does have some advantages.

The following are the advantages of `venv` over `virtualenv`:

- `venv` is distributed with Python 3.3 and above, so no separate install is needed
- `venv` is simple and straightforward with no features besides the bare necessities

Advantages of `virtualenv` over `venv`:

- `virtualenv` is distributed outside of Python, so it can be updated separately.
- `virtualenv` works on old Python versions, but Python 2.6 or a higher version is recommended. However, Python 2.5 support is possible with older versions (1.9.x or lower).
- It supports convenient wrappers, such as `virtualenvwrapper` (`http://virtualenvwrapper.readthedocs.org/`)

In short, if `venv` is enough for you, use it. If you are using an old Python version or want some extra convenience, such as `virtualenvwrapper`, use `virtualenv` instead. Both projects essentially do the same thing, and efforts have been made to easily switch between them. The biggest and most significant difference between the two is the wide variety of Python versions that `virtualenv` supports.

Bootstrapping pip using ensurepip

Slowly, the `pip` package manager has been replacing `easy_install` since its introduction in 2008. Since Python 3.4, it has even become the default and is bundled with Python. Since Python 3.4 onward, it is installed by default within both the regular Python environment and that of `pyvenv`; before that, a manual install is required. To automatically install `pip` in Python 3.4 and above, the `ensurepip` library is used. This is a library that handles automatic installation and/or upgrades of `pip`, so it is at least as recent as the one bundled with `ensurepip`.

ensurepip usage

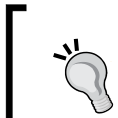
The usage of `ensurepip` is fairly straightforward. Just run python `-m ensurepip` to guarantee a `pip` version or python `-m ensurepip --upgrade` to make sure that `pip` will be at least the version that is bundled with `ensurepip`.

In addition to installing the regular `pip` shortcut, this will also install the `pipX` and `pipX.Y` links, which allow you to select a specific Python version. When using Python 2 and Python 3 simultaneously, this allows you to install packages within Python 2 and Python 3 with `pip2` and `pip3`, respectively. This means that if you use python `-m ensurepip` on Python 3.5 you will get `pip`, `pip3`, and `pip3.5` commands installed in your environment.

Manual pip install

The `ensurepip` package is great if you are using Python 3.4 or above. Below that, however, you need to install `pip` manually. Actually, this is surprisingly easy. It involves just two steps:

1. Download the `get-pip.py` file: `https://bootstrap.pypa.io/get-pip.py`.
2. Execute the `get-pip.py` file: python `get-pip.py`.



If the `ensurepip` command fails due to permission errors, it can be useful to supply the `--user` argument. This allows you to install `pip` inside the user `specific site packages` directory, so root/admin access is not required.

Installing C/C++ packages

Most Python packages are purely Python and blissfully easy to install, just as a simple `pip install packagename` does the trick. However, there are cases where compilation is involved and installation goes from a simple pip install to searching for hours to see which dependencies are needed to install a certain package.

The specific error message will differ as per the project and environment, but there is a common pattern in these errors, and understanding what you are looking at can help a lot when searching for a solution.

For example, when installing `pillow` on a standard Ubuntu machine, you'll get a few pages full of errors, warnings, and other messages that end like this:

```
    x86_64-linux-gnu-gcc: error: build/temp.linux-x86_64-3.4/libImaging/
Jpeg2KDecode.o: No such file or directory
    x86_64-linux-gnu-gcc: error: build/temp.linux-x86_64-3.4/libImaging/
Jpeg2KEncode.o: No such file or directory
    x86_64-linux-gnu-gcc: error: build/temp.linux-x86_64-3.4/libImaging/
BoxBlur.o: No such file or directory
    error: command 'x86_64-linux-gnu-gcc' failed with exit status 1


    ----------------------------------------
Command "python3 -c "import setuptools, tokenize;__file__='/tmp/pip-
build-_f0ryusw/pillow/setup.py';exec(compile(getattr(tokenize, 'open',
open)(__file__).read().replace('\r\n', '\n'), __file__, 'exec'))" install
--record /tmp/pip-kmmobum2-record/install-record.txt --single-version-
externally-managed --compile --install-headers include/site/python3.4/
pillow" failed with error code 1 in /tmp/pip-build-_f0ryusw/pillow
```

Upon seeing messages like these, you might be tempted to search for one of the lines such as `x86_64-linux-gnu-gcc: error: build/temp.linux-x86_64-3.4/libImaging/Jpeg2KDecode.o: No such file or directory`. While this might give you some relevant results, most likely it will not. The trick with installations like these is to scroll up until you see messages about missing headers. Here is an example:

```
  In file included from libImaging/Imaging.h:14:0,
                   from libImaging/Resample.c:16:
  libImaging/ImPlatform.h:10:20: fatal error: Python.h: No such file or
directory
   #include "Python.h"
                     ^
  compilation terminated.
```

The key message here is that `Python.h` is missing. These are part of the Python headers and are needed for the compilation of most C/C++ packages within Python. Depending on the operating system, the solutions will vary—unfortunately. So, I recommend that you skip all parts of this paragraph that are not relevant for your case.

Debian and Ubuntu

In Debian and Ubuntu, the package to be installed is `python3-dev` or `python2-dev` if you're still using Python 2. The command to execute is as follows:

```
# sudo apt-get install python3-dev
```

However, this installs the development headers only. If you want the compiler and other headers bundled with the install, then the `build-dep` command is also very useful. Here is an example:

```
# sudo apt-get build-dep python3
```

Red Hat, CentOS, and Fedora

Red Hat, CentOS, and Fedora are rpm-based distros that use the `yum` package manager to install the requirements. Most development headers are available through `<package-name>-devel` and are easily installable as such. To install the Python 3 development headers, use this line:

```
# sudo apt-get install python3-devel
```

To make sure you have all the requirements such as development headers and compilers to build packages such as Python, the `yum-builddep` command is available:

```
# yum-builddep python3
```

OS X

The install procedure on OS X consists of three steps before the actual package can be installed.

First, you have to install Xcode. This can be done through the OS X App Store at `https://itunes.apple.com/en/app/xcode/id497799835?mt=12`.

Then you have to install the Xcode command-line tools:

```
# xcode-select --install
```

Finally, you need to install the **Homebrew** package manager. The steps are available at `http://brew.sh/`, but the install command is as follows:

```
# /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/
Homebrew/install/master/install)"
```

Other package managers, such as `Macports`, are also possible, but `Homebrew` is currently the OS X package manager with the most active development and community.

Once all of these steps have been completed, you should have a working Homebrew installation. The working of `Homebrew` can be verified using the `brew doctor` command. If there are no major errors in the output, then you should be ready to install your first packages through brew. Now we simply need to install Python and we're done:

```
# brew install python3
```

Windows

On Windows, manual compilation of C Python packages is generally a non-trivial task to say the least. Most packages have been written with Linux/Unix systems in mind (OS X falls under the Unix category), and Windows is a nice-to-have for developers. The result is that packages are difficult to compile on Windows because there are few people testing them and many of the libraries require manual installation, making it a very tedious task. So, unless you really have to, try and stay away from manually compiling Python packages on Windows. Most packages are available as installable binary downloads with a bit of searching, and there are alternatives such as Anaconda that include binary packages for most important C Python packages.

If you still feel inclined to manually compile C Python packages, then there is another option, and it is generally an easier alternative. The Cygwin project (`http://cygwin.com/`) attempts to make Linux applications run natively on Windows. This is generally an easier solution than making packages work with Visual Studio.

If you do wish to take the Visual Studio path, I would like to point you towards *Chapter 14*, *Extensions in C/C++, System Calls, and C/C++ Libraries*, which covers manual writing of C/C++ extensions and some information on which Visual Studio versions you need for your Python version.

Summary

With the inclusion of packages such as `pip` and `venv`, I feel that Python 3 has become a complete package that should suit most people. Beyond legacy applications, there is no real reason not to choose Python 3 anymore. The initial Python 3 release in 2008 was definitely a bit raw compared to the well-rounded Python 2.6 version released the same year, but a lot has changed in that aspect. The last major Python 2 release was Python 2.7, which was released in 2010; within the software world, that is a very, very long time. While Python 2.7 still receives maintenance, it will not receive any of the amazing new features that Python 3 is getting—features such as Unicode strings by default, `dict` generators (Chapter 6, *Generators and Coroutines – Infinity, One Step at a Time*), and `async` methods (Chapter 7, *Async IO – Multithreading without Threads*).

After finishing this chapter, you should be able to create a clean and recreatable virtual environment and know where to look if an installation of C/C++ packages fails.

Here are the most important notes for this chapter:

- For a clean and simple environment, use `venv`. If compatibility with Python 2 is needed, use `virtualenv`.
- If C/C++ packages fail to install, look for the error about missing includes.

The next chapter covers the Python style guide, which rules are important, and why they matter. Readability is one of the most important aspects of the Python philosophy, and you will learn methods and styles for writing cleaner and more readable Python code.

2
Pythonic Syntax, Common Pitfalls, and Style Guide

The design and development of the Python programming language have always been in the hands of its original author, Guido van Rossum, in many cases lovingly referred to as the **Benevolent Dictator For Life** (**BDFL**). Even though van Rossum is thought to have a time machine (he has repeatedly answered feature requests with "I just implemented that last night": `http://www.catb.org/jargon/html/G/Guido.html`), he is still just a human and needs help with the maintenance and development of Python. To facilitate that, the **Python Enhancement Proposal** (**PEP**) process has been developed. This process allows anyone to submit a PEP with a technical specification of the feature and a rationale to defend its usefulness. After a discussion on the Python mailing lists and possibly some improvements, the BDFL will make a decision to accept or reject the proposal.

The Python style guide (`PEP 8`: `https://www.python.org/dev/peps/pep-0008/`) was once submitted as one of those PEPs, and it is has been accepted and improved regularly since. It has a lot of great and widely accepted conventions as well as a few disputed ones. Especially, the maximum line length of 79 characters is a topic of many discussions. Limiting a line to 79 characters does have some merits, however. In addition to this, while just the style guide itself does not make code Pythonic, as "The Zen of Python" (`PEP 20`: `https://www.python.org/dev/peps/pep-0020/`) elegantly says: "Beautiful is better than ugly." `PEP 8` defines how code should be formatted in an exact way, and `PEP 20` is more of a philosophy and mindset.

The common pitfalls are a list of common mistakes made, varying from beginner mistakes to advanced ones. They range from passing a list or dictionary (which are mutable) as arguments to late-binding problems in closures. An even more important issue is how to work around circular imports in a clean way.

Some of the techniques used in the examples in this chapter might be a bit too advanced for such an early chapter, but please don't worry. This chapter is about style and common pitfalls. The inner workings of the techniques used will be covered in later chapters.

We will cover the following topics in this chapter:

- Code style (`PEP 8`, `pyflakes`, `flake8`, and more)
- Common pitfalls (lists as function arguments, pass by value versus pass by reference, and inheritance behavior)

The definition of Pythonic code is highly subjective and mainly reflects the opinion of this author. When working on a project, it is more important to stay consistent with the coding styles of that project than with the coding guidelines given by Python or this book.

Code style – or what is Pythonic code?

Pythonic code—when you first hear of it, you might think it is a programming paradigm, similar to object-oriented or functional programming. While some of it could be considered as such, it is actually more of a design philosophy. Python leaves you free to choose to program in an object-oriented, procedural, functional, aspect-oriented or even logic-oriented way. These freedoms make Python a great language to write in, but as always, freedom has the drawback of requiring a lot of discipline to keep the code clean and readable. The `PEP8` standard tells us how to format code, but there is more to Pythonic code than syntax alone. That is what the Pythonic philosophy (`PEP20`) is all about, code that is:

- Clean
- Simple
- Beautiful
- Explicit
- Readable

Most of these sound like common sense, and I think they should be. There are cases however, where there is not a single obvious way to do it (unless you're Dutch, of course, as you'll read later in this chapter). That is the goal of this chapter—to learn what code is beautiful and why certain decisions have been made in the Python style guide.

Some programmers once asked Guido van Rossum whether Python would ever support braces. Since that day, braces have been available through a `__future__` import:

```
>>> from __future__ import braces
  File "<stdin>", line 1
SyntaxError: not a chance
```

Formatting strings – printf-style or str.format?

Python has supported both `printf-style` (`%`) and `str.format` for a long time, so you are most likely familiar with both already.

Within this book, `printf-style` formatting will be used for a few reasons:

- The most important reason is that it comes naturally to me. I have been using `printf` in many different programming languages for about 20 years now.
- The `printf` syntax is supported in most programming languages, which makes it familiar for a lot of people.
- While only relevant for the purposes of the examples in this book, it takes up slightly less space, requiring less formatting changes. As opposed to monitors, books have not gotten wider over the years.

In general most people recommend `str.format` these days, but it mainly comes down to preference. The `printf-style` is simpler, while the `str.format` method is more powerful.

If you wish to learn more about how `printf-style` formatting can be replaced with `str.format` (or the other way around, of course), then I recommend the PyFormat site at `https://pyformat.info/`.

PEP20, the Zen of Python

Most of the Pythonic philosophy can be explained through PEP20. Python has a nice little Easter egg to always remind you of `PEP20`. Simply type `import this` in a Python console and you will get the `PEP20` lines. To quote `PEP20`:

> *"Long time Pythoneer Tim Peters succinctly channels the BDFL's guiding principles for Python's design into 20 aphorisms, only 19 of which have been written down."*

The next few paragraphs will explain the intentions of these 19 lines.

The examples within the PEP20 section are not necessarily all identical in working, but they do serve the same purpose. Many of the examples here are fictional and serve no purpose other than explaining the rationale of the paragraph.

For clarity, let's see the output of `import this` before we begin:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Beautiful is better than ugly

While beauty is quite subjective, there are some Python style rules to adhere to: limiting line lengths, keeping statements on separate lines, splitting imports on separate lines, and so on.

In short, instead of a somewhat complex function such as this:

```
def filter_modulo(items, modulo):
   output_items = []
   for i in range(len(items)):
       if items[i] % modulo:
           output_items.append(items[i])
   return output_items
```

Or this:

```
filter_modulo = lambda i, m: [i[j] for i in range(len(i))
                              if i[j] % m]
```

Just do the following:

```
def filter_modulo(items, modulo):
    for item in items:
        if item % modulo:
            yield item
```

Simpler, easier to read, and a bit more beautiful!

These examples are not identical in results. The first two return lists whereas the last returns a generator. Generators will be discussed more thoroughly in *Chapter 6*, *Generators and Coroutines – Infinity, One Step at a Time*.

Explicit is better than implicit

Imports, arguments, and variable names are just some of the many cases where explicit code is far easier to read at the cost of a little bit more effort and/or verbosity when writing the code.

Here is an example:

```
from spam import *
from eggs import *

some_function()
```

While this saves you some typing, it becomes quite difficult to see where some_ function is defined. Is it defined in eggs? In spam? Perhaps in both modules? There are editors with advanced introspection that can help you here, but why not keep it explicit so that everyone (even when simply viewing the code online) can see what it's doing?

```
import spam
import eggs

spam.some_function()
eggs.some_function()
```

The added benefit is that we can explicitly call the function from either `spam` or `eggs` here, and everyone will have a better idea what the code does.

The same goes for functions with `*args` and `**kwargs`. They can be very useful at times, but they do have the downside of making it less obvious which arguments are valid for a function:

```
def spam(egg, *args, **kwargs):
    processed_egg = process_egg(egg, *args, **kwargs)
    return Spam(processed_egg)
```

Documentation can obviously help for cases like these and I don't disagree with the usage of `*args` and `**kwargs` in general, but it is definitely a good idea to keep at least the most common arguments explicit. Even when it requires you to repeat the arguments for a parent class, it just makes the code that much clearer. When refactoring the parent class in future, you'll know whether there are subclasses that still use some parameters.

Simple is better than complex

> "*Simple is better than complex. Complex is better than complicated.*"

The most important question to ask yourself when starting a new project is: how complex does it need to be?

For example, let's assume that we've written a small program and now we need to store a bit of data. What options do we have here?

- A full database server, such as PostgreSQL or MySQL
- A simple file system database, such as SQLite or AnyDBM
- Flat file storage, such as CSV and TSV

- Structured storage, such as JSON, YAML, or XML
- Serialized Python, such as Pickle or Marshal

All of these options have their own use cases as well as advantages and disadvantages depending on the use case:

- Are you storing a lot of data? Then full database servers and flat file storage are generally the most convenient options.
- Should it be easily portable to different systems without any package installation? That makes anything besides full database servers convenient options.
- Do we need to search the data? This is much easier using one of the database systems, both filesystem and full servers.
- Are there other applications that need to be able to edit the data? That makes universal formats such as flat file storage and the structured storage convenient options, but excludes serialized Python.

Many questions! But the most important one is: how complex does it need to be? Storing data in a `pickle` file is something you can do in three lines, while connecting to a database (even with SQLite) will be more complicated and, in many cases, not needed:

```
import pickle  # Or json/yaml
With open('data.pickle', 'wb') as fh:
    pickle.dump(data, fh, pickle.HIGHEST_PROTOCOL)
```

Versus:

```
import sqlite3
connection = sqlite3.connect('database.sqlite')
cursor = connection.cursor()
cursor.execute('CREATE TABLE data (key text, value text)')
cursor.execute('''INSERT INTO data VALUES ('key', 'value')''')
connection.commit()
connection.close()
```

These examples are far from identical, of course, as one stores a complete data object whereas the other simply stores some key/value pairs within a SQLite database. That is not the point, however. The point is that the code is far more complex while it is actually less versatile in many cases. With proper libraries, this can be simplified, but the basic premise stays the same. Simple is better than complex and if the complexity is not needed, it's better to avoid it.

Flat is better than nested

Nested code quickly becomes unreadable and hard to understand. There are no strict rules here, but generally when you have three levels of nested loops, it is time to refactor.

Just take a look the following example, which prints a list of two-dimensional matrices. While nothing is specifically wrong here, splitting it into a few more functions might make it easier to understand the purpose and easier to test:

```
def print_matrices():
    for matrix in matrices:
        print('Matrix:')
        for row in matrix:
            for col in row:
                print(col, end='')
            print()
        print()
```

The somewhat flatter version is as follows:

```
def print_row(row):
    for col in row:
        print(col, end='')

def print_matrix(matrix):
    for row in matrix:
        print_row(row)
        print()

def print_matrices(matrices):
    for matrix in matrices:
        print('Matrix:')
        print_matrix(matrix)
        print()
```

This example might be a bit convoluted, but the idea is sound. Having deeply nested code can easily become very unreadable.

Sparse is better than dense

Whitespace is generally a good thing. Yes, it will make your files longer and your code will take more space, but it can help a lot with readability if you split your code logically:

```
>>> def make_eggs(a,b):'while',['technically'];print('correct');\
...     {'this':'is','highly':'unreadable'};print(1-a+b**4/2**2)
```

```
...
>>> make_eggs(1,2)
correct
4.0
```

While technically correct, this is not all that readable. I'm certain that it would take me some effort to find out what the code actually does and what number it would print without trying it:

```
>>> def make_eggs(a, b):
...     'while', ['technically']
...     print('correct')
...     {'this': 'is', 'highly': 'unreadable'}
...     print(1 - a + ((b ** 4) / (2 ** 2)))
...
>>> make_eggs(1, 2)
correct
4.0
```

Still, this is not the best code, but at least it's a bit more obvious what is happening in the code.

Readability counts

Shorter does not always mean easier to read:

```
fib=lambda n:reduce(lambda x,y:(x[0]+x[1],x[0]),[(1,1)]*(n-2))[0]
```

Although the short version has a certain beauty in conciseness, I personally find the following far more beautiful:

```
def fib(n):
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b
```

Practicality beats purity

> *"Special cases aren't special enough to break the rules. Although practicality beats purity."*

Breaking the rules can be tempting at times, but it tends to be a slippery slope. Naturally, this applies to all rules. If your quick fix is going to break the rules, you should really try to refactor it immediately. Chances are that you won't have the time to fix it later and will regret it.

No need to go overboard though. If the solution is good enough and refactoring would be much more work, then choosing the working method might be better. Even though all of these examples pertain to imports, this guideline applies to nearly all cases.

To prevent long lines, imports can be made shorter by using a few methods, adding a backslash, adding parentheses, or just shortening the imports:

```
from spam.eggs.foo.bar import spam, eggs, extra_spam, extra_eggs,
extra_stuff  from spam.eggs.foo.bar import spam, eggs, extra_spam,
extra_eggs
```

This case can easily be avoided by just following `PEP8` (one import per line):

```
from spam.eggs.foo.bar import spam from spam.eggs.foo.bar import eggs
from spam.eggs.foo.bar import extra_spam from spam.eggs.foo.bar import
extra_eggs from spam.eggs.foo.bar import extra_stuff  from spam.eggs.
foo.bar import spam
from spam.eggs.foo.bar import eggs
from spam.eggs.foo.bar import extra_spam
from spam.eggs.foo.bar import extra_eggs
```

But what about really long imports?

```
from spam_eggs_and_some_extra_spam_stuff import
my_spam_and_eggs_stuff_which_is_too_long_for_a_line
```

Yes… even though adding a backslash for imports is generally not recommended, there are some cases where it's still the best option:

```
from spam_eggs_and_some_extra_spam_stuff \
    import my_spam_and_eggs_stuff_which_is_too_long_for_a_line
```

Errors should never pass silently

> "*Errors should never pass silently. Unless explicitly silenced.*"

To paraphrase Jamie Zawinsky: Some people, when confronted with an error, think "I know, I'll use a `try/except/pass` block." Now they have two problems.

Bare or too broad exception catching is already a bad idea. Not passing them along will make you (or some other person working on the code) wonder for ages what is happening:

```
try:
    value = int(user_input)
except:
    pass
```

If you really need to catch all errors, be very explicit about it:

```
try:
    value = int(user_input)
except Exception as e:
    logging.warn('Uncaught exception %r', e)
```

Or even better, catch it specifically and add a sane default:

```
try:
    value = int(user_input)
except ValueError:
    value = 0
```

The problem is actually even more complicated. What about blocks of code that depend on whatever is happening within the exception? For example, consider the following code block:

```
try:
    value = int(user_input)
    value = do_some_processing(value)
    value = do_some_other_processing(value)
except ValueError:
    value = 0
```

If `ValueError` is raised, which line is causing it? Is it `int(user_input)`, `do_some_processing(value)`, or `do_some_other_processing(value)`? With silent catching of the error, there is no way to know when regularly executing the code, and this can be quite dangerous. If for some reason the processing of the other functions changes, it becomes a problem to handle exceptions in this way. So, unless it was actually intended to behave like that, use this instead:

```
try:
    value = int(user_input)
except ValueError:
    value = 0
else:
    value = do_some_processing(value)
    value = do_some_other_processing(value)
```

In the face of ambiguity, refuse the temptation to guess

While guesses will work in many cases, they can bite you if you're not careful. As already demonstrated in the "explicit is better than implicit" paragraph, when having a few `from ... import *`, you cannot always be certain which module is providing you the variable you were expecting.

Ambiguity should generally be avoided, so guessing can be avoided. Clear and unambiguous code generates fewer bugs. A useful case where ambiguity is likely is function calling. Take, for example, the following two function calls:

```
spam(1, 2, 3, 4, 5)
spam(spam=1, eggs=2, a=3, b=4, c=5)
```

They could be the same, but they might also not be. It's impossible to say without seeing the function. If the function were implemented in the following way, the results would be vastly different between the two:

```
def spam(a=0, b=0, c=0, d=0, e=0, spam=1, eggs=2):
    pass
```

I'm not saying you should use keyword arguments in all cases, but if there are many arguments involved and/or hard-to-identify parameters (such as numbers), it would be a good idea. Instead of using keyword arguments, you can choose logical variable names to pass the arguments as well, as long as the meaning is clearly conveyed from the code.

For example, the following is a similar call that uses custom variable names to convey the intent:

```
a = 3
b = 4
c = 5
spam(a, b, c)
```

One obvious way to do it

> "*There should be one – and preferably only one – obvious way to do it. Although that way may not be obvious at first unless you're Dutch.*"

In general, after thinking about a difficult problem for a while, you will find that there is one solution that is clearly preferable over the alternatives. There are cases where this is not the case, however, and in that case, it can be useful if you're Dutch. The joke here is that Guido van Rossum, the BDFL and original author of Python, is Dutch (as is yours truly).

Now is better than never

> "*Now is better than never. Although never is often better than *right* now.*"

It's better to fix a problem right now than push it into the future. There are cases, however, where fixing it right away is not an option. In those cases, a good alternative can be to mark a function as deprecated instead so that there is no chance of accidentally forgetting the problem:

```
import warnings
warnings.warn('Something deprecated', DeprecationWarning)
```

Hard to explain, easy to explain

> "*If the implementation is hard to explain, it's a bad idea. If the implementation is easy to explain, it may be a good idea.*"

As always, keep things as simple as you can. While complicated code can be nice to test with, it is more prone to bugs. The simpler you can keep things, the better.

Namespaces are one honking great idea

> "*Namespaces are one honking great idea – let's do more of those!*"

Namespaces can make code a lot clearer to use. Naming them properly makes it even better. For example, what does the following line of code do?

```
load(fh)
```

Not too clear, right?

How about the version with the namespace?

```
pickle.load(fh)
```

And now we do understand.

To give an example of a namespace, the full length of which renders it impractical to use, we will take a look at the `User` class in Django. Within the Django framework, the `User` class is stored in `django.contrib.auth.models.User`. Many projects use the object in the following way:

```
from django.contrib.auth.models import User
# Use it as: User
```

While this is fairly clear, it might make someone think that the `User` class is local to the current class. Doing the following instead lets people know that it is in a different module:

```
from django.contrib.auth import models
# Use it as: models.User
```

This quickly clashes with other models' imports though, so personally I would recommend the following instead:

```
from django.contrib.auth import models as auth_models
# Use it as auth_models.User
```

Here is another alternative:

```
import django.contrib.auth.models as auth_models
# Use it as auth_models.User
```

Conclusion

Now we should have some idea of what the Pythonic ideology is about. Creating code that is:

- Beautiful
- Readable
- Unambiguous
- Explicit enough
- Not completely void of whitespace

So let's move on to some more examples of how to create beautiful, readable, and simple code using the Python style guide.

Explaining PEP8

The previous paragraphs have already shown a lot of examples using `PEP20` as a reference, but there are a few other important guidelines to note as well. The PEP8 style guide specifies the standard Python coding conventions. Simply following the PEP8 standard doesn't make your code Pythonic though, but it is most certainly a good start. Which style you use is really not that much of a concern as long as you are consistent. The only thing worse than not using a proper style guide is being inconsistent with it.

Duck typing

Duck typing is a method of handling variables by behavior. To quote Alex Martelli (one of my Python heroes, also nicknamed the MartelliBot by many):

> *"Don't check whether it IS-a duck: check whether it QUACKS-like-a duck, WALKS-like-a duck, etc, etc, depending on exactly what subset of duck-like behaviour you need to play your language-games with. If the argument fails this specific-ducklyhood-subset-test, then you can shrug, ask "why a duck?"*

In many cases, when people make a comparison such as `if spam != '':`, they are actually just looking for anything that is considered a true value. While you can compare the value to the string value `''`, you generally don't have to make it so specific. In many cases, simply doing `if spam:` is more than enough and actually functions better.

For example, the following lines of code use the value of `timestamp` to generate a filename:

```
filename = '%s.csv' % timestamp
```

Because it is named `timestamp`, one might be tempted to check whether it is actually a `date` or `datetime` object, like this:

```
import datetime
if isinstance(timestamp, (datetime.date, datetime.datetime)):
    filename = '%s.csv' % timestamp
else:
    raise TypeError(
        'Timestamp %r should be date(time) object, got %s'
        % (timestamp, type(timestamp)))
```

While this is not inherently wrong, comparing types is considered a bad practice in Python, as there is oftentimes no need for it. In Python, duck typing is preferred instead. Just try converting it to a string and don't care what it actually is. To illustrate how little difference this can make for the end result, see the following code:

```
import datetime
timestamp = datetime.date(2000, 10, 5)
filename = '%s.csv' % timestamp
print('Filename from date: %s' % filename)

timestamp = '2000-10-05'
filename = '%s.csv' % timestamp
print('Filename from str: %s' % filename)
```

As you might expect, the result is identical:

```
Filename from date: 2000-10-05.csv
Filename from str: 2000-10-05.csv
```

The same goes for converting a number to a float or an integer; instead of enforcing a certain type, just require certain features. Need something that can pass as a number? Just try to convert to `int` or `float`. Need a `file` object? Why not just check whether there is a `read` method with `hasattr`?

So, don't do this:

```
if isinstance(value, int):
```

Instead, just use the following:

```
value = int(value)
```

And instead of this:

```
import io

if isinstance(fh, io.IOBase):
```

Simply use the following line:

```
if hasattr(fh, 'read'):
```

Differences between value and identity comparisons

There are several methods of comparing objects in Python, the standard greater than and less than, equal and unequal. But there are actually a few more, and one of them is a bit special. That's the identity comparison operator: instead of using `if spam == eggs`, you use `if spam is eggs`. The big difference is that one compares the value and the other compares the identity. This sounds a little vague, but it's actually fairly simple. At least within the CPython implementation, the memory address is being compared, which means that it is one of the lightest lookups you can get. Whereas a value needs to make sure that the types are comparable and perhaps check the sub-values, the identity check just checks whether the unique identifier is the same.

If you've ever written Java, you should be familiar with this principle. In Java, a regular string comparison (`spam == eggs`) will use the identity instead of the value. To compare the value, you need to use `spam.equals(eggs)` to get the correct results.

Look at this example:

```
a = 200 + 56
b = 256
c = 200 + 57
d = 257

print('%r == %r: %r' % (a, b, a == b))
print('%r is %r: %r' % (a, b, a is b))
print('%r == %r: %r' % (c, d, c == d))
print('%r is %r: %r' % (c, d, c is d))
```

While the values are the same, the identities are different. The actual result from this code is as follows:

```
256 == 256: True
256 is 256: True
257 == 257: True
257 is 257: False
```

The catch is that Python keeps an internal array of integer objects for all integers between `-5` and `256`; that's why it works for `256` but not for `257`.

You might wonder why anyone would ever want to use `is` instead of `==`. There are multiple valid answers; depending on the case, one is correct and the other isn't. But performance can also be a very important consideration. The basic guideline is that when comparing Python singletons such as `True`, `False`, and `None`, always compare using `is`.

As for the performance consideration, think of the following example:

```
spam = range(1000000)
eggs = range(1000000)
```

When doing `spam == eggs`, this will compare every item in both lists to each other, so effectively it is doing 1,000,000 comparisons internally. Compare this with only one simple identity check when using `spam is eggs`.

To look at what Python is actually doing internally with the `is` operator, you can use the `id` function. When executing `if spam is eggs`, Python will actually execute `if id(spam) == id(eggs)` internally.

Loops

Coming from other languages, one might be tempted to use `for` loops or even `while` loops to process the items of a `list`, `tuple`, `str`, and so on. While valid, it is more complex than needed. For example, consider this code:

```
i = 0
while i < len(my_list):
    item = my_list[i]
    i += 1
    do_something(i, item)
```

Instead you can do the following:

```
for i, item in enumerate(my_list):
    do_something(i, item)
```

While this can be written even shorter, it's generally not recommended, as it does not improve readability:

```
[do_something(i, item) for i, item in enumerate(my_list)]
```

The last option might be clear to some but not all. Personally, I prefer to limit the usage of list comprehensions, dict comprehensions, and map and filter statements for when the result is actually being stored.

For example:

```
spam_items = [x for x in items if x.startswith('spam_')]
```

But still, only if it doesn't hurt the readability of the code.

Consider this bit of code:

```
eggs = [is_egg(item) or create_egg(item) for item in list_of_items if
egg and hasattr(egg, 'egg_property') and isinstance(egg, Egg)]  eggs =
[is_egg(item) or create_egg(item) for item in list_of_items
        if egg and hasattr(egg, 'egg_property')
        and isinstance(egg, Egg)]
```

Instead of putting everything in the list comprehension, why not split it into a few functions?

```
def to_egg(item):
    return is_egg(item) or create_egg(item)

def can_be_egg(item):
    has_egg_property = hasattr(egg, 'egg_property')
    is_egg_instance = isinstance(egg, Egg)
    return egg and has_egg_property and is_egg_instance

eggs = [to_egg(item) for item in list_of_items if can_be_egg(item)]
eggs = [to_egg(item) for item in list_of_items if
        can_be_egg(item)]
```

While this code is a bit longer, I would personally argue that it's more readable this way.

Maximum line length

Many Python programmers think 79 characters is too constricting and just keep the lines longer. While I am not going to argue for 79 characters specifically, setting a low and fixed limit such as 79 or 99 is a good idea. While monitors get wider and wider, limiting your lines can still help a lot with readability and it allows you to put multiple files next to each other. It's a regular occurrence for me to have four Python files opened next to each other. If the line width were more than 79 characters, that simply wouldn't fit.

The PEP8 guide tells us to use backslashes in cases where lines get too long. While I agree that backslashes are preferable over long lines, I still think they should be avoided if possible. Here's an example from PEP8:

```
with open('/path/to/some/file/you/want/to/read') as file_1, \
        open('/path/to/some/file/being/written', 'w') as file_2:
    file_2.write(file_1.read())
```

Instead of using backslashes, I would reformat it like this:

```
filename_1 = '/path/to/some/file/you/want/to/read'
filename_2 = '/path/to/some/file/being/written'
```

```
with open(filename_1) as file_1, open(filename_2, 'w') as file_2:
    file_2.write(file_1.read())
```

Or perhaps the following:

```
filename_1 = '/path/to/some/file/you/want/to/read'
filename_2 = '/path/to/some/file/being/written'
with open(filename_1) as file_1:
    with open(filename_2, 'w') as file_2:
        file_2.write(file_1.read())
```

Not always an option, of course, but it's a good consideration to keep the code short and readable. It actually gives a bonus of adding more information to the code. If, instead of `filename_1`, you use a name that conveys the goal of the filename, it immediately becomes clearer what you are trying to do.

Verifying code quality, pep8, pyflakes, and more

There are many tools for checking code quality in Python. The simplest ones, such as `pep8`, just validate a few simple `PEP8` errors. The more advanced ones, such as `pylint`, do advanced introspections to detect potential bugs in otherwise working code. A large portion of what `pylint` offers is a bit over the top for many projects, but still interesting to look at.

flake8

The `flake8` tool combines pep8, pyflakes, and McCabe to set up a quality standard for code. The `flake8` tool is one of the most important packages for maintaining code quality in my packages. All the packages that I maintain have a 100% `flake8` compliance requirement. It does not promise readable code, but at least it requires a certain level of consistency, which is very important when writing on a project with multiple programmers.

Pep8

One of the simplest tools used to check the quality of Python code is the `pep8` package. It doesn't check everything that is in the PEP8 standard, but it goes a long way and is still updated regularly to add new checks. Some of the most important things checked by `pep8` are as follows:

- Indentation, while Python will not check how many spaces you use to indent, it does not help with the readability of your code
- Missing whitespace, such as `spam=123`

- Too much whitespace, such as `def eggs(spam = 123):`
- Too many or too few blank lines
- Too long lines
- Syntax and indentation errors
- Incorrect and/or superfluous comparisons (`not in`, `is not`, `if spam is True`, and type comparisons without `isinstance`)

The conclusion is that the `pep8` tool helps a lot with testing whitespace and some of the more common styling issues, but it is still fairly limited.

pyflakes

This is where pyflakes comes in. pyflakes is a bit more intelligent than `pep8` and warns you about style issues such as:

- Unused imports
- Wildcard imports (`from module import *`)
- Incorrect `__future__` imports (after other imports)

But more importantly, it warns about potential bugs, such as the following:

- Redefinitions of names that were imported
- Usage of undefined variables
- Referencing variables before assignment
- Duplicate argument names
- Unused local variables

The last bit of PEP8 is covered by the pep8-naming package. It makes sure that your naming is close to the standard dictated by PEP8:

- Class names as *CapWord*
- Function, variable, and argument names all in lowercase
- Constants as full uppercase and being treated as constants
- The first argument of instance methods and class methods as *self* and *cls*, respectively

McCabe

Lastly, there is the McCabe complexity. It checks the complexity of code by looking at the **Abstract Syntax Tree** (**AST**). It finds out how many lines, levels, and statements are there and warns you if your code has more complexity than a preconfigured threshold. Generally, you will use McCabe through `flake8`, but a manual call is possible as well. Using the following code:

```
def spam():
    pass


def eggs(matrix):
    for x in matrix:
        for y in x:
            for z in y:
                print(z, end='')
            print()
        print()
```

McCabe will give us the following output:

```
# pip install mccabe
...
# python -m mccabe cabe_test.py 1:1: 'spam' 1
5:1: 'eggs' 4
```

Your maximum threshold is configurable, of course, but the default is 10. The McCabe test returns a number that is influenced by parameters such as the size of a function, the nested depths, and a few others. If your function reaches 10, it might be time to refactor the code.

flake8

All of this combined is `flake8`, a tool that combines these tools and outputs a single report. Some of the warnings generated by `flake8` might not fit your taste, so each and every one of the checks can be disabled, both per file and for the entire project if needed. For example, I personally disable `W391` for all my projects, which warns about blank lines at the end of a file. This is something I find useful while working on code so that I can easily jump to the end of the file and start writing code instead of having to append a few lines first.

In general, before committing your code and/or putting it online, just run `flake8` from your source directory to check everything recursively.

Here is a demonstration with some poorly formatted code:

```
def spam(a,b,c):
    print(a,b+c)

def eggs():
    pass
```

It results in the following:

```
# pip install flake8
...
# flake8 flake8_test.py
flake8_test.py:1:11: E231 missing whitespace after ','
flake8_test.py:1:13: E231 missing whitespace after ','
flake8_test.py:2:12: E231 missing whitespace after ','
flake8_test.py:2:14: E226 missing whitespace around arithmetic operator
flake8_test.py:4:1: E302 expected 2 blank lines, found 1
```

Pylint

`pylint` is a far more advanced—and in some cases better—code quality checker. The power of `pylint` does come with a few drawbacks, however. Whereas `flake8` is a really fast, light, and safe quality check, `pylint` has far more advanced introspection and is much slower for this reason. In addition, `pylint` will most likely give you a large number of warnings, which are irrelevant or even wrong. This could be seen as a flaw in `pylint`, but it's actually more of a restriction of passive code analysis. Tools such as `pychecker` actually load and execute your code. In many cases, this is safe, but there are cases where it is not. Just think of what could happen when executing a command that deletes files.

While I have nothing against `pylint`, in general I find that most important problems are handled by `flake8`, and others can easily be avoided with some proper coding standards. It can be a very useful tool if configured correctly, but without configuration, it is very verbose.

Common pitfalls

Python is a language meant to be clear and readable without any ambiguities and unexpected behaviors. Unfortunately, these goals are not achievable in all cases, and that is why Python does have a few corner cases where it might do something different than what you were expecting.

This section will show you some issues that you might encounter when writing Python code.

Scope matters!

There are a few cases in Python where you might not be using the scope that you are actually expecting. Some examples are when declaring a class and with function arguments.

Function arguments

The following example shows a case that breaks due to a careless choice in default parameters:

```
def spam(key, value, list_=[], dict_={}):
    list_.append(value)
    dict_[key] = value

    print('List: %r' % list_)
    print('Dict: %r' % dict_)

spam('key 1', 'value 1')
spam('key 2', 'value 2')
```

You would probably expect the following output:

```
List: ['value 1']
Dict: {'key 1': 'value 1'}
List: ['value 2']
Dict: {'key 2': 'value 2'}
```

But it's actually this:

```
List: ['value 1']
Dict: {'key 1': 'value 1'}
List: ['value 1', 'value 2']
Dict: {'key 1': 'value 1', 'key 2': 'value 2'}
```

The reason is that `list_` and `dict_` are actually shared between multiple calls. The only time this is actually useful is if you are doing something hacky, so please avoid using mutable objects as default parameters in a function.

The safe alternative of the same example is as follows:

```
def spam(key, value, list_=None, dict_=None):
    if list_ is None:
```

```
            list_ = []

        if dict_ is None:
    dict_ = {}

        list_.append(value)
        dict_[key] = value
```

Class properties

The problem also occurs when defining classes. It is very easy to mix class attributes and instance attributes. Especially when coming from other languages such as C#, this can be confusing. Let's illustrate it:

```
class Spam(object):
    list_ = []
    dict_ = {}

    def __init__(self, key, value):
        self.list_.append(value)
        self.dict_[key] = value

        print('List: %r' % self.list_)
        print('Dict: %r' % self.dict_)


Spam('key 1', 'value 1')
Spam('key 2', 'value 2')
```

As with the function arguments, the list and dictionaries are shared. So, the output is as follows:

```
List: ['value 1']
Dict: {'key 1': 'value 1'}
List: ['value 1', 'value 2']
Dict: {'key 1': 'value 1', 'key 2': 'value 2'}
```

A better alternative is to initialize the mutable objects within the `__init__` method of the class. This way, they are not shared between instances:

```
class Spam(object):
    def __init__(self, key, value):
        self.list_ = [key]
        self.dict_ = {key: value}

        print('List: %r' % self.list_)
        print('Dict: %r' % self.dict_)
```

Another important thing to note when dealing with classes is that a class property will be inherited, and that's where things might prove to be confusing. When inheriting, the original properties will stay (unless overwritten), even in subclasses:

```
 >>> class A(object):
...     spam = 1


>>> class B(A):
...     pass


Regular inheritance, the spam attribute of both A and B are 1 as
you would expect.
>>> A.spam
1
>>> B.spam
1

Assigning 2 to A.spam now modifies B.spam as well
>>> A.spam = 2

>>> A.spam
2
>>> B.spam
2
```

While this is to be expected due to inheritance, someone else using the class might not suspect the variable to change in the meantime. After all, we modified `A.spam`, not `B.spam`.

There are two easy ways to prevent this. It is obviously possible to simply set `spam` for every class separately. But the better solution is never to modify class properties. It's easy to forget that the property will change in multiple locations, and if it has to be modifiable anyway, it's usually better to put it in an instance variable instead.

Modifying variables in the global scope

A common problem when accessing variables from the global scope is that setting a variable makes it local, even when accessing the global variable.

This works:

```
 >>> def eggs():
...     print('Spam: %r' % spam)


>>> eggs()
Spam: 1
```

But the following does not:

```
 >>> spam = 1



>>> def eggs():
...     spam += 1
...     print('Spam: %r' % spam)


>>> eggs()
Traceback (most recent call last):
    ...
UnboundLocalError: local variable 'spam' referenced before assignment
```

The problem is that `spam += 1` actually translates to `spam = spam + 1`, and anything containing `spam =` makes the variable local to your scope. Since the local variable is being assigned at that point, it has no value yet and you are trying to use it. For these cases, there is the `global` statement, although I would really recommend that you avoid globals altogether.

Overwriting and/or creating extra built-ins

While it can be useful in some cases, generally you will want to avoid overwriting global functions. The `PEP8` convention for naming your functions—similar to built-in statements, functions, and variables—is to use a trailing underscore.

So, do not use this:

```
list = [1, 2, 3]
```

Instead, use the following:

```
list_ = [1, 2, 3]
```

For lists and such, this is just a good convention. For statements such as `from`, `import`, and `with`, it's a requirement. Forgetting about this can lead to very confusing errors:

```
>>> list = list((1, 2, 3))
>>> list
[1, 2, 3]

>>> list((4, 5, 6))
Traceback (most recent call last):
    ...
TypeError: 'list' object is not callable

>>> import = 'Some import'
Traceback (most recent call last):
    ...
SyntaxError: invalid syntax
```

If you actually want to define a built-in that is available everywhere, it's possible. For debugging purposes, I've been known to add this code to a project while developing:

```
import builtins
import inspect
import pprint
import re


def pp(*args, **kwargs):
    '''PrettyPrint function that prints the variable name when
    available and pprints the data'''
    name = None
    # Fetch the current frame from the stack
    frame = inspect.currentframe().f_back
    # Prepare the frame info
    frame_info = inspect.getframeinfo(frame)

    # Walk through the lines of the function
    for line in frame_info[3]:
        # Search for the pp() function call with a fancy regexp
        m = re.search(r'\bpp\s*\(\s*([^)]*)\s*\)', line)
        if m:
            print('# %s:' % m.group(1), end=' ')
```

```
            break

    pprint.pprint(*args, **kwargs)

builtins.pf = pprint.pformat
builtins.pp = pp
```

Much too hacky for production code, but it is still useful when working on a large project where you need print statements to debug. Alternative (and better) debugging solutions can be found in Chapter 11, *Debugging – Solving the Bugs*.

The usage is quite simple:

```
x = 10
pp(x)
```

Here is the output:

```
# x: 10
```

Modifying while iterating

At one point or another, you will run into this problem: while iterating through mutable objects such as lists, dicts, or sets, you cannot modify them. All of these result in a `RuntimeError` telling you that you cannot modify the object during iteration:

```
dict_ = {'spam': 'eggs'}
list_ = ['spam']
set_ = {'spam', 'eggs'}

for key in dict_:
    del dict_[key]

for item in list_:
    list_.remove(item)

for item in set_:
    set_.remove(item)
```

This can be avoided by copying the object. The most convenient option is by using the `list` function:

```
dict_ = {'spam': 'eggs'}
list_ = ['spam']
set_ = {'spam', 'eggs'}

for key in list(dict_):
```

```
        del dict_[key]

    for item in list(list_):
        list_.remove(item)

    for item in list(set_):
        set_.remove(item)
```

Catching exceptions – differences between Python 2 and 3

With Python 3, catching an exception and storing it has been made more obvious with the `as` statement. The problem is that many people are still used to the `except Exception, variable` syntax, which doesn't work anymore. Luckily, the Python 3 syntax has been backported to Python 2, so now you can use the following syntax everywhere:

```
try:
    ... # do something here
except (ValueError, TypeError) as e:
    print('Exception: %r' % e)
```

Another important difference is that Python 3 makes this variable local to the exception scope. The result is that you need to declare the exception variable before the `try/except` block if you want to use it later:

```
def spam(value):
    try:
        value = int(value)
    except ValueError as exception:
        print('We caught an exception: %r' % exception)

    return exception


spam('a')
```

You might expect that since we get an exception here, this works; but actually, it doesn't, because `exception` does not exist at the point of the `return` statement.

The actual output is as follows:

```
We caught an exception: ValueError("invalid literal for int() with
base 10: 'a'",)
Traceback (most recent call last):
  File "test.py", line 14, in <module>
```

```
    spam('a')
  File "test.py", line 11, in spam
    return exception
UnboundLocalError: local variable 'exception' referenced before
assignment
```

Personally I would argue that the preceding code is broken in any case: what if there isn't an exception somehow? It would have raised the same error. Luckily, the fix is simple; just write the value to a variable outside of the scope. One important thing to note here is that you explicitly need to save the variable to the parent scope. This code does not work either:

```
def spam(value):
    exception = None
    try:
        value = int(value)
    except ValueError as exception:
        print('We caught an exception: %r' % exception)

    return exception
```

We really need to save it explicitly because Python 3 automatically deletes anything saved with `as variable` at the end of the `except` statements. The reason for this is that exceptions in Python 3 contain a `__traceback__` attribute. Having this attribute makes it much more difficult for the garbage collector to handle as it introduces a recursive self-referencing cycle (*exception -> traceback -> exception -> traceback… ad nauseum*). To solve this, Python essentially does the following:

```
exception = None
try:
    value = int(value)
except ValueError as exception:
    try:
        print('We caught an exception: %r' % exception)
    finally:
        del exception
```

The solution is simple enough—luckily—but you should keep in mind that this can introduce memory leaks into your program. The Python garbage collector is smart enough to understand that the variables are not visible anymore and will delete it eventually, but it can take a lot more time. How the garbage collection actually works is covered in *Chapter 12, Performance – Tracking and Reducing Your Memory and CPU Usage*. Here is the working version of the code:

```
def spam(value):
    exception = None
```

```
    try:
        value = int(value)
    except ValueError as e:
        exception = e
        print('We caught an exception: %r' % exception)

    return exception
```

Late binding – be careful with closures

Closures are a method of implementing local scopes in code. They make it possible to locally define variables without overriding variables in the parent (or global) scope and hide the variables from the outside scope later. The problem with closures in Python is that Python tries to bind its variables as late as possible for performance reasons. While generally useful, it does have some unexpected side effects:

```
eggs = [lambda a: i * a for i in range(3)]

for egg in eggs:
    print(egg(5))
```

The expected result? Should be something along the lines of this, right?

```
0
5
10
```

No, unfortunately not. This is similar to how class inheritance works with properties. Due to late binding, the variable `i` gets called from the surrounding scope at call time, and not when it's actually defined.

The actual result is as follows:

```
10
10
10
```

So what to do instead? As with the cases mentioned earlier, the variable needs to be made local. One alternative is to force immediate binding by currying the function with `partial`:

```
import functools


eggs = [functools.partial(lambda i, a: i * a, i) for i in
range(3)]
```

```
for egg in eggs:
    print(egg(5))
```

A better solution would be to avoid binding problems altogether by not introducing extra scopes (the `lambda`), that use external variables. If both `i` and `a` were specified as arguments to `lambda`, this will not be a problem.

Circular imports

Even though Python is fairly tolerant towards circular imports, there are some cases where you will get errors.

Let's assume we have two files.

`eggs.py`:

```
from spam import spam


def eggs():
    print('This is eggs')
    spam()
```

`spam.py`:

```
from eggs import eggs


def spam():
    print('This is spam')


if __name__ == '__main__':
    eggs()
```

Running `spam.py` will result in a circular `import` error:

```
Traceback (most recent call last):
  File "spam.py", line 1, in <module>
    from eggs import eggs
  File "eggs.py", line 1, in <module>
    from spam import spam
  File "spam.py", line 1, in <module>
    from eggs import eggs
ImportError: cannot import name 'eggs'
```

There are a few ways to work around this. Restructuring the code is usually the best to go around, but the best solution depends on the problem. In the preceding case, it can be solved easily. Just use module imports instead of function imports (which I recommend regardless of circular imports).

`eggs.py`:

```
import spam


def eggs():
    print('This is eggs')
    spam.spam()
```

`spam.py`:

```
import eggs


def spam():
    print('This is spam')


if __name__ == '__main__':
    eggs.eggs()
```

An alternative solution is to move the imports within the functions so that they occur at runtime. This is not the prettiest solution but it does the trick in many cases.

`eggs.py`:

```
def eggs():
    from spam import spam
    print('This is eggs')
    spam()
```

`spam.py`:

```
def spam():
    from eggs import eggs
    print('This is spam')


if __name__ == '__main__':
    eggs()
```

Lastly there is the solution of moving the imports below the code that actually uses them. This is generally not recommended because it can make it non-obvious where the imports are, but I still find it preferable to having the `import` within the function calls.

`eggs.py`:

```
    def eggs():
        print('This is eggs')
        spam()


    from spam import spam
```

`spam.py`:

```
    def spam():
        print('This is spam')


    from eggs import eggs


    if __name__ == '__main__':
        eggs()
```

And yes, there are still other solutions such as dynamic imports. One example of this is how the Django `ForeignKey` fields support strings instead of actual classes. But those are generally a really bad idea to use since they will be checked only at runtime. Because of this, bugs will introduce themselves only when executing any code that uses it instead of when modifying the code. So please try to avoid these whenever possible, or make sure you add proper automated tests to prevent unexpected bugs. Especially when they cause circular imports internally, they become an enormous pain to debug.

Import collisions

One problem that can be extremely confusing is having colliding imports—multiple packages/modules with the same name. I have had more than a few bug reports on my packages where, for example, people tried to use my `numpy-stl` project, which resides in a package named `stl` from a test file named `stl.py`. The result: it was importing itself instead of the `stl` package. While this case is difficult to avoid, at least within packages, a relative import is generally a better option. This is because it also tells other programmers that the import comes from the local scope instead of another package. So, instead of writing `import spam`, write `from . import spam`. This way, the code will always load from the current package instead of any global package that happens to have the same name.

In addition to this there is also the problem of packages being incompatible with each other. Common names might be used by several packages, so be careful when installing those packages. When in doubt, just create a new virtual environment and try again. Doing this can save you a lot of debugging.

Summary

This chapter showed us what the Pythonic philosophy is all about and explained to us what the Zen of Python is all about. While code style is highly personal, Python has a few, very helpful guidelines that at least keep people mostly on the same page and style. In the end, we are all consenting adults; everyone has the right to write code as he/she sees fit. But I do request you. Please read through the style guides and try to adhere to them unless you have a really good reason not to.

With all that power comes great responsibility, and so do a few pitfalls, though there aren't too many. Some are tricky enough to fool me regularly and I've been writing Python for a long time! Python improves all the time though. Many pitfalls have been taken care of since Python 2, but some will always remain. For example, recursive imports and definitions can easily bite you in most languages that support them, but that doesn't mean we'll stop trying to improve Python.

A good example of the improvements in Python over the years is the collections module. It contains many useful collections that have been added by users because there was a need. Most of them are actually implemented in pure Python, and because of that, they are easy enough to be read by anyone. Understanding might take a bit more effort, but I truly believe that if you make it to the end of this book, you will have no problem understanding what the collections do. Fully understanding how the internals work is something I cannot promise though; some parts of that go more towards generic computer science than Python mastery.

The next chapter will show you some of the collections available in Python and how they are constructed internally. Even though you are undoubtedly familiar with collections such as lists and dictionaries, you might not be aware of the performance characteristics involved with some of the operations. If some of the examples in this chapter were less than clear, you don't have to worry. The next chapter will at least revisit some of them, and more will come in later chapters.

3

Containers and Collections – Storing Data the Right Way

Python comes bundled with several very useful collections, a few of which are basic Python collection data types. The rest are advanced combinations of these types. In this chapter, we will explain some of these collections, how to use them, and the pros and cons of each of them.

Before we can properly discuss data structures and the related performance, a basic understanding of time complexity (and specifically the big O notation) is required. No need to worry! The concept is really simple, but without it, we cannot easily explain the performance characteristics of operations.

Once the big O notation is clear, we will discuss the basic data structures:

- `list`
- `dict`
- `set`
- `tuple`

Building on the basic data structures, we will continue with more advanced collections, such as the following:

- Dictionary-like types:
  - `ChainMap`
  - `Counter`
  - `Defaultdict`
  - `OrderedDict`

- List types:
  - `Deque`
  - `Heapq`
- Tuple types:
  - `NamedTuple`
- Other types:
  - `Enum`

Time complexity – the big O notation

Before we can begin with this chapter, there is a simple notation that you need to understand. This chapter heavily uses the big O notation to indicate the time complexity for an operation. Feel free to skip this paragraph if you are already familiar with this notation. While this notation sounds really complicated, the concept is actually quite simple.

When we say that a function takes `O(1)` time, it means that it generally only takes `1` step to execute. Similarly, a function with `O(n)` would take `n` steps to execute, where `n` is generally the size of the object. This time complexity is just a basic indication of what to expect when executing the code, as it is generally what matters most.

The purpose of this system is to indicate the approximate performance of an operation; this is separate from code speed but it is still relevant. A piece of code that executes a single step `1000` times faster but needs `O(2**n)` steps to execute will still be slower than another version of it that takes only `O(n)` steps for a value of n equal to `10` or more. This is because `2**n` for `n=10` is `2**10=1024`, that is, 1,024 steps to execute the same code. This makes choosing the right algorithm very important. Even though `C` code is generally faster than Python, if it uses the wrong algorithm, it won't help at all.

For example, suppose you have a list of `1000` items and you walk through them. This will take `O(n)` time because there are `n=1000` items. Checking whether or not an item exists in the list takes `O(n)`, so that's 1,000 steps. Doing this 100 times will take you `100*O(n) = 100 * 1000 = 100,000` steps. When you compare this to a `dict`, where checking whether the item exists or not takes `only O(1)` time the difference is huge. With a `dict`, it would be `100*O(1) = 100 * 1 = 100` steps. So, using a `dict` instead of a `list` will be roughly 1,000 times faster for an object with 1,000 items:

```
n = 1000
a = list(range(n))
b = dict.fromkeys(range(n))
```

```
for i in range(100):
    i in a  # takes n=1000 steps
    i in b  # takes 1 step
```

To illustrate `O(1)`, `O(n)`, and `O(n**2)` functions:

```
def o_one(items):
    return 1  # 1 operation so O(1)

def o_n(items):
    total = 0
    # Walks through all items once so O(n)
    for item in items:
        total += item
    return total

def o_n_squared(items):
    total = 0
    # Walks through all items n*n times so O(n**2)
    for a in items:
        for b in items:
            total += a * b
    return total

n = 10
items = range(n)
o_one(items)  # 1 operation
o_n(items)  # n = 10 operations
o_n_squared(items)  # n*n = 10*10 = 100 operations
```

It should be noted that the big O in this chapter is about the average case and not the worst case. In some cases, they can be much worse, but those are rare enough to be ignored for the general case.

Core collections

Before we can look at the more advanced combined collections later in this chapter, you need to understand the workings of the core Python collections. This is not just about the usage, however; it is also about the time complexities involved, which can strongly affect how your application will behave as it grows. If you are well versed with the time complexities of these objects and know the possibilities of Python 3's tuple packing and unpacking by heart, then feel free to jump to the *Advanced collections* section.

list – a mutable list of items

The `list` is most likely the container structure that you've used most in Python. It is simple in its usage, and for most cases, it exhibits great performance.

While you may already be well versed with the usage of list, you might not be aware of the time complexities of the `list` object. Luckily, many of the time complexities of `list` are very low; `append`, `get`, `set`, and `len` all take `O(1)` time—the best possible. However, you might not be aware of the fact that `remove` and `insert` have `O(n)` time complexity. So, to delete a single item out of 1,000 items, Python will have to walk-through 1,000 items. Internally, the `remove` and `insert` operations execute something along these lines:

```
>>> def remove(items, value):
...     new_items = []
...     found = False
...     for item in items:
...         # Skip the first item which is equal to value
...         if not found and item == value:
...             found = True
...             continue
...         new_items.append(item)
...
...     if not found:
...         raise ValueError('list.remove(x): x not in list')
...
...     return new_items


>>> def insert(items, index, value):
...     new_items = []
...     for i, item in enumerate(items):
...         if i == index:
...             new_items.append(value)
...         new_items.append(item)
...     return new_items

>>> items = list(range(10))
>>> items
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> items = remove(items, 5)
>>> items
[0, 1, 2, 3, 4, 6, 7, 8, 9]

>>> items = insert(items, 2, 5)
>>> items
[0, 1, 5, 2, 3, 4, 6, 7, 8, 9]
```

To remove or insert a single item from/into the list, Python needs to copy the entire list, which is especially heavy with larger lists. When executing this only once, it is of course not all that bad. But when executing a large number of deletions, a `filter` or `list` comprehension is a much faster solution because, if properly structured, it needs to copy the list only once. For example, suppose we wish to remove a specific set of numbers from the list. We have quite a few options for this. The first is a solution using `remove`, followed by a list comprehension, and then comes a `filter` statement. *Chapter 4*, *Functional Programming – Readability Versus Brevity*, will explain `list` comprehensions and the `filter` statement in more detail. But first, let's check out the example:

```
>>> primes = set((1, 2, 3, 5, 7))

# Classic solution
>>> items = list(range(10))
>>> for prime in primes:
...     items.remove(prime)
>>> items
[0, 4, 6, 8, 9]

# List comprehension
>>> items = list(range(10))
>>> [item for item in items if item not in primes]
[0, 4, 6, 8, 9]

# Filter
>>> items = list(range(10))
>>> list(filter(lambda item: item not in primes, items))
[0, 4, 6, 8, 9]
```

The latter two are much faster for large lists of items. This is because the operations are much faster. To compare using `n=len(items)` and `m=len(primes)`, the first takes `O(m*n)=5*10=50` operations, whereas the latter two take `O(n*1)=10*1=10` operations.

The first method is actually slightly better than that since `n` decreases during the loop. So, it's effectively `10+9+8+7+6=40`, but this is an effect that is negligible enough to ignore. In the case of `n=1000`, that would be the difference between `1000+999+998+997+996=4990` and `5*1000=5000`, which is negligible in most cases.

Of course, `min`, `max`, and `in` all take `O(n)` as well, but that is expected for a structure that is not optimized for these types of lookups.

They can be implemented like this:

```
>>> def in_(items, value):
...     for item in items:
...         if item == value:
...             return True
...     return False

>>> def min_(items):
...     current_min = items[0]
...     for item in items[1:]:
...         if current_min > item:
...             current_min = item
...     return current_min

>>> def max_(items):
...     current_max = items[0]
...     for item in items[1:]:
...         if current_max < item:
...             current_max = item
...     return current_max

>>> items = range(5)
>>> in_(items, 3)
True
```

```
>>> min_(items)
0
>>> max_(items)
4
```

With these examples, it's obvious as well that the `in` operator could work `O(1)` if you're lucky, but we count it as `O(n)` because it might not exist, in which case all values need to be checked.

dict – unsorted but a fast map of items

The `dict` has to be at least among the top three container structures you use in Python. It's fast, simple to use, and very effective. The average time complexity is exactly as you would expect—`O(1)` for `get`, `set`, and `del`—but there are cases where this is not true. The way a `dict` works is by converting the key to a hash using the `hash` function (calls the `__hash__` function of an object) and storing it in a hash table. There are two problems with hash tables, however. The first and the most obvious is that the items will be sorted by hash, which appears at random in most cases. The second problem with hash tables is that they can have hash collisions, and the result of a hash collision is that in the worst case, all the former operations can take `O(n)` instead. Hash collisions are not all that likely to occur, but they can occur, and if a large `dict` performs subpar, that's the place to look.

Let's see how this actually works in practice. For the sake of this example, I will use the simplest hashing algorithm I can think of, which is the most significant digit of a number. So, for the case of `12345`, it will return `1`, and for `56789`, it will return `5`:

```
>>> def most_significant(value):
...     while value >= 10:
...         value //= 10
...     return value

>>> most_significant(12345)
1
>>> most_significant(99)
9
>>> most_significant(0)
0
```

Now we will emulate a `dict` using a `list` of lists with this hashing method. We know that our hashing method can only return numbers from `0` to `9`, so we need only 10 buckets in our list. Now we will add a few values and show how the spam in eggs might work:

```
>>> def add(collection, key, value):
...     index = most_significant(key)
...     collection[index].append((key, value))

>>> def contains(collection, key):
...     index = most_significant(key)
...     for k, v in collection[index]:
...         if k == key:
...             return True
...     return False

# Create the collection of 10 lists
>>> collection = [[], [], [], [], [], [], [], [], [], []]

# Add some items, using key/value pairs
>>> add(collection, 123, 'a')
>>> add(collection, 456, 'b')
>>> add(collection, 789, 'c')
>>> add(collection, 101, 'c')

# Look at the collection
>>> collection
[[], [(123, 'a'), (101, 'c')], [], [],
 [(456, 'b')], [], [], [(789, 'c')], [], []]

# Check if the contains works correctly
>>> contains(collection, 123)
True
>>> contains(collection, 1)
False
```

This code is obviously not identical to the `dict` implementation, but it is actually quite similar internally. Because we can just get item `1` for a value of `123` by simple indexing, we have only `O(1)` lookup costs in the general case. However, since both keys, `123` and `101`, are within the `1` bucket, the runtime can actually increase to `O(n)` in the worst case where all keys have the same hash. That is what we call a hash collision.

To debug hash collisions, you can use the `hash()` function paired with the counter collection, discussed in the *counter – keeping track of the most occurring elements* section.

In addition to the hash collision performance problem, there is another behavior that might surprise you. When deleting items from a dictionary, it won't actually resize the dictionary in memory yet. The result is that both copying and iterating the entire dictionary take `O(m)` time (where m is the maximum size of the dictionary); `n`, the current number of items is not used. So, if you add 1,000 items to a `dict` and remove 999, iterating and copying will still take 1,000 steps. The only way to work around this issue is by recreating the dictionary, which is something that both the `copy` and `insert` operations will perform internally. Note that recreation during an `insert` operation is not guaranteed and depends on the number of free slots available internally.

set – like a dict without values

A `set` is a structure that uses the hash method to get a unique collection of values. Internally, it is very similar to a `dict`, with the same hash collision problem, but there are a few handy features of set that need to be shown:

```
# All output in the table below is generated using this function
>>> def print_set(expression, set_):
...     'Print set as a string sorted by letters'
...     print(expression, ''.join(sorted(set_)))

>>> spam = set('spam')
>>> print_set('spam:', spam)
spam: amps

>>> eggs = set('eggs')
>>> print_set('eggs:', spam)
eggs: amps
```

The first few are pretty much as expected. At the operators, it gets interesting.

Expression	Output	Explanation
`spam`	`amps`	All unique items. A `set` doesn't allow for duplicates.
`eggs`	`egs`	
`spam & eggs`	`s`	Every item in both.
`spam | eggs`	`aegmps`	Every item in either or both.
`spam ^ eggs`	`aegmp`	Every item in either but not in both.
`spam - eggs`	`amp`	Every item in the first but not the latter.
`eggs - spam`	`eg`	
`spam > eggs`	`False`	True if every item in the latter is in the first.
`eggs > spam`	`False`	
`spam > sp`	`True`	
`spam < sp`	`False`	True if every item in the first is contained in the latter.

One useful example for `set` operations is calculating the differences between two objects. For example, let's assume we have two lists:

- `current_users`: The current users in a group
- `new_users`: The new list of users in a group

In permission systems, this is a very common scenario—mass adding and/or removing users from a group. Within many permission databases, it's not easily possible to set the entire list at once, so you need a list to insert and a list to delete. This is where `set` comes in really handy:

```
The set function takes a sequence as argument so the double ( is
required.
>>> current_users = set((
...     'a',
...     'b',
...     'd',
... ))

>>> new_users = set((
...     'b',
...     'c',
...     'd',
...     'e',
```

```
... ))

>>> to_insert = new_users - current_users
>>> sorted(to_insert)
['c', 'e']
>>> to_delete = current_users - new_users
>>> sorted(to_delete)
['a']
>>> unchanged = new_users & current_users
>>> sorted(unchanged)
['b', 'd']
```

Now we have lists of all users who were added, removed, and unchanged. Note that `sorted` is only needed for consistent output, since a `set`, similar to a `dict`, has no predefined sort order.

tuple – the immutable list

A `tuple` is an object that you use very often without even noticing it. When you look at it initially, it seems like a useless data structure. It's like a list that you can't modify, so why not just use a `list`? There are a few cases where a `tuple` offers some really useful functionalities that a `list` does not.

Firstly, they are hashaable. This means that you can use a `tuple` as a key in a `dict`, which is something a `list` can't do:

```
>>> spam = 1, 2, 3
>>> eggs = 4, 5, 6

>>> data = dict()
>>> data[spam] = 'spam'
>>> data[eggs] = 'eggs'

>>> import pprint  # Using pprint for consistent and sorted output
>>> pprint.pprint(data)
{(1, 2, 3): 'spam', (4, 5, 6): 'eggs'}
```

However, it can actually be more than simple numbers. As long as all elements of a `tuple` are hashable, it will work. This means that you can use nested tuples, strings, numbers, and anything else for which the `hash()` function returns a consistent result:

```
>>> spam = 1, 'abc', (2, 3, (4, 5)), 'def'
>>> eggs = 4, (spam, 5), 6


>>> data = dict()
>>> data[spam] = 'spam'
>>> data[eggs] = 'eggs'
>>> import pprint  # Using pprint for consistent and sorted output
>>> pprint.pprint(data)
{(1, 'abc', (2, 3, (4, 5)), 'def'): 'spam',
 (4, ((1, 'abc', (2, 3, (4, 5)), 'def'), 5), 6): 'eggs'}
```

You can make these as complex as you need. As long as all the parts are hashable, it will function as expected.

Perhaps, even more useful is the fact that tuples also support tuple packing and unpacking:

```
# Assign using tuples on both sides
>>> a, b, c = 1, 2, 3
>>> a
1


# Assign a tuple to a single variable
>>> spam = a, (b, c)
>>> spam
(1, (2, 3))


# Unpack a tuple to two variables
>>> a, b = spam
>>> a
1
>>> b
(2, 3)
```

In addition to regular packing and unpacking, from Python 3 onwards, we can actually pack and unpack objects with a variable number of items:

```
# Unpack with variable length objects which actually assigns as a
list, not a tuple
>>> spam, *eggs = 1, 2, 3, 4
>>> spam
1
>>> eggs
[2, 3, 4]

# Which can be unpacked as well of course
>>> a, b, c = eggs
>>> c
4

# This works for ranges as well
>>> spam, *eggs = range(10)
>>> spam
0
>>> eggs
[1, 2, 3, 4, 5, 6, 7, 8, 9]

# Which works both ways
>>> a
2
>>> a, b, *c = a, *eggs
>>> a, b
(2, 1)
>>> c
[2, 3, 4, 5, 6, 7, 8, 9]
```

This very method can be applied in many cases, even for function arguments:

```
>>> def eggs(*args):
...     print('args:', args)

>>> eggs(1, 2, 3)
args: (1, 2, 3)
```

And its equally useful to return multiple arguments from a function:

```
>>> def spam_eggs():
...     return 'spam', 'eggs'

>>> spam, eggs = spam_eggs()
>>> print('spam: %s, eggs: %s' % (spam, eggs))
spam: spam, eggs: eggs
```

Advanced collections

The following collections are mostly just extensions of base collections, some of them fairly simple and others a bit more advanced. For all of them though, it is important to know the characteristics of the underlying structures. Without understanding them, it will be difficult to comprehend the characteristics of these collections.

There are a few collections that are implemented in native C code for performance reasons, but all of them can easily be implemented in pure Python as well.

ChainMap – the list of dictionaries

Introduced in Python 3.3, `ChainMap` allows you to combine multiple mappings (dictionaries for example) into one. This is especially useful when combining multiple contexts. For example, when looking for a variable in your current scope, by default, Python will search in `locals()`, `globals()`, and lastly `builtins`.

Normally, you would do something like this:

```
import builtins

builtin_vars = vars(builtins)
if key in locals():
    value = locals()[key]
elif key in globals():
    value = globals()[key]
elif key in builtin_vars:
    value = builtin_vars[key]
else:
    raise NameError('name %r is not defined' % key)
```

This works, but it's ugly to say the least. We can make it prettier, of course:

```
import builtins

mappings = globals(), locals(), vars(builtins)
for mapping in mappings:
    if key in mapping:
        value = mapping[key]
        break
else:
    raise NameError('name %r is not defined' % key)
```

A lot better! Moreover, this can actually be considered a nice solution. But since Python 3.3, it's even easier. Now we can simply use the following code:

```
import builtins
import collections

mappings = collections.ChainMap(globals(), locals(), vars(builtins))
value = mappings[key]
```

The `ChainMap` collection is very useful for command-line applications. The most important configuration happens through command-line arguments, followed by directory local config files, followed by global config files, followed by defaults:

```
import argparse
import collections

defaults = {
    'spam': 'default spam value',
    'eggs': 'default eggs value',
}

parser = argparse.ArgumentParser()
parser.add_argument('--spam')
parser.add_argument('--eggs')

args = vars(parser.parse_args())
# We need to check for empty/default values so we can't simply use
vars(args)
filtered_args = {k: v for k, v in args.items() if v}

combined = collections.ChainMap(filtered_args, defaults)

print(combined ['spam'])
```

Note that accessing specific mappings is still possible:

```
print(combined.maps[1]['spam'])

for map_ in combined.maps:
    print(map_.get('spam'))
```

counter – keeping track of the most occurring elements

The `counter` is a class for keeping track of the number of occurrences of an element. Its basic usage is as you would expect:

```
>>> import collections


>>> counter = collections.Counter('eggs')
>>> for k in 'eggs':
...     print('Count for %s: %d' % (k, counter[k]))
Count for e: 1
Count for g: 2
Count for g: 2
Count for s: 1
```

However, `counter` can do more than simply return the count. It also has a few very useful and fast (it uses `heapq`) methods for getting the most common elements. Even with a million elements added to the counter, it still executes within a second:

```
>>> import math
>>> import collections

>>> counter = collections.Counter()
>>> for i in range(0, 100000):
...    counter[math.sqrt(i) // 25] += 1

>>> for key, count in counter.most_common(5):
...     print('%s: %d' % (key, count))
11.0: 14375
10.0: 13125
9.0: 11875
8.0: 10625
12.0: 10000
```

But wait, there's more! In addition to getting the most frequent elements, it's also possible to add, subtract, intersect, and "union" counters very similarly to the `set` operations that we saw earlier. So what is the difference between adding two counters and making a union of them? As you would expect, they are similar, but there is a small difference. Let's look at its workings:

```
>>> import collections


>>> def print_counter(expression, counter):
...     sorted_characters = sorted(counter.elements())
...     print(expression, ''.join(sorted_characters))

>>> eggs = collections.Counter('eggs')
>>> spam = collections.Counter('spam')
>>> print_counter('eggs:', eggs)
eggs: eggs
>>> print_counter('spam:', spam)
spam: amps
>>> print_counter('eggs & spam:', eggs & spam)
eggs & spam: s
>>> print_counter('spam & eggs:', spam & eggs)
spam & eggs: s
>>> print_counter('eggs - spam:', eggs - spam)
eggs - spam: egg
>>> print_counter('spam - eggs:', spam - eggs)
spam - eggs: amp
>>> print_counter('eggs + spam:', eggs + spam)
eggs + spam: aeggmpss
>>> print_counter('spam + eggs:', spam + eggs)
spam + eggs: aeggmpss
>>> print_counter('eggs | spam:', eggs | spam)
eggs | spam: aeggmps
>>> print_counter('spam | eggs:', spam | eggs)
spam | eggs: aeggmps
```

The first two are obvious. The `eggs` string is just a sequence of characters with two "`g`"s, one "`s`", and one "`e`", and spam is almost the same but with different letters.

The result of `spam & eggs` (and the reverse) is also quite predictable. The only letter that's shared between spam and eggs is `s`, so that's the result. When it comes to counts, it simply does a `min(element_a, element_b)` per shared element from both and gets the lowest.

When subtracting the letters `s`, `p`, `a`, and `m` from eggs, you are left with `e` and `g`. Similarly, when removing `e`, `g`, and `s` from spam, you are left with `p`, `a`, and `m`.

Now, adding is as you would expect—just an element-by-element addition of both counters.

So how is the union (OR) any different? It gets the `max(element_a, element_b)` per element in either of the counters instead of adding them; regardless as is the case with the addition.

Lastly, as is demonstrated in the preceding code, the elements method returns an expanded list of all elements repeated by the count.

The `Counter` object will automatically remove elements that are zero or less during the execution of mathematical operations.

deque – the double ended queue

The `deque` (short for Double Ended Queue) object is one of the oldest collections. It was introduced in Python 2.4, so it has been available for over 10 years by now. Generally, this object will be too low-level for most purposes these days, as many operations that would otherwise use it have well-supported libraries available, but that doesn't make it less useful.

Internally, `deque` is created as a doubly linked list, which means that every item points to the next and the previous item. Since `deque` is double-ended, the list itself points to both the first and the last element. This makes both adding and removing items from either the beginning or the end a very light `O(1)` operation, since only the pointer to the beginning/end of the list needs to change and a pointer needs to be added to the first/last item, depending on whether an item is added at the beginning or the end.

For simple stack/queue purposes, it seems wasteful to use a double-ended queue, but the performance is good enough for us not to care about the overhead incurred. The `deque` class is fully implemented in C (with CPython).

Its usage as a queue is very straightforward:

```
>>> import collections

>>> queue = collections.deque()
>>> queue.append(1)
>>> queue.append(2)
>>> queue
deque([1, 2])
>>> queue.popleft()
1
>>> queue.popleft()
2
>>> queue.popleft()
Traceback (most recent call last):
    ...
IndexError: pop from an empty deque
```

As expected, the items are followed by an `IndexError` since there are only two items and we are trying to get three.

The usage as a stack is almost identical, but we have to use `pop` instead of `popleft` (or `appendleft` instead of `append`):

```
>>> import collections

>>> queue = collections.deque()
>>> queue.append(1)
>>> queue.append(2)
>>> queue
deque([1, 2])
>>> queue.pop()
2
>>> queue.pop()
1
>>> queue.pop()
Traceback (most recent call last):
    ...
IndexError: pop from an empty deque
```

Another very useful feature is that `deque` can be used as a circular queue with the `maxlen` parameter. By using this, it can be used to keep the last `n` status messages or something similar:

```
>>> import collections


>>> circular = collections.deque(maxlen=2)
>>> for i in range(5):
...     circular.append(i)
...     circular
deque([0], maxlen=2)
deque([0, 1], maxlen=2)
deque([1, 2], maxlen=2)
deque([2, 3], maxlen=2)
deque([3, 4], maxlen=2)
>>> circular
deque([3, 4], maxlen=2)
```

Whenever you require a queue or stack class within a single-threaded application, `deque` is a very convenient option. If you require the object to be synchronized for multithreading operations, then the `queue.Queue` class would be better suited. Internally, it wraps `deque`, but it's a thread-safe alternative. In the same category, there is also an `asyncio.Queue` for asynchronous operations and `multiprocessing.Queue` for multiprocessing operations. Examples of `asyncio` and multiprocessing can be found in *Chapter 7*, *Async IO – Multithreading without Threads* and *Chapter 13*, *Multiprocessing – When a Single CPU Core Is Not Enough* respectively.

defaultdict – dictionary with a default value

The `defaultdict` is by far my favorite object in the collections package. I still remember writing my own versions of it before it was added to the core. While it's a fairly simple object, it is extremely useful for all sorts of design patterns. Instead of having to check for the existence of a key and adding a value every time, you can just declare the default from the beginning, and there is no need to worry about the rest.

For example, let's say we are building a very basic graph structure from a list of connected nodes.

This is our list of connected nodes (one way):

```
nodes = [
    ('a', 'b'),
```

```
        ('a', 'c'),
        ('b', 'a'),
        ('b', 'd'),
        ('c', 'a'),
        ('d', 'a'),
        ('d', 'b'),
        ('d', 'c'),
    ]
```

Now let's put this graph into a normal dictionary:

```
>>> graph = dict()
>>> for from_, to in nodes:
...     if from_ not in graph:
...         graph[from_] = []
...     graph[from_].append(to)

>>> import pprint
>>> pprint.pprint(graph)
{'a': ['b', 'c'],
 'b': ['a', 'd'],
 'c': ['a'],
 'd': ['a', 'b', 'c']}
```

Some variations are possible, of course, using `setdefault` for example. But they remain more complex than they need to be.

The truly Pythonic version uses `defaultdict` instead:

```
>>> import collections

>>> graph = collections.defaultdict(list)
>>> for from_, to in nodes:
...     graph[from_].append(to)

>>> import pprint
>>> pprint.pprint(graph)
defaultdict(<class 'list'>,
            {'a': ['b', 'c'],
             'b': ['a', 'd'],
             'c': ['a'],
             'd': ['a', 'b', 'c']})
```

Isn't that a beautiful bit of code? The `defaultdict` can actually be seen as the precursor of the `counter` object. It's not as fancy and doesn't have all the bells and whistles that `counter` has, but it does the job in many cases:

```
>>> counter = collections.defaultdict(int)
>>> counter['spam'] += 5
>>> counter
defaultdict(<class 'int'>, {'spam': 5})
```

The default value for `defaultdict` needs to be a callable object. In the previous cases, these were `int` and `list`, but you can easily define your own functions to use as a default value. That's what the following example uses, although I won't recommend production usage since it lacks a bit of readability. I do believe, however, that it is a beautiful example of the power of Python.

This is how we create a `tree` in a single line of Python:

```
import collections
def tree(): return collections.defaultdict(tree)
```

Brilliant, isn't it? Here's how we can actually use it:

```
>>> import json
>>> import collections


>>> def tree():
...     return collections.defaultdict(tree)

>>> colours = tree()
>>> colours['other']['black'] = 0x000000
>>> colours['other']['white'] = 0xFFFFFF
>>> colours['primary']['red'] = 0xFF0000
>>> colours['primary']['green'] = 0x00FF00
>>> colours['primary']['blue'] = 0x0000FF
>>> colours['secondary']['yellow'] = 0xFFFF00
>>> colours['secondary']['aqua'] = 0x00FFFF
>>> colours['secondary']['fuchsia'] = 0xFF00FF

>>> print(json.dumps(colours, sort_keys=True, indent=4))
{
```

```
    "other": {
        "black": 0,
        "white": 16777215
    },
    "primary": {
        "blue": 255,
        "green": 65280,
        "red": 16711680
    },
    "secondary": {
        "aqua": 65535,
        "fuchsia": 16711935,
        "yellow": 16776960
    }
}
```

The nice thing is that you can make it go as deep as you'd like. Because of the `defaultdict` base, it generates itself recursively.

namedtuple – tuples with field names

The `namedtuple` object is exactly what the name implies—a tuple with a name. It has a few useful use cases, though I must admit that I haven't found too many in the wild, except for some Python modules such as inspect and `urllib.parse`. Points in 2D or 3D space are a nice example of where it is definitely useful:

```
>>> import collections

>>> Point = collections.namedtuple('Point', ['x', 'y', 'z'])
>>> point_a = Point(1, 2, 3)
>>> point_a
Point(x=1, y=2, z=3)

>>> point_b = Point(x=4, z=5, y=6)
>>> point_b
Point(x=4, y=6, z=5)
```

Not too much can be said about `namedtuple`; it does what you would expect, and the greatest advantage is that the properties can be executed both by name and by index, which makes tuple unpacking quite easy:

```
>>> x, y, z = point_a
>>> print('X: %d, Y: %d, Z: %d' % (x, y, z))
X: 1, Y: 2, Z: 3
>>> print('X: %d, Y: %d, Z: %d' % point_b)
X: 4, Y: 6, Z: 5
>>> print('X: %d' % point_a.x)
```

enum – a group of constants

The `enum` package is quite similar to `namedtuple` but has a completely different goal and interface. The basic `enum` object makes it really easy to have constants in your modules while still avoiding magic numbers. This is a basic example:

```
>>> import enum


>>> class Color(enum.Enum):
...     red = 1
...     green = 2
...     blue = 3

>>> Color.red
<Color.red: 1>
>>> Color['red']
<Color.red: 1>
>>> Color(1)
<Color.red: 1>
>>> Color.red.name
'red'
>>> Color.red.value
1
>>> isinstance(Color.red, Color)
True
>>> Color.red is Color['red']
```

```
True
>>> Color.red is Color(1)
True
```

A few of the handy features of the `enum` package are that the objects are iterable, accessible through both numeric and textual representation of the values, and, with proper inheritance, even comparable to other classes.

The following code shows the usage of a basic API:

```
>>> for color in Color:
...     color
<Color.red: 1>
<Color.green: 2>
<Color.blue: 3>


>>> colors = dict()
>>> colors[Color.green] = 0x00FF00
>>> colors
{<Color.green: 2>: 65280}
```

There is more though. One of the lesser known possibilities from the `enum` package is that you can make value comparisons work through inheritance of specific types, and this works for every type—not just integers but (your own) custom types as well.

This is the regular `enum`:

```
>>> import enum



>>> class Spam(enum.Enum):
...     EGGS = 'eggs'

>>> Spam.EGGS == 'eggs'
False
```

The following is `enum` with `str` inheritance:

```
>>> import enum



>>> class Spam(str, enum.Enum):
```

```
...     EGGS = 'eggs'

>>> Spam.EGGS == 'eggs'
True
```

OrderedDict – a dictionary where the insertion order matters

`OrderdDict` is a `dict` that keeps track of the order in which the items were inserted. Whereas a normal `dict` will return your keys in the order of hash, `OrderedDict` will return your keys by the order of insertion. So, it's not ordered by key or value, but that is easily possible too:

```
>>> import collections


>>> spam = collections.OrderedDict()
>>> spam['b'] = 2
>>> spam['c'] = 3
>>> spam['a'] = 1
>>> spam
OrderedDict([('b', 2), ('c', 3), ('a', 1)])

>>> for key, value in spam.items():
...     key, value
('b', 2)
('c', 3)
('a', 1)

>>> eggs = collections.OrderedDict(sorted(spam.items()))
>>> eggs
OrderedDict([('a', 1), ('b', 2), ('c', 3)])
```

While you can probably guess how this works, the internals might surprise you a little. I know I was expecting a different implementation.

Internally, `OrderedDict` uses a normal `dict` for key/value storage, and in addition to that, it uses a doubly linked list to keep track of the next/previous items. To keep track of the reverse relation (from the doubly linked list back to the keys), there is an extra `dict` stored internally.

Put simply, `OrderedDict` can be a very handy tool for keeping your `dict` sorted, but it does come at a cost. The system is structured in such a way that `set` and `get` are really fast `O(1)`, but the object is still a lot heavier (double or more memory usage) when compared to a regular `dict`. In many cases, the memory usage of the objects inside will outweigh the memory usage of the `dict` itself, of course, but this is something to keep in mind.

heapq – the ordered list

The `heapq` module is a great little module that makes it very easy to create a priority queue in Python. A structure that will always make the smallest (or largest, depending on the implementation) item available with minimum effort. The API is quite simple, and one of the best examples of its usage can be seen in the `OrderedDict` object. You probably don't want to use `heapq` directly, but understanding the inner workings is important in order to analyze how classes such as `OrderedDict` work.

If you are looking for a structure to keep your list always sorted, try the `bisect` module instead.

The basic usage is quite simple though:

```
>>> import heapq


>>> heap = [1, 3, 5, 7, 2, 4, 3]
>>> heapq.heapify(heap)
>>> heap
[1, 2, 3, 7, 3, 4, 5]

>>> while heap:
...     heapq.heappop(heap), heap
(1, [2, 3, 3, 7, 5, 4])
(2, [3, 3, 4, 7, 5])
(3, [3, 5, 4, 7])
(3, [4, 5, 7])
(4, [5, 7])
(5, [7])
(7, [])
```

One important thing to note here—something that you have probably already understood from the preceding example—is that the `heapq` module does not create a special object. It is simply a bunch of methods for treating a regular list as a `heap`. That doesn't make it less useful, but it is something to take into consideration. You may also wonder why the `heap` isn't sorted. Actually, it is sorted but not the way you expect it to be. If you view the `heap` as a tree, it becomes much more obvious:

```
    1
  2   3
 7 3 4 5
```

The smallest number is always at the top and the biggest numbers are always at the bottom of the tree. Because of that, it's really easy to find the smallest number, but finding the largest is not so easy. To get the sorted version of the heap, we simply need to keep removing the top of the tree until all items are gone.

bisect – the sorted list

We have seen the `heapq` module in the previous paragraph, which makes it really simple to always get the smallest number from a list, and therefore makes it easy to sort a list of objects. While the `heapq` module appends items to form a tree-like structure, the `bisect` module inserts items in such a way that they stay sorted. A big difference is that adding/removing items with the `heapq` module is very light whereas finding items is really light with the `bisect` module. If your primary purpose is searching, then `bisect` should be your choice.

As is the case with `heapq`, `bisect` does not really create a special data structure. It just works on a standard `list` and expects that `list` to always be sorted. It is important to understand the performance implications of this; simply adding items to the list using the `bisect` algorithm can be very slow because an insert on a list takes `O(n)`. Effectively, creating a sorted list using bisect takes `O(n*n)`, which is quite slow, especially because creating the same sorted list using `heapq` or sorted takes `O(n * log(n))` instead.

The `log(n)` refers to the base 2 logarithm function. To calculate this value, the `math.log2()` function can be used. This results in an increase of 1 every time the number doubles in size. For `n=2`, the value of `log(n)` is `1`, and consequently for `n=4` and `n=8`, the log values are `2` and `3`, respectively.

This means that a 32-bit number, which is `2**32 = 4294967296`, has a log of `32`.

If you have a sorted structure and you only need to add a single item, then the `bisect` algorithm can be used for insertion. Otherwise, it's generally faster to simply append the items and call a `.sort()` afterwards.

To illustrate, we have these lines:

```
>>> import bisect

Using the regular sort:
>>> sorted_list = []
>>> sorted_list.append(5)  # O(1)
>>> sorted_list.append(3)  # O(1)
>>> sorted_list.append(1)  # O(1)
>>> sorted_list.append(2)  # O(1)
>>> sorted_list.sort()  # O(n * log(n)) = O(4 * log(4)) = O(8)
>>> sorted_list
[1, 2, 3, 5]

Using bisect:
>>> sorted_list = []
>>> bisect.insort(sorted_list, 5)  # O(n) = O(1)
>>> bisect.insort(sorted_list, 3)  # O(n) = O(2)
>>> bisect.insort(sorted_list, 1)  # O(n) = O(3)
>>> bisect.insort(sorted_list, 2)  # O(n) = O(4)
>>> sorted_list
[1, 2, 3, 5]
```

For a small number of items, the difference is negligible, but it quickly grows to a point where the difference will be large. For `n=4`, the difference is just between `4 * 1 + 8 = 12` and `1 + 2 + 3 + 4 = 10` making the bisect solution faster. But if we were to insert 1,000 items, it would be `1000 + 1000 * log(1000) = 10966` versus `1 + 2 + … 1000 = 1000 * (1000 + 1) / 2 = 500500`. So, be very careful while inserting many items.

Searching within the list is very fast though; because it is sorted, we can use a very simple binary search algorithm. For example, what if we want to check whether a few numbers exist within the list?

```
>>> import bisect



>>> sorted_list = [1, 2, 3, 5]
```

```
>>> def contains(sorted_list, value):
...     i = bisect.bisect_left(sorted_list, value)
...     return i < len(sorted_list) and sorted_list[i] == value

>>> contains(sorted_list, 2)
True
>>> contains(sorted_list, 4)
False
>>> contains(sorted_list, 6)
False
```

As you can see, the `bisect_left` function finds the position at which the number is supposed to be. This is actually what the `insort` function does as well; it inserts the number at the correct position by searching for the location of the number.

So how is this different from a regular value in `sorted_list`? The biggest difference is that `bisect` does a binary search internally, which means that it starts in the middle and jumps left or right depending on whether the value is bigger or smaller than the value. To illustrate, we will search for `4` in a list of numbers from `0` to `14`:

```
sorted_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
Step 1: 4 > 7                       ^
Step 2: 4 > 3           ^
Step 3: 4 > 5                 ^
Step 4: 4 > 5              ^
```

As you can see, after only four steps (actually three; the fourth is just for illustration), we have found the number we searched for. Depending on the number (`7`, for example), it may go faster, but it will never take more than `O(log(n))` steps to find a number.

With a regular list, a search would simply walk through all items until it finds the desired item. If you're lucky, it could be the first number you encounter, but if you're unlucky, it could be the last item. In the case of 1,000 items, that would be the difference between 1,000 steps and `log(1000) = 10` steps.

Summary

Python has quite a few very useful collections built in. Since more and more collections are added regularly, the best thing to do is simply keep track of the collections manual. And do you ever wonder how or why any of the structures works? Just look at the source here:

`https://hg.python.org/cpython/file/default/Lib/collections/__init__.py`

After finishing this chapter, you should be aware of both the core collections and the most important collections from the collections module, but more importantly the performance characteristics of these collections in several scenarios. Selecting the correct data structure within your applications is by far the most important performance factor that your code will ever experience, making this essential knowledge for any programmer.

Next, we will continue with functional programming which covers `lambda` functions, `list` comprehensions, `dict` comprehensions, `set` comprehensions and an array of related topics. This includes some background information on the mathematics involved which could be interesting but can safely be skipped.

4
Functional Programming – Readability Versus Brevity

Python is one of the few (or at least the earliest) nonfunctional languages to incorporate functional features. While Guido van Rossum has tried to remove some of them a few times, they have become ingrained in the Python community, and list comprehensions (`dict` and `set` comprehensions soon to follow) are widely used in all sorts of code. The most important thing about code shouldn't be how cool your `reduce` statement is or how you can fit the entire function in a single line with an incomprehensible list comprehension. Readability counts (once again, `PEP20`)!

This chapter will show you some of the cool tricks that functional programming in Python gives you, and it will explain some of the limitations of Python's implementation. While we will try to steer clear of lambda calculus (λ-calculus) as much as possible, the **Y combinator** will be discussed briefly.

The last few paragraphs will list (and explain) the usage of the `functools` and `itertools` libraries. If you are familiar with these libraries, feel free to skip them, but note that some of these will be used heavily in the later chapters about decorators (*Chapter 5*, *Decorators – Enabling Code Reuse by Decorating*), generators (*Chapter 6*, *Generators and Coroutines – Infinity, One Step at a Time*), and performance (*Chapter 12*, *Performance – Tracking and Reducing Your Memory and CPU Usage*).

These are the topics covered in this chapter:

- The theory behind functional programming
- `list` comprehensions
- `dict` comprehensions
- `set` comprehensions
- `lambda` functions
- `functools` (`partial`, and `reduce`)
- `itertools` (`accumulate`, `chain`, `dropwhile`, `starmap`, and so on)

Functional programming

Functional programming is a paradigm that originates from the lambda calculus. Without diving too much into the lambda calculus (λ-calculus), this roughly means that computation is performed through the use of mathematical functions, which avoids mutable data and changing state of surroundings. The idea of a strictly functional language is that all function outputs are dependent only on the input and not on any external state. Since Python is not strictly a programming language, this doesn't necessarily hold true, but it is a good idea to adhere to this paradigm as mixing these can cause unforeseen bugs as discussed in *Chapter 2*, *Pythonic Syntax, Common Pitfalls, and Style Guide*.

Even outside of functional programming, this is a good idea. Keeping functions purely functional (relying only on the given input) makes code clearer, easier to understand, and better to test as there are less dependencies. Well-known examples can be found within the `math` module. These functions (`sin`, `cos`, `pow`, `sqrt`, and so on) have an input and an output that is strictly dependent on the input.

list comprehensions

The Python `list` comprehensions are a very easy way to apply a function or filter to a list of items. List comprehensions can be very useful if used correctly but very unreadable if you're not careful.

Let's dive right into a few examples. The basic premise of a `list` comprehension looks like this:

```
>>> squares = [x ** 2 for x in range(10)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

We can easily expand this with a filter:

```
>>> uneven_squares = [x ** 2 for x in range(10) if x % 2]
>>> uneven_squares
[1, 9, 25, 49, 81]
```

The syntax is pretty close to regular Python for loops, but the `if` statement and automatic storing of results makes it quite useful for some cases. The regular Python equivalent is not much longer, however:

```
>>> uneven_squares = []
>>> for x in range(10):
...     if x % 2:
...         uneven_squares.append(x ** 2)

>>> uneven_squares
[1, 9, 25, 49, 81]
```

Care must be taken though; because of the special list comprehension structure, some types of operations are not as obvious as you might expect. This time, we are looking for random numbers greater than `0.5`:

```
>>> import random
>>> [random.random() for _ in range(10) if random.random() >= 0.5]
[0.5211948104577864, 0.650010512129705, 0.021427316545174158]
```

See that last number? It's actually less than `0.5`. This happens because the first and the last random calls are actually separate calls and return different results.

One way to counter this is by creating the list separate from the filter:

```
>>> import random
>>> numbers = [random.random() for _ in range(10)]
>>> [x for x in numbers if x >= 0.5]
[0.715510247827078, 0.8426277505519564, 0.5071133900377911]
```

That obviously works, but it's not all that pretty. So what other options are there? Well, there are a few but the readability is a bit questionable, so these are not the solutions that I would recommend. It's good to see them at least once, however.

Here is a `list` comprehension in a list comprehension:

```
>>> import random
>>> [x for x in [random.random() for _ in range(10)] if x >= 0.5]
```

And here's one that quickly becomes an incomprehensible `list` comprehension:

```
>>> import random
>>> [x for _ in range(10) for x in [random.random()] if x >= 0.5]
```

Caution is needed with these options as the double list comprehension actually works like a nested `for` loop would, so it quickly generates a lot of results. To elaborate on this regard:

```
>>> [(x, y) for x in range(3) for y in range(3, 5)]
[(0, 3), (0, 4), (1, 3), (1, 4), (2, 3), (2, 4)]
```

This effectively does the following:

```
>>> results = []
>>> for x in range(3):
...     for y in range(3, 5):
...         results.append((x, y))
...
>>> results
[(0, 3), (0, 4), (1, 3), (1, 4), (2, 3), (2, 4)]
```

These can be useful for some cases, but I would recommend that you limit their usage, as they have a tendency to quickly become unreadable. I would strongly advise against using `list` comprehensions within `list` comprehensions for the sake of readability. It's still important to understand what is happening, so let's look at one more example. The following `list` comprehension swaps the column and row counts, so a 3 x 4 matrix becomes 4 x 3:

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]

>>> reshaped_matrix = [
...     [
...         [y for x in matrix for y in x][i * len(matrix) + j]
...         for j in range(len(matrix))
...     ]
...     for i in range(len(matrix[0]))
```

```
... ]


>>> import pprint
>>> pprint.pprint(reshaped_matrix, width=40)
[[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9],
 [10, 11, 12]]
```

Even with the extra indentation, the `list` comprehension just isn't all that readable. With four nested loops, that is expectedly so, of course. There are rare cases where nested list comprehensions might be justified, but generally I won't recommend their usage.

dict comprehensions

`dict` comprehensions are very similar to list comprehensions, but the result is a `dict` instead. Other than this, the only real difference is that you need to return both a key and a value, whereas a `list` comprehension accepts any type of value. The following is a basic example:

```
>>> {x: x ** 2 for x in range(10)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}


>>> {x: x ** 2 for x in range(10) if x % 2}
{1: 1, 3: 9, 9: 81, 5: 25, 7: 49}
```

Since the output is a dictionary, the key needs to be hashable for the `dict` comprehension to work.

The funny thing is that you can mix these two, of course, for even more unreadable magic:

```
>>> {x ** 2: [y for y in range(x)] for x in range(5)}
{0: [], 1: [0], 4: [0, 1], 16: [0, 1, 2, 3], 9: [0, 1, 2]}
```

Obviously, you need to be careful with these. They can be very useful if used correctly, but the output quickly becomes unreadable, even with proper whitespace.

set comprehensions

Just as you can create a `set` using curly brackets ({}), you can also create a set using `set` comprehensions. These work in a way similar to `list` comprehensions, but the values are unique (and without sort order):

```
>>> [x*y for x in range(3) for y in range(3)]
[0, 0, 0, 0, 1, 2, 0, 2, 4]

>>> {x*y for x in range(3) for y in range(3)}
{0, 1, 2, 4}
```

As is the case with the regular set, `set` comprehensions support only hashable types.

lambda functions

The `lambda` statement in Python is simply an anonymous function. Due to the syntax, it is slightly more limited than regular functions, but a lot can be done through it. As always though, readability counts, so generally it is a good idea to keep it as simple as possible. One of the more common use cases is the `sort` key for the `sorted` function:

```
>>> class Spam(object):
...     def __init__(self, value):
...         self.value = value
...
...     def __repr__(self):
...         return '<%s: %s>' % (self.__class__.__name__, self.value)
...
>>> spams = [Spam(5), Spam(2), Spam(4), Spam(1)]
>>> sorted_spams = sorted(spams, key=lambda spam: spam.value)
>>> spams
[<Spam: 5>, <Spam: 2>, <Spam: 4>, <Spam: 1>]
>>> sorted_spams
[<Spam: 1>, <Spam: 2>, <Spam: 4>, <Spam: 5>]
```

While the function could have been written separately or the `__cmp__` method of `Spam` could have been overwritten in this case, in many cases, this is an easy way to get a quick sort function as you would want it.

It's not that the regular function would be verbose, but by using an anonymous function, you have a small advantage; you are not contaminating your local scope with an extra function:

```
>>> def key_function(spam):
...     return spam.value


>>> spams = [Spam(5), Spam(2), Spam(4), Spam(1)]
>>> sorted_spams = sorted(spams, key=lambda spam: spam.value)
```

As for style, do note that `PEP8` dictates that assigning a lambda to a variable is a bad idea. And logically, it is. The idea of an anonymous function is that it is just that—anonymous. If you are giving it an identity, you should define it as a normal function. It really isn't much longer if you want to keep it short. Note that both of the following statements are considered bad style and are for example purposes only:

```
>>> def key(spam): return spam.value


>>> key = lambda spam: spam.value
```

In my opinion, the only valid use case for `lambda` functions is as anonymous functions used as function parameters, and preferably only if they are short enough to fit on a single line.

The Y combinator

Note that this paragraph can easily be skipped. It is mostly an example of the mathematical value of the lambda statement.

The Y combinator is probably the most famous example of the λ-calculus:

$$Y = \lambda f.\left(\lambda x.\, f(x\,x)\right)\left(\lambda x.\, f(x\,x)\right)$$

All this looks very complicated, but that's also because it has used the lambda calculus notation. You should read this syntax, $\lambda x.\, x^2$, as an anonymous (lambda) function that takes `x` as an input and returns x^2. In Python, this would be expressed almost exactly as it is in the original lambda calculus, except for replacing λ with lambda and `.` with `:`, so it results in lambda `x: x^2`.

With some algebra, this can be reduced to $Y\ f = f(Y\ f)$, or a function that takes the `f` function and applies it to itself. The λ-calculus notation of this function is as follows:

$$\lambda x.\ f(xx)$$

Here is the Python notation:

```
Y = lambda f: lambda *args: f(Y(f))(*args)
```

The following is the longer version:

```
def Y(f):
    def y(*args):
        y_function = f(Y(f))
        return y_function(*args)
    return y
```

This might still be a bit unclear to you, so let's look at an example that actually uses it:

```
>>> Y = lambda f: lambda *args: f(Y(f))(*args)

>>> def factorial(combinator):
...     def _factorial(n):
...         if n:
...             return n * combinator(n - 1)
...         else:
...             return 1
...     return _factorial
>>> Y(factorial)(5)
120
```

The following is the short version, where the power of the Y combinator actually appears, with a recursive but still anonymous function:

```
>>> Y = lambda f: lambda *args: f(Y(f))(*args)

>>> Y(lambda c: lambda n: n and n * c(n - 1) or 1)(5)
120
```

Note that the `n` and `n * c(n - 1)` or `1` part is short for the `if` statement used in the longer version of the function. Alternatively, this can be written using the Python ternary operator:

```
>>> Y = lambda f: lambda *args: f(Y(f))(*args)


>>> Y(lambda c: lambda n: n * c(n - 1) if n else 1)(5)
120
```

You might be wondering about the point of this entire exercise. Can't you write a factorial shorter/easier? Yes, you can. The importance of the Y combinator is that it can be applied to any function and is very close to the mathematical definition.

One final example of the Y combinator will be given by the definition of `quicksort` in a few lines:

```
>>> quicksort = Y(lambda f:
...     lambda x: (
...         f([item for item in x if item < x[0]])
...         + [y for y in x if x[0] == y]
...         + f([item for item in x if item > x[0]])
...     ) if x else [])


>>> quicksort([1, 3, 5, 4, 1, 3, 2])
[1, 1, 2, 3, 3, 4, 5]
```

While the Y combinator most likely doesn't have much practical use in Python, it does show the power of the `lambda` statement and how close Python is to the mathematical definition. Essentially, the difference is only in the notation and not in the functionality.

functools

In addition to the `list/dict/set` comprehensions, Python also has a few (more advanced) functions that can be really convenient when coding functionally. The `functools` library is a collection of functions that return callable objects. Some of these functions are used as decorators (we'll cover more about that in *Chapter 5*, *Decorators – Enabling Code Reuse by Decorating*), but the ones that we are going to talk about are used as straight-up functions to make your life easier.

partial – no need to repeat all arguments every time

The `partial` function is really convenient for adding some default arguments to a function that you use often but can't (or don't want to) redefine. With object-oriented code, you can usually work around cases similar to these, but with procedural code, you will often have to repeat your arguments. Let's take the `heapq` functions from *Chapter 3*, *Containers and Collections – Storing Data the Right Way*, as an example:

```
>>> import heapq
>>> heap = []
>>> heapq.heappush(heap, 1)
>>> heapq.heappush(heap, 3)
>>> heapq.heappush(heap, 5)
>>> heapq.heappush(heap, 2)
>>> heapq.heappush(heap, 4)
>>> heapq.nsmallest(3, heap)
[1, 2, 3]
```

Almost all the `heapq` functions require a `heap` argument, so why not make a shortcut for it? This is where `functools.partial` comes in:

```
>>> import functools
>>> import heapq
>>> heap = []
>>> push = functools.partial(heapq.heappush, heap)
>>> smallest = functools.partial(heapq.nsmallest, iterable=heap)

>>> push(1)
>>> push(3)
>>> push(5)
>>> push(2)
>>> push(4)
>>> smallest(3)
[1, 2, 3]
```

Seems a bit cleaner, right? In this case, both versions are fairly short and readable, but it's a convenient function to have.

Why should we use `partial` instead of writing a `lambda` argument? Well, it's mostly about convenience, but it also helps solve the late binding problem discussed in *Chapter 2, Pythonic Syntax, Common Pitfalls, and Style Guide*. Additionally, partial functions can be pickled whereas `lambda` statements cannot.

reduce – combining pairs into a single result

The `reduce` function implements a mathematical technique called `fold`. It basically applies a function to the first and second elements, uses that result to apply together with the third element, and continues until the list is exhausted.

The `reduce` function is supported by many languages but in most cases using different names such as `curry`, `fold`, `accumulate`, or `aggregate`. Python has actually supported `reduce` for a very long time, but since Python 3, it has been moved from the global scope to the `functools` library. Some code can be simplified beautifully using the `reduce` statement; whether it's readable or not is debatable, however.

Implementing a factorial function

One of the most used examples of `reduce` is for calculating factorials, which is indeed quite simple:

```
>>> import operator
>>> import functools
>>> functools.reduce(operator.mul, range(1, 6))
120
```

The preceding code uses `operator.mul` instead of `lambda a, b: a * b`. While they produce the same results, the former can be quite faster.

Internally, the `reduce` function will do the following:

```
>>> import operator
>>> f = operator.mul
>>> f(f(f(f(1, 2), 3), 4), 5)
120
```

To clarify this further, let's look at it like this:

```
>>> iterable = range(1, 6)
>>> import operator
```

```
# The initial values:
>>> a, b, *iterable = iterable
>>> a, b, iterable
(1, 2, [3, 4, 5])

# First run
>>> a = operator.mul(a, b)
>>> b, *iterable = iterable
>>> a, b, iterable
(2, 3, [4, 5])

# Second run
>>> a = operator.mul(a, b)
>>> b, *iterable = iterable
>>> a, b, iterable
(6, 4, [5])

# Third run
>>> a = operator.mul(a, b)
>>> b, *iterable = iterable
>>> a, b, iterable
(24, 5, [])

# Fourth and last run
>>> a = operator.mul (a, b)
>>> a
120
```

Or with a simple `while` loop using the `deque` collection:

```
>>> import operator
>>> import collections
>>> iterable = collections.deque(range(1, 6))

>>> value = iterable.popleft()
>>> while iterable:
...     value = operator.mul(value, iterable.popleft())

>>> value
120
```

Processing trees

Trees are a case where the `reduce` function really shines. Remember the one-line tree definition using a `defaultdict` from *Chapter 3, Containers and Collections – Storing Data the Right Way*? What would be a good way to access the keys inside of that object? Given a path of a tree item, we can use `reduce` to easily access the items inside:

```
>>> import json
>>> import functools
>>> import collections

>>> def tree():
...     return collections.defaultdict(tree)

# Build the tree:
>>> taxonomy = tree()
>>> reptilia = taxonomy['Chordata']['Vertebrata']['Reptilia']
>>> reptilia['Squamata']['Serpentes']['Pythonidae'] = [
...     'Liasis', 'Morelia', 'Python']

# The actual contents of the tree
>>> print(json.dumps(taxonomy, indent=4))
{
    "Chordata": {
        "Vertebrata": {
            "Reptilia": {
                "Squamata": {
                    "Serpentes": {
                        "Pythonidae": [
                            "Liasis",
                            "Morelia",
                            "Python"
                        ]
                    }
                }
            }
        }
    }
}

# The path we wish to get
```

```
>>> path = 'Chordata.Vertebrata.Reptilia.Squamata.Serpentes'

# Split the path for easier access
>>> path = path.split('.')

# Now fetch the path using reduce to recursively fetch the items
>>> family = functools.reduce(lambda a, b: a[b], path, taxonomy)
>>> family.items()
dict_items([('Pythonidae', ['Liasis', 'Morelia', 'Python'])])

# The path we wish to get
>>> path = 'Chordata.Vertebrata.Reptilia.Squamata'.split('.')

>>> suborder = functools.reduce(lambda a, b: a[b], path, taxonomy)
>>> suborder.keys()
dict_keys(['Serpentes'])
```

And lastly, some people might be wondering why Python only has `fold_left` and no `fold_right`. In my opinion, you don't really need both of them as you can easily reverse the operation.

The regular `reduce`—the `fold left` operation:

```
fold_left = functools.reduce(
    lambda x, y: function(x, y),
    iterable,
    initializer,
)
```

The reverse—the `fold right` operation:

```
fold_right = functools.reduce(
    lambda x, y: function(y, x),
    reversed(iterable),
    initializer,
)
```

While this one is definitely very useful in purely functional languages—where these operations are used quite often—initially there were plans to remove the `reduce` function from Python with the introduction of Python 3. Luckily, that plan was modified, and instead of being removed, it has been moved from `reduce` to `functools.reduce`. There may not be many useful cases for `reduce`, but there are some cool use cases. Especially traversing recursive data structures is far more easily done using `reduce`, since it would otherwise involve more complicated loops or recursive functions.

itertools

The `itertools` library contains iterable functions inspired by those available in functional languages. All of these are iterable and have been constructed in such a way that only a minimal amount of memory is required to process even the largest of datasets. While you can easily write most of these functions yourself using a simple function, I would still recommend using the ones available in the `itertools` library. These are all fast, memory efficient, and—perhaps more importantly—tested.

Even though the titles of the paragraphs are capitalized, the functions themselves are not. Be careful not to accidently type `Accumulate` instead of `accumulate`.

accumulate – reduce with intermediate results

The `accumulate` function is very similar to the `reduce` function, which is why some languages actually have `accumulate` instead of `reduce` as the folding operator.

The major difference between the two is that the `accumulate` function returns the immediate results. This can be useful when summing the results of a company's sales, for example:

```
>>> import operator
>>> import itertools

# Sales per month
>>> months = [10, 8, 5, 7, 12, 10, 5, 8, 15, 3, 4, 2]
>>> list(itertools.accumulate(months, operator.add))
[10, 18, 23, 30, 42, 52, 57, 65, 80, 83, 87, 89]
```

It should be noted that the `operator.add` function is actually optional in this case as the default behavior of accumulate is to sum the results. In some other languages and libraries, this function is called `cumsum` (cumulative sum).

chain – combining multiple results

The `chain` function is a simple but useful function that combines the results of multiple iterators. Very simple but also very useful if you have multiple lists, iterators, and so on—just combine them with a simple chain:

```
>>> import itertools
>>> a = range(3)
```

```
>>> b = range(5)
>>> list(itertools.chain(a, b))
[0, 1, 2, 0, 1, 2, 3, 4]
```

It should be noted that there is a small variant of `chain` that accepts an iterable containing iterables, namely `chain.from_iterable`. They work nearly identically, except for the fact that you need to pass along an iterable item instead of passing a list of arguments. Your initial response might be that this can be achieved simply by unpacking the (`*args`) tuple, as we will see in *Chapter 6, Generators and Coroutines – Infinity, One Step at a Time*. However, this is not always the case. For now, just remember that if you have a iterable containing iterables, the easiest method is to use `itertools.chain.from_iterable`.

combinations – combinatorics in Python

The `combinations` iterator produces results exactly as you would expect from the mathematical definition. All combinations with a specific length from a given list of items:

```
>>> import itertools
>>> list(itertools.combinations(range(3), 2))
[(0, 1), (0, 2), (1, 2)]
```

The `combinations` function gives all possible combinations of the given items of a given length. The number of possible combinations is given by the binomial coefficient, the `nCr` button on many calculators. It is commonly denoted as follows:

$$\binom{n}{k}$$

We have `n=2` and `k=4` in this case.

Here is the variant with repetition of elements:

```
>>> import itertools
>>> list(itertools.combinations_with_replacement(range(3), 2))
[(0, 0), (0, 1), (0, 2), (1, 1), (1, 2), (2, 2)]
```

The `combinations_with_repetitions` function is very similar to the regular `combinations` function, except that the items can be combined with themselves as well. To calculate the number of results, the binomial coefficient described earlier can be used with the parameters as `n=n+k-1` and `k=k`.

Let's look at a little combination of combinations and chain for generating a `powerset`:

```
>>> import itertools

>>> def powerset(iterable):
...     return itertools.chain.from_iterable(
...         itertools.combinations(iterable, i)
...         for i in range(len(iterable) + 1))
>>> list(powerset(range(3)))
[(), (0,), (1,), (2,), (0, 1), (0, 2), (1, 2), (0, 1, 2)]
```

The `powerset` is essentially the combined result of all combinations from `0` to `n`, meaning that it also includes elements with zero items (the empty set, or `()`), elements with `1` item, and all the way up to `n`. The number of items in the `powerset` is easily calculated using the power operator: `2**n`.

permutations – combinations where the order matters

The `permutations` function is quite similar to the `combinations` function. The only real difference is that `(a, b)` is considered distinct from `(b, a)`. In other words, the order matters:

```
>>> import itertools
>>> list(itertools.permutations(range(3), 2))
[(0, 1), (0, 2), (1, 0), (1, 2), (2, 0), (2, 1)]
```

compress – selecting items using a list of Booleans

The `compress` function is one of those that you won't need too often, but it can be very useful when you do need it. It applies a Boolean filter to your iterable, making it return only the ones you actually need. The most important thing to note here is that it's all executed lazily and that `compress` will stop if either the data or the selectors collection is exhausted. So, even with infinite ranges, it works without a hitch:

```
>>> import itertools
>>> list(itertools.compress(range(1000), [0, 1, 1, 1, 0, 1]))
[1, 2, 3, 5]
```

dropwhile/takewhile – selecting items using a function

The `dropwhile` function will drop all results until a given predicate evaluates to true. This can be useful if you are waiting for a device to finally return an expected result. That's a bit difficult to demonstrate here, so I'll just show an example with the basic usage—waiting for a number greater than `3`:

```
>>> import itertools
>>> list(itertools.dropwhile(lambda x: x <= 3, [1, 3, 5, 4, 2]))
[5, 4, 2]
```

As you might expect, the `takewhile` function is the reverse of this. It will simply return all rows until the predicate turns false:

```
>>> import itertools
>>> list(itertools.takewhile(lambda x: x <= 3, [1, 3, 5, 4, 2]))
[1, 3]
```

Simply adding the two will give you the original result again.

count – infinite range with decimal steps

The `count` function is quite similar to the `range` function, but there are two significant differences.

The first is that this range is infinite, so don't even try to do `list(itertools.count())`. You'll definitely run out of memory immediately and it might even freeze your system.

The second difference is that unlike the `range` function, you can actually use floating-point numbers here, so there is no need of whole/integer numbers.

Since listing the entire range will kill our Python interpreter, we'll simply use `zip` both to limit the results and to compare the results of the regular `range` function. In a later paragraph, we will see a more convenient option using `itertools.islice`. The `count` function takes two optional parameters: a `start` parameter, which defaults to `0`, and a `step` parameter, which defaults to `1`:

```
>>> import itertools

# Except for being infinite, the standard version returns the same
# results as the range function does.
>>> for a, b in zip(range(3), itertools.count()):
...     a, b
(0, 0)
(1, 1)
(2, 2)

# With a different starting point the results are still the same
>>> for a, b in zip(range(5, 8), itertools.count(5)):
...     a, b
(5, 5)
(6, 6)
(7, 7)

# And a different step works the same as well
>>> for a, b in zip(range(5, 10, 2), itertools.count(5, 2)):
...     a, b
(5, 5)
(7, 7)
(9, 9)

# Unless you try to use floating point numbers
>>> range(5, 10, 0.5)
Traceback (most recent call last):
    ...
```

```
TypeError: 'float' object cannot be interpreted as an integer


# Which does work for count
>>> for a, b in zip(range(5, 10), itertools.count(5, 0.5)):
...     a, b
(5, 5)
(6, 5.5)
(7, 6.0)
(8, 6.5)
(9, 7.0)
```

The `itertools.islice` function is also very useful in conjunction with `itertools.count`, as we'll see in a later paragraph.

groupby – grouping your sorted iterable

The `groupby` function is a really convenient function for grouping results. The usage and use cases are probably clear, but there are some important things to keep in mind when using this function:

- The input needs to be sorted by the `group` parameter. Otherwise, it will be added as a separate group.
- The results are available for use only once. So, after processing a group, it will not be available anymore.

Here is an example of the proper use of `groupby`:

```
>>> import itertools
>>> items = [('a', 1), ('a', 2), ('b', 2), ('b', 0), ('c', 3)]

>>> for group, items in itertools.groupby(items, lambda x: x[0]):
...     print('%s: %s' % (group, [v for k, v in items]))
a: [1, 2]
b: [2, 0]
c: [3]
```

And then there are cases where you might get unexpected results:

```
>>> import itertools
>>> items = [('a', 1), ('b', 0), ('b', 2), ('a', 2), ('c', 3)]
>>> groups = dict()
```

```
>>> for group, items in itertools.groupby(items, lambda x: x[0]):
...     groups[group] = items
...     print('%s: %s' % (group, [v for k, v in items]))
a: [1]
b: [0, 2]
a: [2]
c: [3]

>>> for group, items in sorted(groups.items()):
...     print('%s: %s' % (group, [v for k, v in items]))
a: []
b: []
c: []
```

Now we see two groups containing `a`. So, make sure you sort by the grouping parameter before trying to group. Additionally, walking through the same group a second time offers no results. This can be fixed easily using `groups[group] = list(items)` instead, but it can give quite a few unexpected bugs if you are not aware of this.

islice – slicing any iterable

When working with the `itertools` functions, you might notice that you cannot slice these objects. That is because they are generators, a topic that we will discuss in *Chapter 6, Generators and Coroutines – Infinity, One Step at a Time*. Luckily, the `itertools` library has a function for slicing these objects as well—`islice`.

Let's take `itertools.counter` from before as an example:

```
>>> import itertools
>>> list(itertools.islice(itertools.count(), 2, 7))
[2, 3, 4, 5, 6]
```

So, instead of the regular `slice`:

```
itertools.count()[:10]
```

We enter the `slice` parameters to the function:

```
itertools.islice(itertools.count(), 10)
```

What you should note from this is actually more than the inability to slice the objects. It is not just that slicing doesn't work, but it is not possible to get the length either—at least not without counting all items separately—and with infinite iterators, even that is not possible. The only understanding you actually get from a generator is that you can fetch items one at a time. You won't even know in advance whether you're at the end of the generator or not.

Summary

For some reason, functional programming is a paradigm that scares many people, but really it shouldn't. The most important difference between functional and procedural programming (within Python) is the mindset. Everything is executed using simple (and often translations of the mathematical equivalent) functions without any storage of variables. Simply put, a functional program consists of many functions having a simple input and output, without using (or even having) any outside scope or context to access. Python is not a purely functional language, so it is easy to cheat and work outside of the local scope, but that is not recommended.

This chapter covered the basics of functional programming within Python and some of the mathematics behind it. In addition to this, some of the many useful libraries that can be used in a very convenient way by using functional programming were covered.

The most important outtakes should be the following:

- Lambda statements are not inherently bad but it would be best to make them use variables from the local scope only, and they should not be longer than a single line.
- Functional programming can be very powerful, but it has a tendency to quickly become unreadable. Care must be taken.
- `list/dict/set` comprehensions are very useful, but they should generally not be nested, and for the purpose of readability, they should be kept short as well.

Ultimately, it is a matter of preference. For the sake of readability, I recommend limiting the usage of the functional paradigm when there is no obvious benefit. Having said that, when executed correctly, it can be a thing of beauty.

Next up are decorators—methods to wrap your functions and classes in other functions and/or classes to modify their behavior and extend their functionality.

5
Decorators – Enabling Code Reuse by Decorating

In this chapter, you are going to learn about Python decorators. Decorators are essentially function/class wrappers that can be used to modify the input, output, or even the function/class itself before executing it. This type of wrapping can just as easily be achieved by having a separate function that calls the inner function, or via mixins. As is the case with many Python constructs, decorators are not the only way to reach the goal but are definitely convenient in many cases.

While you can live perfectly without knowing too much about decorators, they give you a lot of "reuse power" and are therefore used heavily in framework libraries such as web frameworks. Python actually comes bundled with some useful decorators, most notably the `property` decorator.

There are, however, some particularities to take note of: wrapping a function creates a new function and makes it harder to reach the inner function and its properties. One example of this is the `help(function)` functionality of Python; by default, you will lose function properties such as the help text and the module the function exists in.

This chapter will cover the usage of both function and class decorators as well as the intricate details you need to know when decorating functions within classes.

The following are the topics covered:

- Decorating functions
- Decorating class functions
- Decorating classes
- Using classes as decorators
- Useful decorators in the Python standard library

Decorating functions

Essentially, a decorator is nothing more than a function or class wrapper. If we have a function called `spam` and a decorator called `eggs`, then the following would decorate `spam` with `eggs`:

```
spam = eggs(spam)
```

To make the syntax easier to use, Python has a special syntax for this case. So, instead of adding a line such as the preceding one below the function, you can simply decorate a function using the `@` operator:

```
@eggs
def spam():
    pass
```

The decorator simply receives the function and returns a—usually different—function. The simplest possible decorator is:

```
def eggs(function):
    return function
```

Looking at the earlier example, we realize that this gets `spam` as the argument for `function` and returns that function again, effectively changing nothing. Most decorators nest functions, however. The following decorator will print all arguments sent to `spam` and pass them to `spam` unmodified:

```
>>> import functools


>>> def eggs(function):
...     @functools.wraps(function)
...     def _eggs(*args, **kwargs):
...         print('%r got args: %r and kwargs: %r' % (
...             function.__name__, args, kwargs))
...         return function(*args, **kwargs)
...
...     return _eggs

>>> @eggs
... def spam(a, b, c):
```

```
...     return a * b + c


>>> spam(1, 2, 3)
'spam' got args: (1, 2, 3) and kwargs: {}
5
```

This should indicate how powerful decorators can be. By modifying `*args` and `**kwargs`, you can add, modify and remove arguments completely. Additionally, the return statement can be modified as well. Instead of `return function(...)`, you can return something completely different if you wish.

Why functools.wraps is important

Whenever you are writing a decorator, always be sure to add `functools.wraps` to wrap the inner function. Without wrapping it, you will lose all properties from the original function, which can lead to confusion. Take a look at the following code without `functools.wraps`:

```
>>> def eggs(function):
...     def _eggs(*args, **kwargs):
...         return function(*args, **kwargs)
...     return _eggs

>>> @eggs
... def spam(a, b, c):
...     '''The spam function Returns a * b + c'''
...     return a * b + c


>>> help(spam)
Help on function _eggs in module ...:
<BLANKLINE>
_eggs(*args, **kwargs)
<BLANKLINE>

>>> spam.__name__
'_eggs'
```

Now, our `spam` method has no documentation anymore and the name is gone. It has been renamed to `_eggs`. Since we are indeed calling `_eggs`, this is understandable, but it's very inconvenient for code that relies on this information. Now we will try the same code with the minor difference; we will use `functools.wraps`:

```
>>> import functools


>>> def eggs(function):
...     @functools.wraps(function)
...     def _eggs(*args, **kwargs):
...         return function(*args, **kwargs)
...     return _eggs

>>> @eggs
... def spam(a, b, c):
...     '''The spam function Returns a * b + c'''
...     return a * b + c


>>> help(spam)
Help on function spam in module ...:
<BLANKLINE>
spam(a, b, c)
    The spam function Returns a * b + c
<BLANKLINE>

>>> spam.__name__
'spam'
```

Without any further changes, we now have documentation and the expected function name. The working of `functools.wraps` is nothing magical though; it simply copies and updates several attributes. Specifically, the following attributes are copied:

- `__doc__`
- `__name__`
- `__module__`
- `__annotations__`
- `__qualname__`

Additionally, `__dict__` is updated using `_eggs.__dict__.update(spam.__dict__)`, and a new property called `__wrapped__` is added, which contains the original (`spam` in this case) function. The actual `wraps` function is available in the `functools.py` file of your Python distribution.

How are decorators useful?

The use cases for decorators are plentiful, but some of the most useful cases are with debugging. More extensive examples of this will be covered in *Chapter 11*, *Debugging – Solving the Bugs* but I can give you a sneak preview of how to use decorators to keep track of what your code is doing.

Let's assume you have a bunch of functions that may or may not be called, and you're not entirely sure what kind of input and output each of these is getting. In this case, you could, of course, modify the function and add some print statements at the beginning and the end to print the output. This quickly gets tedious, however, and it's one of those cases where a simple decorator will make it easy to do the same thing.

For this example, we are using a very simple function, but we all know that in real life, we're not always that lucky:

```
>>> def spam(eggs):
...     return 'spam' * (eggs % 5)
...
>>> output = spam(3)
```

Let's take our simple `spam` function and add some output so that we can see what happens internally:

```
>>> def spam(eggs):
...     output = 'spam' * (eggs % 5)
...     print('spam(%r): %r' % (eggs, output))
...     return output
...
>>> output = spam(3)
spam(3): 'spamspamspam'
```

While this works, wouldn't it be far nicer to have a little decorator that takes care of this problem?

```
>>> def debug(function):
...     @functools.wraps(function)
...     def _debug(*args, **kwargs):
```

```
...         output = function(*args, **kwargs)
...         print('%s(%r, %r): %r' % (function.__name__, args, kwargs,
output))
...         return output
...     return _debug
...
>>>
>>> @debug
... def spam(eggs):
...     return 'spam' * (eggs % 5)
...
>>> output = spam(3)
spam((3,), {}): 'spamspamspam'
```

Now we have a decorator that we can easily reuse for any function that prints the input, output, and function name. This type of decorator can also be very useful for logging applications, as we will see in *Chapter 10*, *Testing and Logging – Preparing for Bugs*. It should be noted that you can use this example even if you are not able to modify the module containing the original code. We can wrap the function locally and even monkey-patch the module if needed:

```
import some_module


# Regular call
some_module.some_function()

# Wrap the function
debug_some_function = debug(some_module.some_function)

# Call the debug version
debug_some_function()

# Monkey patch the original module
some_module.some_function = debug_some_function

# Now this calls the debug version of the function
some_module.some_function()
```

Naturally, monkey-patching is not a good idea in production code, but it can be very useful when debugging.

Memoization using decorators

Memoization is a simple trick for making some code run a bit faster. The basic trick here is to store a mapping of the input and expected output so that you have to calculate a value only once. One of the most common examples of this technique is when demonstrating the naïve (recursive) Fibonacci function:

```
>>> import functools

>>> def memoize(function):
...     function.cache = dict()
...
...     @functools.wraps(function)
...     def _memoize(*args):
...         if args not in function.cache:
...             function.cache[args] = function(*args)
...         return function.cache[args]
...     return _memoize

>>> @memoize
... def fibonacci(n):
...     if n < 2:
...         return n
...     else:
...         return fibonacci(n - 1) + fibonacci(n - 2)

>>> for i in range(1, 7):
...     print('fibonacci %d: %d' % (i, fibonacci(i)))
fibonacci 1: 1
fibonacci 2: 1
fibonacci 3: 2
fibonacci 4: 3
fibonacci 5: 5
fibonacci 6: 8

>>> fibonacci.__wrapped__.cache
{(5,): 5, (0,): 0, (6,): 8, (1,): 1, (2,): 1, (3,): 2, (4,): 3}
```

While this example would work just fine without any memoization, for larger numbers, it would kill the system. For `n=2`, the function would execute `fibonacci(n - 1)` and `fibonacci(n - 2)` recursively, effectively giving an exponential time complexity. Also, effectively for `n=30`, the Fibonacci function is called 2,692,537 times which is still doable nonetheless. At `n=40`, it is going to take you quite a very long time to calculate.

The memoized version, however, doesn't even break a sweat and only needs to execute `31` times for `n=30`.

This decorator also shows how a context can be attached to a function itself. In this case, the cache property becomes a property of the internal (wrapped `fibonacci`) function so that an extra `memoize` decorator for a different object won't clash with any of the other decorated functions.

Note, however, that implementing the memoization function yourself is generally not that useful anymore since Python introduced `lru_cache` (least recently used cache) in Python 3.2. The `lru_cache` is similar to the preceding memoize function but a bit more advanced. It only maintains a fixed (`128` by default) cache size to save memory and uses some statistics to check whether the cache size should be increased.

To demonstrate how `lru_cache` works internally, we will calculate `fibonacci(100)`, which would keep our computer busy until the end of the universe without any caching. Moreover, to make sure that we can actually see how many times the `fibonacci` function is being called, we'll add an extra decorator that keeps track of the count, as follows:

```
>>> import functools


# Create a simple call counting decorator
>>> def counter(function):
...     function.calls = 0
...     @functools.wraps(function)
...     def _counter(*args, **kwargs):
...         function.calls += 1
...         return function(*args, **kwargs)
...     return _counter


# Create a LRU cache with size 3
>>> @functools.lru_cache(maxsize=3)
... @counter
... def fibonacci(n):
```

```
...     if n < 2:
...         return n
...     else:
...         return fibonacci(n - 1) + fibonacci(n - 2)

>>> fibonacci(100)
354224848179261915075

# The LRU cache offers some useful statistics
>>> fibonacci.cache_info()
CacheInfo(hits=98, misses=101, maxsize=3, currsize=3)

# The result from our counter function which is now wrapped both by
# our counter and the cache
>>> fibonacci.__wrapped__.__wrapped__.calls
101
```

You might wonder why we need only 101 calls with a cache size of `3`. That's because we recursively require only `n - 1` and `n - 2`, so we have no need of a larger cache in this case. With others, it would still be useful though.

Additionally, this example shows the usage of two decorators for a single function. You can see these as the layers of an onion. The first one is the outer layer and it works towards the inside. When calling `fibonacci`, `lru_cache` will be called first because it's the first decorator in the list. Assuming there is no cache available yet, the `counter` decorator will be called. Within the counter, the actual `fibonacci` function will be called.

Returning the values works in the reverse order, of course; `fibonacci` returns its value to `counter`, which passes the value along to `lru_cache`.

Decorators with (optional) arguments

The previous examples mostly used simple decorators without any arguments. As we have already seen with `lru_cache`, decorators can accept arguments as well since they are just regular functions, but this adds an extra layer to a decorator. This means that adding an argument can be as simple as the following:

```
>>> import functools

>>> def add(extra_n=1):
```

```
...     'Add extra_n to the input of the decorated function'
...
...     # The inner function, notice that this is the actual
...     # decorator
...     def _add(function):
...         # The actual function that will be called
...         @functools.wraps(function)
...         def __add(n):
...             return function(n + extra_n)
...
...         return __add
...
...     return _add

>>> @add(extra_n=2)
... def eggs(n):
...     return 'eggs' * n

>>> eggs(2)
'eggseggseggseggs'
```

Optional arguments are a different matter, however, because they make the extra function layer optional. With arguments, you need three layers, but without arguments, you need only two layers. Since decorators are essentially regular functions that return functions, the difference would be to return the sub-function or the sub-sub-function, based on the parameters. This leaves just one issue—detecting whether the parameter is a function or a regular parameter. To illustrate, with the parameters the actual call looks like the following:

```
add(extra_n=2)(eggs)(2)
```

Whereas the call without arguments would look like this:

```
add(eggs)(2)
```

To detect whether the decorator was called with a function or a regular argument as a parameter, we have several options, none of which are completely ideal in my opinion:

- Using keyword arguments for decorator arguments so that the regular argument will always be the function
- Detecting whether the first and only argument is callable

In my opinion, the first one—using keyword arguments—is the better of the two options because it is somewhat more explicit and leaves less room for confusion. The second option could be problematic if, for some reason, your argument is callable as well.

Using the first method, the normal (non-keyword) argument has to be the decorated function and the other two checks can still apply. We can still check whether the function is indeed callable and whether there is only a single argument available. Here is an example using a modified version of the previous example:

```
>>> import functools


>>> def add(*args, **kwargs):
...     'Add n to the input of the decorated function'
...
...     # The default kwargs, we don't store this in kwargs
...     # because we want to make sure that args and kwargs
...     # can't both be filled
...     default_kwargs = dict(n=1)
...
...     # The inner function, notice that this is actually a
...     # decorator itself
...     def _add(function):
...         # The actual function that will be called
...         @functools.wraps(function)
...         def __add(n):
...             default_kwargs.update(kwargs)
...             return function(n + default_kwargs['n'])
...
...         return __add
...
...     if len(args) == 1 and callable(args[0]) and not kwargs:
...         # Decorator call without arguments, just call it
...         # ourselves
...         return _add(args[0])
...     elif not args and kwargs:
...         # Decorator call with arguments, this time it will
...         # automatically be executed with function as the
```

```
...         # first argument
...         default_kwargs.update(kwargs)
...         return _add
...     else:
...         raise RuntimeError('This decorator only supports '
...                            'keyword arguments')

>>> @add
... def spam(n):
...     return 'spam' * n

>>> @add(n=3)
... def eggs(n):
...     return 'eggs' * n

>>> spam(3)
'spamspamspamspam'

>>> eggs(2)
'eggseggseggseggseggs'

>>> @add(3)
... def bacon(n):
...     return 'bacon' * n
Traceback (most recent call last):
  ...
RuntimeError: This decorator only supports keyword arguments
```

Whenever you have the choice available, I recommend that you either have a decorator with arguments or without them, instead of having optional arguments. However, if you have a really good reason for making the arguments optional, then you have a relatively safe method of making this possible.

Creating decorators using classes

Similar to how we create regular function decorators, it is also possible to create decorators using classes instead. After all, a function is just a callable object and a class can implement the callable interface as well. The following decorator works similarly to the `debug` decorator we used earlier, but uses a class instead of a regular function:

```
>>> import functools


>>> class Debug(object):
...
...     def __init__(self, function):
...         self.function = function
...         # functools.wraps for classes
...         functools.update_wrapper(self, function)
...
...     def __call__(self, *args, **kwargs):
...         output = self.function(*args, **kwargs)
...         print('%s(%r, %r): %r' % (
...             self.function.__name__, args, kwargs, output))
...         return output



>>> @Debug
... def spam(eggs):
...     return 'spam' * (eggs % 5)
...
>>> output = spam(3)
spam((3,), {}): 'spamspamspam'
```

The only notable difference between functions and classes is that `functools.wraps` is now replaced with `functools.update_wrapper` in the `__init__` method.

Decorating class functions

Decorating class functions is very similar to regular functions, but you need to be aware of the required first argument, `self`—the class instance. You have most likely already used a few class function decorators. The `classmethod`, `staticmethod`, and `property` decorators for example, are used in many different projects. To explain how all this works, we will build our own versions of the `classmethod`, `staticmethod`, and `property` decorators. First, let's look at a simple decorator for class functions to show the difference from regular decorators:

```
>>> import functools


>>> def plus_one(function):
...     @functools.wraps(function)
...     def _plus_one(self, n):
...         return function(self, n + 1)
...     return _plus_one


>>> class Spam(object):
...     @plus_one
...     def get_eggs(self, n=2):
...         return n * 'eggs'


>>> spam = Spam()
>>> spam.get_eggs(3)
'eggseggseggseggs'
```

As is the case with regular functions, the class function decorator now gets passed along `self` as the instance. Nothing unexpected!

Skipping the instance – classmethod and staticmethod

The difference between a `classmethod` and a `staticmethod` is fairly simple. The `classmethod` passes a class object instead of a class instance (`self`), and `staticmethod` skips both the class and the instance entirely. This effectively makes `staticmethod` very similar to a regular function outside of a class.

Before we recreate `classmethod` and `staticmethod`, we need to take a look at the expected behavior of these methods:

```
>>> import pprint


>>> class Spam(object):
...
...     def some_instancemethod(self, *args, **kwargs):
...         print('self: %r' % self)
...         print('args: %s' % pprint.pformat(args))
...         print('kwargs: %s' % pprint.pformat(kwargs))
...
...     @classmethod
...     def some_classmethod(cls, *args, **kwargs):
...         print('cls: %r' % cls)
...         print('args: %s' % pprint.pformat(args))
...         print('kwargs: %s' % pprint.pformat(kwargs))
...
...     @staticmethod
...     def some_staticmethod(*args, **kwargs):
...         print('args: %s' % pprint.pformat(args))
...         print('kwargs: %s' % pprint.pformat(kwargs))

# Create an instance so we can compare the difference between
# executions with and without instances easily
>>> spam = Spam()

# With an instance (note the lowercase spam)
>>> spam.some_instancemethod(1, 2, a=3, b=4)
self: <...Spam object at 0x...>
args: (1, 2)
kwargs: {'a': 3, 'b': 4}

# Without an instance (note the capitalized Spam)
>>> Spam.some_instancemethod()
```

```
Traceback (most recent call last):

    ...
TypeError: some_instancemethod() missing 1 required positional argument:
'self'

# But what if we add parameters? Be very careful with these!
# Our first argument is now used as an argument, this can give
# very strange and unexpected errors
>>> Spam.some_instancemethod(1, 2, a=3, b=4)
self: 1
args: (2,)
kwargs: {'a': 3, 'b': 4}


# Classmethods are expectedly identical
>>> spam.some_classmethod(1, 2, a=3, b=4)
cls: <class '...Spam'>
args: (1, 2)
kwargs: {'a': 3, 'b': 4}

>>> Spam.some_classmethod()
cls: <class '...Spam'>
args: ()
kwargs: {}

>>> Spam.some_classmethod(1, 2, a=3, b=4)
cls: <class '...Spam'>
args: (1, 2)
kwargs: {'a': 3, 'b': 4}


# Staticmethods are also identical
>>> spam.some_staticmethod(1, 2, a=3, b=4)
args: (1, 2)
kwargs: {'a': 3, 'b': 4}

>>> Spam.some_staticmethod()
```

```
args: ()
kwargs: {}


>>> Spam.some_staticmethod(1, 2, a=3, b=4)
args: (1, 2)
kwargs: {'a': 3, 'b': 4}
```

Note that calling `some_instancemethod` without an instance results in an error whereby `self` is missing. As expected (since we didn't instantiate the class in that case), for the version with the arguments, it seems to work but it is actually broken. This is because the first argument is now assumed to be `self`. This is obviously incorrect in this case, where you pass an integer, but if you had passed along some other class instance, this could be a source of very strange bugs. Both `classmethod` and `staticmethod` handle this correctly.

Before we can continue with decorators, you need to be aware of how Python descriptors function. Descriptors can be used to modify the binding behavior of object attributes. This means that if a descriptor is used as the value of an attribute, you can modify which value is being set, get, and deleted when these operations are called on the attribute. Here is a basic example of this behavior:

```
>>> class MoreSpam(object):
...
...     def __init__(self, more=1):
...         self.more = more
...
...     def __get__(self, instance, cls):
...         return self.more + instance.spam
...
...     def __set__(self, instance, value):
...         instance.spam = value - self.more


>>> class Spam(object):
...
...     more_spam = MoreSpam(5)
...
...     def __init__(self, spam):
...         self.spam = spam
```

```
>>> spam = Spam(1)
>>> spam.spam
1
>>> spam.more_spam
6

>>> spam.more_spam = 10
>>> spam.spam
5
```

As you can see, whenever we set or get values from `more_spam`, it actually calls `__get__` or `__set__` on `MoreSpam`. A very useful feat for automatic conversions and type checking, the `property` decorator we will see in the next paragraph is just a more convenient implementation of this technique.

Now that we know how descriptors work, we can continue with creating the `classmethod` and `staticmethod` decorators. For these two, we simply need to modify `__get__` instead of `__call__` so that we can control which type of instance (or none at all) is passed along:

```
import functools


class ClassMethod(object):

    def __init__(self, method):
        self.method = method

    def __get__(self, instance, cls):
        @functools.wraps(self.method)
        def method(*args, **kwargs):
            return self.method(cls, *args, **kwargs)
        return method


class StaticMethod(object):

    def __init__(self, method):
        self.method = method

    def __get__(self, instance, cls):
        return self.method
```

The `ClassMethod` decorator still features a sub-function to actually produce a working decorator. Looking at the function, you can most likely guess how it functions. Instead of passing `instance` as the first argument to `self.method`, it passes `cls`.

`StaticMethod` is even simpler, because it completely ignores both the `instance` and the `cls`. It can just return the original method unmodified. Because it returns the original method without any modifications, we have no need for the `functools.wraps` call either.

Properties – smart descriptor usage

The `property` decorator is probably the most used decorator in Python land. It allows you to add getters/setters to existing instance properties so that you can add validators and modify your values before setting them to your instance properties. The `property` decorator can be used both as an assignment and as a decorator. The following example shows both syntaxes so that we know what to expect from the `property` decorator:

```
>>> class Spam(object):
...
...     def get_eggs(self):
...         print('getting eggs')
...         return self._eggs
...
...     def set_eggs(self, eggs):
...         print('setting eggs to %s' % eggs)
...         self._eggs = eggs
...
...     def delete_eggs(self):
...         print('deleting eggs')
...         del self._eggs
...
...     eggs = property(get_eggs, set_eggs, delete_eggs)
...
...     @property
...     def spam(self):
...         print('getting spam')
...         return self._spam
...
```

```
...     @spam.setter
...     def spam(self, spam):
...         print('setting spam to %s' % spam)
...         self._spam = spam
...
...     @spam.deleter
...     def spam(self):
...         print('deleting spam')
...         del self._spam


>>> spam = Spam()
>>> spam.eggs = 123
setting eggs to 123
>>> spam.eggs
getting eggs
123
>>> del spam.eggs
deleting eggs
```

Note that the `property` decorator works only if the class inherits `object`.

Similar to how we implemented the `classmethod` and `staticmethod` decorators, we need the Python descriptors again. This time, we require the full power of the descriptors, however—not just `__get__` but `__set__` and `__delete__` as well:

```
class Property(object):
    def __init__(self, fget=None, fset=None, fdel=None,
                 doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        # If no specific documentation is available, copy it
        # from the getter
        if fget and not doc:
            doc = fget.__doc__
        self.__doc__ = doc
```

```
        def __get__(self, instance, cls):
            if instance is None:
                # Redirect class (not instance) properties to
                # self
                return self
            elif self.fget:
                return self.fget(instance)
            else:
                raise AttributeError('unreadable attribute')

        def __set__(self, instance, value):
            if self.fset:
                self.fset(instance, value)
            else:
                raise AttributeError("can't set attribute")

        def __delete__(self, instance):
            if self.fdel:
                self.fdel(instance)
            else:
                raise AttributeError("can't delete attribute")

        def getter(self, fget):
            return type(self)(fget, self.fset, self.fdel)

        def setter(self, fset):
            return type(self)(self.fget, fset, self.fdel)

        def deleter(self, fdel):
            return type(self)(self.fget, self.fset, fdel)
```

As you can see, most of the `Property` implementation is simply an implementation of the descriptor methods. The `getter`, `setter`, and `deleter` functions are simply shortcuts for making the usage of the decorator possible, which is why we have to `return self` if no `instance` is available.

Naturally, there are more methods of achieving this effect. In the previous paragraph, we saw the bare descriptor implementation, and in our previous example, we saw the property decorator. A somewhat more generic solution for a class is to implement `__getattr__` or `__getattribute__`. Here's a simple demonstration:

```
>>> class Spam(object):
...     def __init__(self):
...         self.registry = {}
```

```
...
...     def __getattr__(self, key):
...         print('Getting %r' % key)
...         return self.registry.get(key, 'Undefined')
...
...     def __setattr__(self, key, value):
...         if key == 'registry':
...             object.__setattr__(self, key, value)
...         else:
...             print('Setting %r to %r' % (key, value))
...             self.registry[key] = value
...
...     def __delattr__(self, key):
...         print('Deleting %r' % key)
...         del self.registry[key]


>>> spam = Spam()

>>> spam.a
Getting 'a'
'Undefined'

>>> spam.a = 1
Setting 'a' to 1

>>> spam.a
Getting 'a'
1

>>> del spam.a
Deleting 'a'
```

The `__getattr__` method looks for the key in `instance.__dict__` first and is called only if it does not exist. That's why we never see a `__getattr__` for the registry attribute. The `__getattribute__` method is called in all cases, which makes it a bit more dangerous to use. With the `__getattribute__` method, you will need a specific exclusion for `registry` since it will be executed recursively if you try to access `self.registry`.

There is rarely a need to look at descriptors, but they are used by several internal Python processes, such as the `super()` method when inheriting classes.

Decorating classes

Python 2.6 introduced the class decorator syntax. As is the case with the function decorator syntax, this is not really a new technique either. Even without the syntax, a class can be decorated simply by executing `DecoratedClass = decorator(RegularClass)`. After the previous paragraphs, you should be familiar with writing decorators. Class decorators are no different from regular ones, except for the fact that they take a class instead of a function. As is the case with functions, this happens at declaration time and *not* at instantiating/calling time.

Because there are quite a few alternative ways to modify how classes work, such as standard inheritance, mixins, and metaclasses (more about that in *Chapter 8*, *Metaclasses – Making Classes (Not Instances) Smarter*), class decorators are never strictly needed. This does not reduce their usefulness, but it does offer an explanation of why you will most likely not see too many examples of class decorating in the wild.

Singletons – classes with a single instance

Singletons are classes that always allow only a single instance to exist. So, instead of getting an instance specifically for your call, you always get the same one. These can be very useful for things such as a database connection pool, where you don't want to keep opening connections all of the time but want to reuse the original ones:

```
>>> import functools


>>> def singleton(cls):
...     instances = dict()
...     @functools.wraps(cls)
...     def _singleton(*args, **kwargs):
...         if cls not in instances:
...             instances[cls] = cls(*args, **kwargs)
...         return instances[cls]
...     return _singleton

>>> @singleton
... class Spam(object):
...     def __init__(self):
```

```
...         print('Executing init')


>>> a = Spam()
Executing init
>>> b = Spam()

>>> a is b
True

>>> a.x = 123
>>> b.x
123
```

As you can see in the `a is b` comparison, both objects have the same identity, so we can conclude that they are indeed the same object. As is the case with regular decorators, due to the `functools.wraps` functionality, we can still access the original class through `Spam.__wrapped__` if needed.

The `is` operator compares objects by identity, which is implemented as the memory address in CPython. If `a is b` returns `True`, we can conclude that both `a` and `b` are the same instance.

Total ordering – sortable classes the easy way

At some point or the other, you have probably needed to sort data structures. While this is easily achievable using the key parameter to the `sorted` function, there is a more convenient way if you need to do this often—by implementing the `__gt__`, `__ge__`, `__lt__`, `__le__`, and `__eq__` functions. That seems a bit verbose, doesn't it? If you want the best performance, it's still a good idea, but if you can take a tiny performance hit and some slightly more complicated stack traces, then `total_ordering` might be a nice alternative. The `total_ordering` class decorator can implement all required sort functions based on a class that possesses an `__eq__` function and one of the comparison functions (`__lt__`, `__le__`, `__gt__`, or `__ge__`). This means you can seriously shorten your function definitions. Let's compare the regular one and the one using the `total_ordering` decorator:

```
>>> import functools


>>> class Value(object):
```

```
...     def __init__(self, value):
...         self.value = value
...
...     def __repr__(self):
...         return '<%s[%d]>' % (self.__class__, self.value)


>>> class Spam(Value):
...     def __gt__(self, other):
...         return self.value > other.value
...
...     def __ge__(self, other):
...         return self.value >= other.value
...
...     def __lt__(self, other):
...         return self.value < other.value
...
...     def __le__(self, other):
...         return self.value <= other.value
...
...     def __eq__(self, other):
...         return self.value == other.value


>>> @functools.total_ordering
... class Egg(Value):
...     def __lt__(self, other):
...         return self.value < other.value
...
...     def __eq__(self, other):
...         return self.value == other.value


>>> numbers = [4, 2, 3, 4]
>>> spams = [Spam(n) for n in numbers]
>>> eggs = [Egg(n) for n in numbers]

>>> spams
```

```
[<<class 'H05.Spam'>[4]>, <<class 'H05.Spam'>[2]>,
<<class 'H05.Spam'>[3]>, <<class 'H05.Spam'>[4]>]

>>> eggs
[<<class 'H05.Egg'>[4]>, <<class 'H05.Egg'>[2]>,
<<class 'H05.Egg'>[3]>, <<class 'H05.Egg'>[4]>]

>>> sorted(spams)
[<<class 'H05.Spam'>[2]>, <<class 'H05.Spam'>[3]>,
<<class 'H05.Spam'>[4]>, <<class 'H05.Spam'>[4]>]

>>> sorted(eggs)
[<<class 'H05.Egg'>[2]>, <<class 'H05.Egg'>[3]>,
<<class 'H05.Egg'>[4]>, <<class 'H05.Egg'>[4]>]

# Sorting using key is of course still possible and in this case
# perhaps just as easy:
>>> values = [Value(n) for n in numbers]
>>> values
[<<class 'H05.Value'>[4]>, <<class 'H05.Value'>[2]>,
<<class 'H05.Value'>[3]>, <<class 'H05.Value'>[4]>]

>>> sorted(values, key=lambda v: v.value)
[<<class 'H05.Value'>[2]>, <<class 'H05.Value'>[3]>,
<<class 'H05.Value'>[4]>, <<class 'H05.Value'>[4]>]
```

Now, you might be wondering, "Why isn't there a class decorator to make a class sortable using a specified key property?" Well, that might indeed be a good idea for the `functools` library but it isn't there yet. So let's see how we would implement something like it:

```
>>> def sort_by_attribute(attr, keyfunc=getattr):
...     def _sort_by_attribute(cls):
...         def __gt__(self, other):
...             return getattr(self, attr) > getattr(other, attr)
...
...         def __ge__(self, other):
...             return getattr(self, attr) >= getattr(other, attr)
...
```

```
...         def __lt__(self, other):
...             return getattr(self, attr) < getattr(other, attr)
...
...         def __le__(self, other):
...             return getattr(self, attr) <= getattr(other, attr)
...
...         def __eq__(self, other):
...             return getattr(self, attr) <= getattr(other, attr)
...
...         cls.__gt__ = __gt__
...         cls.__ge__ = __ge__
...         cls.__lt__ = __lt__
...         cls.__le__ = __le__
...         cls.__eq__ = __eq__
...
...         return cls
...     return _sort_by_attribute


>>> class Value(object):
...     def __init__(self, value):
...         self.value = value
...
...     def __repr__(self):
...         return '<%s[%d]>' % (self.__class__, self.value)


>>> @sort_by_attribute('value')
... class Spam(Value):
...     pass


>>> numbers = [4, 2, 3, 4]
>>> spams = [Spam(n) for n in numbers]
>>> sorted(spams)
[<<class '...Spam'>[2]>, <<class '...Spam'>[3]>,
<<class '...Spam'>[4]>, <<class '...Spam'>[4]>]
```

Certainly, this greatly simplifies the making of a sortable class. And if you would rather have your own key function instead of `getattr`, it's even easier. Simply replace the `getattr(self, attr)` call with `key_function(self)`, do that for `other` as well, and change the argument for the decorator to your function. You can even use that as the base function and implement `sort_by_attribute` by simply passing a wrapped `getattr` function.

Useful decorators

In addition to the ones already mentioned in this chapter, Python comes bundled with a few other useful decorators. There are some that aren't in the standard library (yet?).

Single dispatch – polymorphism in Python

If you've used C++ or Java before, you're probably used to having ad hoc polymorphism available—different functions being called depending on the argument types. Python being a dynamically typed language, most people would not expect the possibility of a single dispatch pattern. Python, however, is a language that is not only dynamically typed but also strongly typed, which means we can rely on the type we receive.

A dynamically typed language does not require strict type definitions. On the other hand, a language such as C would require the following to declare an integer:

```
int some_integer = 123;
```

Python simply accepts that your value has a type:

```
some_integer = 123
```

As opposed to languages such as JavaScript and PHP, however, Python does very little implicit type conversion. In Python, the following will return an error, whereas JavaScript would execute it without any problems:

```
'spam' + 5
```

In Python, the result is a `TypeError`. In Javascript, it's `'spam5'`.

The idea of single dispatch is that depending on the type you pass along, the correct function is called. Since `str + int` results in an error in Python, this can be very convenient to automatically convert your arguments before passing them to your function. This can be useful to separate the actual workings of your function from the type conversions.

Since Python 3.4, there is a decorator that makes it easily possible to implement the single dispatch pattern in Python. For one of those cases that you need to handle a specific type different from the normal execution. Here is the basic example:

```
>>> import functools


>>> @functools.singledispatch
... def printer(value):
...     print('other: %r' % value)

>>> @printer.register(str)
... def str_printer(value):
...     print(value)

>>> @printer.register(int)
... def int_printer(value):
...     printer('int: %d' % value)

>>> @printer.register(dict)
... def dict_printer(value):
...     printer('dict:')
...     for k, v in sorted(value.items()):
...         printer('    key: %r, value: %r' % (k, v))


>>> printer('spam')
spam

>>> printer([1, 2, 3])
other: [1, 2, 3]

>>> printer(123)
```

```
int: 123

>>> printer({'a': 1, 'b': 2})
dict:
    key: 'a', value: 1
    key: 'b', value: 2
```

See how, depending on the type, the other functions were called? This pattern can be very useful for reducing the complexity of a single function that takes several types of argument.

When naming the functions, make sure that you do not overwrite the original `singledispatch` function. If we had named `str_printer` as just `printer`, it would overwrite the initial `printer` function. This would make it impossible to access the original `printer` function and make all `register` operations after that fail as well.

Now, a slightly more useful example—differentiating between a filename and a file handler:

```
>>> import json
>>> import functools


>>> @functools.singledispatch
... def write_as_json(file, data):
...     json.dump(data, file)


>>> @write_as_json.register(str)
... @write_as_json.register(bytes)
... def write_as_json_filename(file, data):
...     with open(file, 'w') as fh:
...         write_as_json(fh, data)


>>> data = dict(a=1, b=2, c=3)
```

```
>>> write_as_json('test1.json', data)
>>> write_as_json(b'test2.json', 'w')
>>> with open('test3.json', 'w') as fh:
...     write_as_json(fh, data)
```

So now we have a single `write_as_json` function; it calls the right code depending on the type. If it's an `str` or `bytes` object, it will automatically open the file and call the regular version of `write_as_json`, which accepts file objects.

Writing a decorator that does this is not that hard to do, of course, but it's still quite convenient to have it in the base library. It most certainly beats a couple of `isinstance` calls in your function. To see which function will be called, you can use the `write_as_json.dispatch` function with a specific type. When passing along an `str`, you will get the `write_as_json_filename` function. It should be noted that the name of the dispatched functions is completely arbitrary. They are accessible as regular functions, of course, but you can name them anything you like.

To check the registered types, you can access the registry, which is a dictionary, through `write_as_json.registry`:

```
>>> write_as_json.registry.keys()
dict_keys([<class 'bytes'>, <class 'object'>, <class 'str'>])
```

Contextmanager, with statements made easy

Using the `contextmanager` class, we can make the creation of a context wrapper very easy. Context wrappers are used whenever you use a `with` statement. One example is the open function, which works as a context wrapper as well, allowing you to use the following code:

```
with open(filename) as fh:
    pass
```

Let's just assume for now that the `open` function is not usable as a context manager and that we need to build our own function to do this. The standard method of creating a context manager is by creating a class that implements the `__enter__` and `__exit__` methods, but that's a bit verbose. We can have it shorter and simpler:

```
>>> import contextlib



>>> @contextlib.contextmanager
... def open_context_manager(filename, mode='r'):
...     fh = open(filename, mode)
```

```
...     yield fh
...     fh.close()


>>> with open_context_manager('test.txt', 'w') as fh:
...     print('Our test is complete!', file=fh)
```

Simple, right? However, I should mention that for this specific case—the closing of objects—there is a dedicated function in `contextlib`, and it is even easier to use. Let's demonstrate it:

```
>>> import contextlib

>>> with contextlib.closing(open('test.txt', 'a')) as fh:
...     print('Yet another test', file=fh)
```

For a `file` object, this is of course not needed since it already functions as a context manager. However, some objects such as requests made by `urllib` don't support automatic closing in that manner and benefit from this function.

But wait; there's more! In addition to being usable in a `with` statement, the results of a `contextmanager` are actually usable as decorators since Python 3.2. In older Python versions, it was simply a small wrapper, but since Python 3.2 it's based on the `ContextDecorator` class, which makes it a decorator. The previous decorator isn't really suitable for that task since it yields a result (more about that in *Chapter 6*, *Generators and Coroutines – Infinity, One Step at a Time*), but we can think of other functions:

```
>>> @contextlib.contextmanager
... def debug(name):
...     print('Debugging %r:' % name)
...     yield
...     print('End of debugging %r' % name)


>>> @debug('spam')
... def spam():
...     print('This is the inside of our spam function')

>>> spam()
Debugging 'spam':
This is the inside of our spam function
End of debugging 'spam'
```

There are quite a few nice use cases for this, but at the very least, it's just a convenient way to wrap a function in a context without all the (nested) `with` statements.

Validation, type checks, and conversions

While checking for types is usually not the best way to go in Python, at times it can be useful if you know that you will need a specific type (or something that can be cast to that type). To facilitate this, Python 3.5 introduces a type hinting system so that you can do the following:

```
def spam(eggs: int):
    pass
```

Since Python 3.5 is not that common yet, here's a decorator that achieves the same with more advanced type checking. To allow for this type of checking, some magic has to be used, specifically the usage of the `inspect` module. Personally, I am not a great fan of inspecting code to perform tricks like these, as they are easy to break. This piece of code actually breaks when a regular decorator (one that doesn't copy `argspec`) is used between the function and this decorator, but it's a nice example nonetheless:

```
>>> import inspect
>>> import functools



>>> def to_int(name, minimum=None, maximum=None):
...     def _to_int(function):
...         # Use the method signature to map *args to named
...         # arguments
...         signature = inspect.signature(function)
...
...         # Unfortunately functools.wraps doesn't copy the
...         # signature (yet) so we do it manually.
...         # For more info: http://bugs.python.org/issue23764
...         @functools.wraps(function, ['__signature__'])
...         @functools.wraps(function)
...         def __to_int(*args, **kwargs):
...             # Bind all arguments to the names so we get a single
...             # mapping of all arguments
...             bound = signature.bind(*args, **kwargs)
...
```

```
...             # Make sure the value is (convertible to) an integer
...             default = signature.parameters[name].default
...             value = int(bound.arguments.get(name, default))
...
...             # Make sure it's within the allowed range
...             if minimum is not None:
...                 assert value >= minimum, (
...                     '%s should be at least %r, got: %r' %
...                     (name, minimum, value))
...
...             if maximum is not None:
...                 assert value <= maximum, (
...                     '%s should be at most %r, got: %r' %
...                     (name, maximum, value))
...
...             return function(*args, **kwargs)
...         return __to_int
...     return _to_int

>>> @to_int('a', minimum=10)
... @to_int('b', maximum=10)
... @to_int('c')
... def spam(a, b, c=10):
...     print('a', a)
...     print('b', b)
...     print('c', c)

>>> spam(10, b=0)
a 10
b 0
c 10

>>> spam(a=20, b=10)
a 20
b 10
c 10

>>> spam(1, 2, 3)
```

```
Traceback (most recent call last):
    ...
AssertionError: a should be at least 10, got: 1

>>> spam()
Traceback (most recent call last):
    ...
TypeError: 'a' parameter lacking default value

>>> spam('spam', {})
Traceback (most recent call last):
    ...
ValueError: invalid literal for int() with base 10: 'spam'
```

Because of the `inspect` magic, I'm still not sure whether I would recommend using the decorator like this. Instead, I would opt for a simpler version that uses no `inspect` whatsoever and simply parses the arguments from `kwargs`:

```
>>> import functools


>>> def to_int(name, minimum=None, maximum=None):
...     def _to_int(function):
...         @functools.wraps(function)
...         def __to_int(**kwargs):
...             value = int(kwargs.get(name))
...
...             # Make sure it's within the allowed range
...             if minimum is not None:
...                 assert value >= minimum, (
...                     '%s should be at least %r, got: %r' %
...                     (name, minimum, value))
...
...             if maximum is not None:
...                 assert value <= maximum, (
...                     '%s should be at most %r, got: %r' %
...                     (name, maximum, value))
...
```

```
...             return function(**kwargs)
...         return __to_int
...     return _to_int

>>> @to_int('a', minimum=10)
... @to_int('b', maximum=10)
... def spam(a, b):
...     print('a', a)
...     print('b', b)

>>> spam(a=20, b=10)
a 20
b 10

>>> spam(a=1, b=10)
Traceback (most recent call last):
    ...
AssertionError: a should be at least 10, got: 1
```

However, as demonstrated, supporting both `args` and `kwargs` is not impossible as long as you keep in mind that `__signature__` is not copied by default. Without `__signature__`, the inspect module won't know which parameters are allowed and which aren't.

The missing `__signature__` issue is currently being discussed and might be solved in a future Python version: `http://bugs.python.org/issue23764`.

Useless warnings – how to ignore them

Generally when writing Python, warnings are very useful the first time when you're actually writing the code. When executing it, however, it is not useful to get that same message every time you run your script/application. So, let's create some code that allows easy hiding of the expected warnings, but not all of them so that we can easily catch new ones:

```
import warnings
import functools


def ignore_warning(warning, count=None):
```

```
    def _ignore_warning(function):
        @functools.wraps(function)
        def __ignore_warning(*args, **kwargs):
            # Execute the code while recording all warnings
            with warnings.catch_warnings(record=True) as ws:
                # Catch all warnings of this type
                warnings.simplefilter('always', warning)
                # Execute the function
                result = function(*args, **kwargs)

            # Now that all code was executed and the warnings
            # collected, re-send all warnings that are beyond our
            # expected number of warnings
            if count is not None:
                for w in ws[count:]:
                    warnings.showwarning(
                        message=w.message,
                        category=w.category,
                        filename=w.filename,
                        lineno=w.lineno,
                        file=w.file,
                        line=w.line,
                    )

            return result
        return __ignore_warning
    return _ignore_warning


@ignore_warning(DeprecationWarning, count=1)
def spam():
    warnings.warn('deprecation 1', DeprecationWarning)
    warnings.warn('deprecation 2', DeprecationWarning)
```

Using this method, we can catch the first (expected) warning and still see the second (not expected) warning.

Summary

This chapter showed us some of the places where decorators can be used to make our code simpler and add some fairly complex behavior to very simple functions. Truthfully, most decorators are more complex than the regular function would have been by simply adding the functionality directly, but the added advantage of applying the same pattern to many functions and classes is generally well worth it.

Decorators have so many uses to make your functions and classes smarter and more convenient to use:

- Debugging
- Validation
- Argument convenience (pre-filling or converting arguments)
- Output convenience (converting the output to a specific type)

The most important takeaway of this chapter should be to never forget `functools.wraps` when wrapping a function. Debugging decorated functions can be rather difficult because of (unexpected) behavior modification, but losing attributes as well can make that problem much worse.

The next chapter will show us how and when to use `generators` and `coroutines`. This chapter has already shown us the usage of the `with` statement slightly, but `generators` and `coroutines` go much further with this. We will still be using decorators often though, so make sure you have a good understanding of how they work.

6

Generators and Coroutines – Infinity, One Step at a Time

A generator is a specific type of iterator that generates values through a function. While traditional methods build and return a `list` of items, a generator will simply `yield` every value separately at the moment when they are requested by the caller. This method has several benefits:

- Generators pause execution completely until the next value is yielded, which makes them completely lazy. If you fetch five items from a generator, only five items will be generated, so no other computation is needed.
- Generators have no need to save values. Whereas a traditional function would require creating a `list` and storing all results until they are returned, a generator only needs to store a single value.
- Generators can have infinite size. There is no requirement to stop at a certain point.

These benefits come at a price, however. The immediate results of these benefits are a few disadvantages:

- Until you are done processing, you never know how many values are left; it could even be infinite. This makes usage dangerous in some cases; executing `list(some_infinite_generator)` will run out of memory.
- You cannot slice generators.
- You cannot get specific items without yielding all values before that index.
- You cannot restart a generator. All values are yielded exactly once.

In addition to generators, there is a variation to the generator's syntax that creates coroutines. Coroutines are functions that allow for multitasking without requiring multiple threads or processes. Whereas generators can only yield values to the caller, coroutines actually receive values from the caller while it is still running. While this technique has a few limitations, if it suits your purpose, it can result in great performance at a very little cost.

In short, the topics covered in this chapter are:

- The characteristics and uses of generators
- Generator comprehensions
- Generator functions
- Generator classes
- Bundled generators
- Coroutines

What are generators?

A generator, in its simplest form, is a function that returns elements one at a time instead of returning a collection of items. The most important advantage of this is that it requires very little memory and that it doesn't need to have a predefined size. Creating an endless generator (such as the `itertools.count` iterator discussed in *Chapter 4*, *Functional Programming – Readability Versus Brevity*) is actually quite easy, but it does come with a cost, of course. Not having the size of an object available makes certain patterns difficult to achieve.

The basic trick in writing generators (as functions) is using the `yield` statement. Let's use the `itertools.count` generator as an example and extend it with a `stop` variable:

```
>>> def count(start=0, step=1, stop=10):
...     n = start
...     while n <= stop:
...         yield n
...         n += step


>>> for x in count(10, 2.5, 20):
...     print(x)
10
12.5
```

```
15.0
17.5
20.0
```

Due to the potentially infinite nature of generators, caution is required. Without the `stop` variable, simply doing `list(count())` would result in an out-of-memory situation quite fast.

So how does this work? It's just a normal `for` loop, but the big difference between this and the regular method of returning a list of items is that the `yield` statement returns the items one at a time. An important thing to note here is that the `return` statement results in a `StopIteration` and passing something along to `return` will be the argument to the `StopIteration`. It should be noted that this behavior changed in Python 3.3; in Python 3.2 and earlier versions, it was simply not possible to return anything other than `None`. Here is an example:

```
>>> def generator():
...     yield 'this is a generator'
...     return 'returning from a generator'


>>> g = generator()
>>> next(g)
'this is a generator'
>>> next(g)
Traceback (most recent call last):
    ...
StopIteration: returning from a generator
```

Of course, as always, there are multiple ways of creating generators with Python. Other than functions, there are also generator comprehensions and classes that can do the same thing. Generator comprehensions are pretty much identical to list comprehensions but use parentheses instead of brackets, like this for example:

```
>>> generator = (x ** 2 for x in range(4))


>>> for x in generator:
...    print(x)
0
1
4
9
```

For completeness, the class version of the `count` function is as follows:

```
>>> class Count(object):
...     def __init__(self, start=0, step=1, stop=10):
...         self.n = start
...         self.step = step
...         self.stop = stop
...
...     def __iter__(self):
...         return self
...
...     def __next__(self):
...         n = self.n
...         if n > self.stop:
...             raise StopIteration()
...
...         self.n += self.step
...         return n

>>> for x in Count(10, 2.5, 20):
...     print(x)
10
12.5
15.0
17.5
20.0
```

The biggest difference between the class and the function-based approach is that you are required to raise a `StopIteration` explicitly instead of just returning it. Beyond that, they are quite similar, although the class-based version obviously adds some verbosity.

Advantages and disadvantages of generators

You have seen a few examples of generators and know the basics of what you can do with them. However, it is important to keep their advantages and disadvantages in mind.

The following are most important pros:

- Memory usage. Items can be processed one at a time, so there is generally no need to keep the entire list in memory.
- The results can depend on outside factors, instead of having a static list. Think of processing a queue/stack for example.
- Generators are lazy. This means that if you're using only the first five results of a generator, the rest won't even be calculated.
- Generally, it is simpler to write than list generating functions.

The most important cons:

- The results are available only once. After processing the results of a generator, it cannot be used again.
- The size is unknown until you are done processing, which can be detrimental to certain algorithms.
- Generators are not indexable, which means that `some_generator[5]` will not work.

Considering all the advantages and disadvantages, my general advice would be to use generators if possible and only return a `list` or `tuple` when you actually need to. Converting a generator to a `list` is as simple as `list(some_generator)`, so that shouldn't stop you since generator functions tend to be simpler than the equivalents that produce `list`.

The memory usage advantage is understandable; one item requires less memory than many items. The lazy part, however, needs some additional explanation as it has a small snag:

```
>>> def generator():
...     print('Before 1')
...     yield 1
...     print('After 1')
...     print('Before 2')
...     yield 2
...     print('After 2')
```

```
...     print('Before 3')
...     yield 3
...     print('After 3')

>>> g = generator()
>>> print('Got %d' % next(g))
Before 1
Got 1

>>> print('Got %d' % next(g))
After 1
Before 2
Got 2
```

As you can see, the generator effectively freezes right after the `yield` statement, so even the `After 2` won't print until `3` is yielded.

This has important advantages, but it's definitely something you need to take into consideration. You can't have your cleanup right after the `yield` as it won't be executed until the next `yield`.

Pipelines – an effective use of generators

The theoretical possibilities of generators are infinite (no pun intended), but their practical uses can be difficult to find. If you are familiar with the Unix/Linux shell, you must have probably used pipes before, something like `ps aux | grep python'` for example to list all Python processes. There are many ways to do this, of course, but let's emulate something similar in Python to see a practical example. To create an easy and consistent output, we will create a file called `lines.txt` with the following lines:

```
spam
eggs
spam spam
eggs eggs
spam spam spam
eggs eggs eggs
```

Now, let's take the following Linux/Unix/Mac shell command to read the file with some modifications:

```
# cat lines.txt | grep spam | sed 's/spam/bacon/g'
bacon
bacon bacon
bacon bacon bacon
```

This reads the file using `cat`, outputs all lines that contain `spam` using `grep`, and replaces `spam` with `bacon` using the `sed` command. Now let's see how we can recreate this with the use of Python generators:

```
>>> def cat(filename):
...     for line in open(filename):
...         yield line.rstrip()
...
>>> def grep(sequence, search):
...     for line in sequence:
...         if search in line:
...             yield line
...
>>> def replace(sequence, search, replace):
...     for line in sequence:
...         yield line.replace(search, replace)
...
>>> lines = cat('lines.txt')
>>> spam_lines = grep(lines, 'spam')
>>> bacon_lines = replace(spam_lines, 'spam', 'bacon')

>>> for line in bacon_lines:
...     print(line)
...
bacon
bacon bacon
bacon bacon bacon
```

```
# Or the one-line version, fits within 78 characters:
>>> for line in replace(grep(cat('lines.txt'), 'spam'),
...                     'spam', 'bacon'):
...     print(line)
...
bacon
bacon bacon
bacon bacon bacon
```

That's the big advantage of generators. You can wrap a list or sequence multiple times with very little performance impact. Not a single one of the functions involved executes anything until a value is requested.

tee – using an output multiple times

As mentioned before, one of the biggest disadvantages of generators is that the results are usable only once. Luckily, Python has a function that allows you to copy the output to several generators. The name `tee` might be familiar to you if you are used to working in a command-line shell. The `tee` program allows you to write outputs to both the screen and a file, so you can store an output while still maintaining a live view of it.

The Python version, `itertools.tee`, does a similar thing except that it returns several iterators, allowing you to process the results separately.

By default, `tee` will split your generator into a tuple containing two different generators, which is why tuple unpacking works nicely here. By passing along the `n` parameter, this can easily be changed to support more than 2 generators. Here is an example:

```
>>> import itertools


>>> def spam_and_eggs():
...     yield 'spam'
...     yield 'eggs'


>>> a, b = itertools.tee(spam_and_eggs())
>>> next(a)
'spam'
>>> next(a)
'eggs'
```

```
>>> next(b)
'spam'
>>> next(b)
'eggs'
>>> next(b)
Traceback (most recent call last):
    ...
StopIteration
```

After seeing this code, you might be wondering about the memory usage of `tee`. Does it need to store the entire list for you? Luckily, no. The `tee` function is pretty smart in handling this. Assume you have a generator that contains 1,000 items, and you read the first 100 items from `a` and the first `75` items from `b` simultaneously. Then `tee` will only keep the difference (`100 - 75 = 25` items) in the memory and drop the rest while you are iterating the results.

Whether `tee` is the best solution in your case or not depends, of course. If instance `a` is read from the beginning to (nearly) the end before instance `b` is read, then it would not be a great idea to use `tee`. Simply converting the generator to a `list` would be faster since it involves much fewer operations.

Generating from generators

As we have seen before, we can use generators to filter, modify, add, and remove items. In many cases, however, you'll notice that when writing generators, you'll be returning from sub-generators and/or sequences. An example of this is when creating a `powerset` using the `itertools` library:

```
>>> import itertools


>>> def powerset(sequence):
...     for size in range(len(sequence) + 1):
...         for item in itertools.combinations(sequence, size):
...             yield item


>>> for result in powerset('abc'):
...     print(result)
()
('a',)
('b',)
```

```
('c',)
('a', 'b')
('a', 'c')
('b', 'c')
('a', 'b', 'c')
```

This pattern was so common that the yield syntax was actually enhanced to make this even easier. Instead of manually looping over the results, Python 3.3 introduced the `yield from` syntax, which makes this common pattern even simpler:

```
>>> import itertools

>>> def powerset(sequence):
...     for size in range(len(sequence) + 1):
...         yield from itertools.combinations(sequence, size)

>>> for result in powerset('abc'):
...     print(result)
()
('a',)
('b',)
('c',)
('a', 'b')
('a', 'c')
('b', 'c')
('a', 'b', 'c')
```

And that's how you create a powerset in only three lines of code.

Perhaps, a more useful example of this is flattening a sequence recursively:

```
>>> def flatten(sequence):
...     for item in sequence:
...         try:
...             yield from flatten(item)
...         except TypeError:
...             yield item
...
>>> list(flatten([1, [2, [3, [4, 5], 6], 7], 8]))
[1, 2, 3, 4, 5, 6, 7, 8]
```

Note that this code uses `TypeError` to detect non-iterable objects. The result is that if the sequence (which could be a generator) returns a `TypeError`, it will silently hide it.

Also note that this is a very basic flattening function that has no type checking whatsoever. An iterable containing an `str` for example will be flattened recursively until the maximum recursion depth is reached, since every item in an `str` also returns an `str`.

Context managers

As with most of the techniques described in this book, Python also comes bundled with a few useful generators. Some of these (`itertools` and `contextlib.contextmanager` for example) have already been discussed in *Chapter 4*, *Functional Programming – Readability Versus Brevity* and *Chapter 5*, *Decorators – Enabling Code Reuse by Decorating* but we can use some extra examples to demonstrate how simple and powerful they can be.

The Python context managers do not appear to be directly related to generators, but that's a large part of what they use internally:

```
>>> import datetime
>>> import contextlib


# Context manager that shows how long a context was active
>>> @contextlib.contextmanager
... def timer(name):
...     start_time = datetime.datetime.now()
...     yield
...     stop_time = datetime.datetime.now()
...     print('%s took %s' % (name, stop_time - start_time))


# The write to log function writes all stdout (regular print data) to
# a file. The contextlib.redirect_stdout context wrapper
# temporarily redirects standard output to a given file handle, in
# this case the file we just opened for writing.
>>> @contextlib.contextmanager
... def write_to_log(name):
...     with open('%s.txt' % name, 'w') as fh:
...         with contextlib.redirect_stdout(fh):
```

```
...             with timer(name):
...                 yield

# Use the context manager as a decorator
>>> @write_to_log('some function')
... def some_function():
...     print('This function takes a bit of time to execute')
...     ...
...     print('Do more...')

>>> some_function()
```

While all this works just fine, the three levels of context managers tend to get a bit unreadable. Generally, decorators can solve this. In this case, however, we need the output from one context manager as the input for the next.

That's where `ExitStack` comes in. It allows easy combining of multiple context managers:

```
>>> import contextlib



>>> @contextlib.contextmanager
... def write_to_log(name):
...     with contextlib.ExitStack() as stack:
...         fh = stack.enter_context(open('stdout.txt', 'w'))
...         stack.enter_context(contextlib.redirect_stdout(fh))
...         stack.enter_context(timer(name))
...
...         yield

>>> @write_to_log('some function')
... def some_function():
...     print('This function takes a bit of time to execute')
...     ...
...     print('Do more...')

>>> some_function()
```

Looks at least a bit simpler, doesn't it? While the necessity is limited in this case, the convenience of `ExitStack` becomes quickly apparent when you need to do specific teardowns. In addition to the automatic handling as seen before, it's also possible to transfer the contexts to a new `ExitStack` and manually handle the closing:

```
>>> import contextlib


>>> with contextlib.ExitStack() as stack:
...     spam_fh = stack.enter_context(open('spam.txt', 'w'))
...     eggs_fh = stack.enter_context(open('eggs.txt', 'w'))
...     spam_bytes_written = spam_fh.write('writing to spam')
...     eggs_bytes_written = eggs_fh.write('writing to eggs')
...     # Move the contexts to a new ExitStack and store the
...     # close method
...     close_handlers = stack.pop_all().close

>>> spam_bytes_written = spam_fh.write('still writing to spam')
>>> eggs_bytes_written = eggs_fh.write('still writing to eggs')

# After closing we can't write anymore
>>> close_handlers()
>>> spam_bytes_written = spam_fh.write('cant write anymore')
Traceback (most recent call last):
    ...
ValueError: I/O operation on closed file.
```

Most of the `contextlib` functions have extensive documentation available in the Python manual. `ExitStack` in particular is documented using many examples at `https://docs.python.org/3/library/contextlib.html#contextlib.ExitStack`. I recommend keeping an eye on the `contextlib` documentation as it is improving greatly with every Python version.

Coroutines

Coroutines are subroutines that offer non-pre-emptive multitasking through multiple entry points. The basic premise is that coroutines allow two functions to communicate with each other while running. Normally, this type of communication is reserved only for multitasking solutions, but coroutines offer a relatively simple way of achieving this at almost no added performance cost.

Since generators are lazy by default, the working of coroutines is fairly obvious. Until a result is consumed, the generator sleeps; but while consuming a result, the generator becomes active. The difference between regular generators and coroutines is that coroutines don't simply return values to the calling function but can receive values as well.

A basic example

In the previous paragraphs, we saw how regular generators can yield values. But that's not all that generators can do. They can actually receive values as well. The basic usage is fairly simple:

```
>>> def generator():
...     value = yield 'spam'
...     print('Generator received: %s' % value)
...     yield 'Previous value: %r' % value


>>> g = generator()
>>> print('Result from generator: %s' % next(g))
Result from generator: spam
>>> print(g.send('eggs'))
Generator received: eggs
Previous value: 'eggs'
```

And that's all there is to it. The function is frozen until the `send` method is called, at which point it will process up to the next `yield` statement.

Priming

Since generators are lazy, you can't just send a value to a brand new generator. Before a value can be sent to the generator, either a result must be fetched using `next()` or a `send(None)` has to be issued so that the code is actually reached. The need for this is understandable but a bit tedious at times. Let's create a simple decorator to omit the need for this:

```
>>> import functools


>>> def coroutine(function):
...     @functools.wraps(function)
...     def _coroutine(*args, **kwargs):
...         active_coroutine = function(*args, **kwargs)
...         next(active_coroutine)
...         return active_coroutine
...     return _coroutine


>>> @coroutine
... def spam():
...     while True:
...         print('Waiting for yield...')
...         value = yield
...         print('spam received: %s' % value)

>>> generator = spam()
Waiting for yield...

>>> generator.send('a')
spam received: a
Waiting for yield...

>>> generator.send('b')
spam received: b
Waiting for yield...
```

As you've probably noticed, even though the generator is still lazy, it now automatically executes all of the code until it reaches the `yield` statement again. At that point, it will stay dormant until new values are sent.

Note that the `coroutine` decorator will be used throughout this chapter from this point onwards. For brevity, we will omit it from the following examples.

Closing and throwing exceptions

Unlike regular generators, which simply exit as soon as the input sequence is exhausted, coroutines generally employ infinite `while` loops, which means that they won't be torn down the normal way. That's why coroutines also support both `close` and `throw` methods, which will exit the function. The important thing here is not the closing but the possibility of adding a teardown method. Essentially, it is very comparable to how context wrappers function with an `__enter__` and `__exit__` method, but with coroutines in this case:

```
@coroutine
def simple_coroutine():
    print('Setting up the coroutine')
    try:
        while True:
            item = yield
            print('Got item: %r' % item)
    except GeneratorExit:
        print('Normal exit')
    except Exception as e:
        print('Exception exit: %r' % e)
        raise
    finally:
        print('Any exit')


print('Creating simple coroutine')
active_coroutine = simple_coroutine()
print()

print('Sending spam')
active_coroutine.send('spam')
print()
```

```
    print('Close the coroutine')
    active_coroutine.close()
    print()


    print('Creating simple coroutine')
    active_coroutine = simple_coroutine()
    print()

    print('Sending eggs')
    active_coroutine.send('eggs')
    print()

    print('Throwing runtime error')
    active_coroutine.throw(RuntimeError, 'Oops...')
    print()
```

This generates the following output, which should be as expected—no strange behavior but simply two methods of exiting a coroutine:

```
# python3 H06.py
Creating simple coroutine
Setting up the coroutine

Sending spam
Got item: 'spam'

Close the coroutine
Normal exit
Any exit

Creating simple coroutine
Setting up the coroutine

Sending eggs
Got item: 'eggs'

Throwing runtime error
Exception exit: RuntimeError('Oops...',)
Any exit
```

```
Traceback (most recent call last):
...
  File ... in <module>
    active_coroutine.throw(RuntimeError, 'Oops...')
  File ... in simple_coroutine
    item = yield
RuntimeError: Oops...
```

Bidirectional pipelines

In the previous paragraphs, we saw pipelines; they process the output sequentially and one-way. However, there are cases where this is simply not enough—times where you need a pipe that not only sends values to the next pipe but also receives information back from the sub-pipe. Instead of always having a single list that is processed, we can maintain the state of the generator between executions this way. So, let's start by converting the earlier pipelines to coroutines. First, the `lines.txt` file again:

```
spam
eggs
spam spam
eggs eggs
spam spam spam
eggs eggs eggs
```

Now, the coroutine pipeline. The functions are the same as before but using coroutines instead:

```
>>> @coroutine
... def replace(search, replace):
...     while True:
...         item = yield
...         print(item.replace(search, replace))


>>> spam_replace = replace('spam', 'bacon')
>>> for line in open('lines.txt'):
...     spam_replace.send(line.rstrip())
bacon
eggs
```

```
bacon bacon
eggs eggs
bacon bacon bacon
eggs eggs eggs
```

Given this example, you might be wondering why we are now printing the value instead of yielding it. Well! We can, but remember that generators freeze until a value is yielded. Let's see what would happen if we simply `yield` the value instead of calling `print`. By default, you might be tempted to do this:

```
>>> @coroutine
... def replace(search, replace):
...     while True:
...         item = yield
...         yield item.replace(search, replace)


>>> spam_replace = replace('spam', 'bacon')
>>> spam_replace.send('spam')
'bacon'
>>> spam_replace.send('spam spam')
>>> spam_replace.send('spam spam spam')
'bacon bacon bacon'
```

Half of the values have disappeared now, so the question is, "Where did they go?" Notice that the second `yield` isn't storing the results. That's where the values are disappearing. We need to store those as well:

```
>>> @coroutine
... def replace(search, replace):
...     item = yield
...     while True:
...         item = yield item.replace(search, replace)


>>> spam_replace = replace('spam', 'bacon')
>>> spam_replace.send('spam')
```

```
'bacon'
>>> spam_replace.send('spam spam')
'bacon bacon'
>>> spam_replace.send('spam spam spam')
'bacon bacon bacon'
```

But even this is far from optimal. We are essentially using coroutines to mimic the behavior of generators right now. Although it works, it's just a tad silly and not all that clear. Let's make a real pipeline this time where the coroutines send the data to the next coroutine (or coroutines) and actually show the power of coroutines by sending the results to multiple coroutines:

```
# Grep sends all matching items to the target
>>> @coroutine
... def grep(target, pattern):
...     while True:
...         item = yield
...         if pattern in item:
...             target.send(item)

# Replace does a search and replace on the items and sends it to
# the target once it's done
>>> @coroutine
... def replace(target, search, replace):
...     while True:
...         target.send((yield).replace(search, replace))

# Print will print the items using the provided formatstring
>>> @coroutine
... def print_(formatstring):
...     while True:
...         print(formatstring % (yield))

# Tee multiplexes the items to multiple targets
>>> @coroutine
... def tee(*targets):
...     while True:
```

```
...         item = yield
...         for target in targets:
...             target.send(item)


# Because we wrap the results we need to work backwards from the
# inner layer to the outer layer.

# First, create a printer for the items:
>>> printer = print_('%s')

# Create replacers that send the output to the printer
>>> replacer_spam = replace(printer, 'spam', 'bacon')
>>> replacer_eggs = replace(printer, 'spam spam', 'sausage')

# Create a tee to send the input to both the spam and the eggs
# replacers
>>> branch = tee(replacer_spam, replacer_eggs)

# Send all items containing spam to the tee command
>>> grepper = grep(branch, 'spam')

# Send the data to the grepper for all the processing
>>> for line in open('lines.txt'):
...     grepper.send(line.rstrip())
bacon
spam
bacon bacon
sausage
bacon bacon bacon
sausage spam
```

This makes the code much simpler and more readable, but more importantly, it shows how a single source can be split into multiple destinations. While this might not look too exciting, it most certainly is. If you look closely, you will see that the `tee` method splits the input into two different outputs, but both of those outputs write back to the same `print_` instance. This means that it's possible to route your data along whichever way is convenient for you while still having it end up at the same endpoint with no effort whatsoever.

Regardless, the example is still not that useful, as these functions still don't use all of the coroutine's power. The most important feature, a consistent state, is not really used in this case.

The most important lesson to learn from these lines is that mixing generators and coroutines is not a good idea in most cases since it can have very strange side effects if used incorrectly. Even though both use the `yield` statement, they are significantly different creatures with different behavior. The next paragraph will show one of the few cases where mixing coroutines and generators can be useful.

Using the state

Now that we know how to write basic coroutines and which pitfalls we have to take care of, how about writing a function where remembering the state is required? That is, a function that always gives you the average value of all sent values. This is one of the few cases where it is still relatively safe and useful to combine the coroutine and generator syntax:

```
>>> @coroutine
... def average():
...     count = 1
...     total = yield
...     while True:
...         total += yield total / count
...         count += 1

>>> averager = average()
>>> averager.send(20)
20.0
>>> averager.send(10)
15.0
>>> averager.send(15)
15.0
>>> averager.send(-25)
5.0
```

It still requires some extra logic to work properly though. To make sure we don't divide by zero, we initialize the `count` to `1`. After that, we fetch our first item using `yield`, but we don't send any data at that point because the first `yield` is the primer and is executed before we get the value. Once that's all set up, we can easily yield the average value while summing. Not all that bad, but the pure coroutine version is slightly simpler to understand since we don't have to worry about priming:

```
>>> @coroutine
... def print_(formatstring):
...     while True:
...         print(formatstring % (yield))



>>> @coroutine
... def average(target):
...     count = 0
...     total = 0
...     while True:
...         count += 1
...         total += yield
...         target.send(total / count)

>>> printer = print_('%.1f')
>>> averager = average(printer)
>>> averager.send(20)
20.0
>>> averager.send(10)
15.0
>>> averager.send(15)
15.0
>>> averager.send(-25)
5.0
```

As simple as it should be, just keeping the count and the total value and simply send the new average for every new value.

Another nice example is `itertools.groupby`, also quite simple to do with coroutines. For comparison, we will once again show both the generator coroutine and the pure coroutine version:

```
>>> @coroutine
... def groupby():
...     # Fetch the first key and value and initialize the state
...     # variables
...     key, value = yield
...     old_key, values = key, []
...     while True:
...         # Store the previous value so we can store it in the
...         # list
...         old_value = value
...         if key == old_key:
...             key, value = yield
...         else:
...             key, value = yield old_key, values
...             old_key, values = key, []
...         values.append(old_value)


>>> grouper = groupby()
>>> grouper.send(('a', 1))
>>> grouper.send(('a', 2))
>>> grouper.send(('a', 3))
>>> grouper.send(('b', 1))
('a', [1, 2, 3])
>>> grouper.send(('b', 2))
>>> grouper.send(('a', 1))
('b', [1, 2])
>>> grouper.send(('a', 2))
>>> grouper.send((None, None))
('a', [1, 2])
```

As you can see, this function uses a few tricks. We store the previous `key` and `value` so that we can detect when the group (`key`) changes. And that is the second issue; we obviously cannot recognize a group until the group has changed, so only after the group has changed will the results be returned. This means that the last group will be sent only if a different group is sent after it, hence the `(None, None)`. And now, here is the pure coroutine version:

```
>>> @coroutine
... def print_(formatstring):
...     while True:
...         print(formatstring % (yield))


>>> @coroutine
... def groupby(target):
...     old_key = None
...     while True:
...         key, value = yield
...         if old_key != key:
...             # A different key means a new group so send the
...             # previous group and restart the cycle.
...             if old_key and values:
...                 target.send((old_key, values))
...             values = []
...             old_key = key
...         values.append(value)


>>> grouper = groupby(print_('group: %s, values: %s'))
>>> grouper.send(('a', 1))
>>> grouper.send(('a', 2))
>>> grouper.send(('a', 3))
>>> grouper.send(('b', 1))
group: a, values: [1, 2, 3]
>>> grouper.send(('b', 2))
>>> grouper.send(('a', 1))
group: b, values: [1, 2]
>>> grouper.send(('a', 2))
>>> grouper.send((None, None))
group: a, values: [1, 2]
```

While the functions are fairly similar, the pure coroutine version is, once again, quite a bit simpler. This is because we don't have to think about priming and values that might get lost.

Summary

This chapter showed us how to create generators and both the strengths and weaknesses that they possess. Additionally, it should now be clear how to work around their limitations and the implications of doing so.

While the paragraphs about coroutines should have provided some insights into what they are and how they can be used, not everything has been shown yet. We saw the constructs of both pure coroutines and coroutines that are generators at the same time, but they are still all synchronous. The coroutines allow sending the results to many other coroutines, therefore effectively executing many functions at once, but they can still freeze Python completely if an operation turns out to be blocking. That's where our next chapter will help.

Python 3.5 introduced a few useful features, such as the `async` and `await` statements. These make it possible to make coroutines fully asynchronous and non-blocking, whereas this chapter uses the basic coroutine features that have been available since Python 2.5.

The next chapter will expand on the newer features, including the `asyncio` module. This module makes it almost simple to use coroutines for asynchronous I/O to endpoints such as TCP, UDP, files, and processes.

7
Async IO – Multithreading without Threads

The previous chapter showed us the basic implementation of synchronous coroutines. Whenever you are dealing with external resources, however, synchronous coroutines are a bad idea. Just a single stalling remote connection can cause your entire process to hang, unless you are using multiprocessing (explained in *Chapter 13*, *Multiprocessing – When a Single CPU Core Is Not Enough*) or asynchronous functions that is.

Asynchronous IO makes it possible to access external resources without having to worry about slowing down or stalling your application. Instead of actively waiting for results, the Python interpreter can simply continue with other tasks until it is needed again. This is very similar to the functioning of Node.js and AJAX calls in JavaScript. Within Python, we have seen libraries such as `asyncore`, `gevent`, and `eventlet` that have made this possible for years. With the introduction of the `asyncio` module, however, it has become significantly easier to use.

This chapter will explain how asynchronous functions can be used in Python (particularly 3.5 and above) and how code can be restructured in such a way that it still functions even though it doesn't follow the standard procedural coding pattern of returning values.

The following topics will be covered in this chapter:

- Functions using:
  - `async def`
  - `async for`
  - `async with`
  - `await`
- Parallel execution
- Servers
- Clients
- Eventual results using `Future`

Introducing the asyncio library

The `asyncio` library was created to make asynchronous processing much easier and results more predictable. It was introduced with the purpose of replacing the `asyncore` module, which has been available for a very long time (since Python 1.5 in fact). The `asyncore` module was never very usable, which prompted the creation of the `gevent` and `eventlet` third-party libraries. Both `gevent` and `eventlet` make asynchronous programming much easier than `asyncore` ever did, but I feel that both have been made largely obsolete with the introduction of `asyncio`. Even though I have to admit that `asyncio` still has quite a few rough edges, it is in very active development, which makes me think that all the rough edges will soon be fixed by either the core Python library or third-party wrappers.

The `asyncio` library was officially introduced for Python 3.4, but a back port for Python 3.3 is available through the Python Package Index. With that in mind, while some portions of this chapter will be able to run on Python 3.3, most of it has been written with Python 3.5 and the newly introduced `async` and `await` keywords in mind.

The async and await statements

Before we continue with any example, it is important to know how the Python 3.4 and Python 3.5 code syntaxes relate. Even though the `asyncio` library was introduced only in Python 3.4, a large portion of the generic syntax has already been replaced in Python 3.5. Not forcefully, but the easier and therefore recommended syntax using `async` and `await` has been introduced.

Python 3.4

For the traditional Python 3.4 usage, a few things need to be considered:

- Functions should be declared using the `asyncio.coroutine` decorator
- Asynchronous results should be fetched using `yield from coroutine()`
- Asynchronous loops are not directly supported but can be emulated using `while True: yield from coroutine()`

Here is an example:

```
import asyncio


@asyncio.coroutine
def sleeper():
    yield from asyncio.sleep(1)
```

Python 3.5

In Python 3.5, a new syntax was introduced to mark a function as asynchronous. Instead of the `asyncio.coroutine` decorator, the `async` keyword can be used. Also, instead of the confusing `yield from` syntax, Python now supports the `await` statement. The `yield from` statement was slightly confusing because it might give someone the idea that a value is being exchanged, which is not always the case.

The following is the `async` statement:

```
async def some_coroutine():
    pass
```

It can be used instead of the decorator:

```
import asyncio


@asyncio.coroutine
def some_coroutine():
    pass
```

Within Python 3.5, and most likely in future versions as well, the `coroutine` decorator will still be supported, but if backwards compatibility is not an issue, I strongly recommend the new syntax.

Additionally, instead of the `yield from` statement, we can use the much more logical `await` statement. So, the example from the previous paragraph becomes as simple as the following:

```
import asyncio


async def sleeper():
    await asyncio.sleep(1)
```

The `yield from` statement originated from the original coroutines implementation in Python and was a logical extension from the `yield` statement used within synchronous coroutines. Actually, the `yield from` statement still works and the `await` statement is just a wrapper for it, with some added checks. While using `await`, the interpreter checks whether the object is an awaitable object, meaning it needs to be one of the following:

- A native coroutine created with the `async def` statement
- A coroutine created with the `asyncio.coroutine` decorator
- An object that implements the `__await__` method

This check alone makes the `await` statement preferable over the `yield from` statement, but I personally think that `await` conveys the meaning of the statement much better as well.

To summarize, to convert to the new syntax, make the following changes:

- Functions should be declared using `async def` instead of `def`
- Asynchronous results should be fetched using `await coroutine()`
- Asynchronous loops can be created using `async for ... in ...`
- Asynchronous `with` statements can be created using `async with ...`

Choosing between the 3.4 and 3.5 syntax

Unless you really need Python 3.3 or 3.4 support, I would strongly recommend the Python 3.5 syntax. The new syntax is clearer and supports more features, such as asynchronous `for` loops and `with` statements. Unfortunately, they are not fully compatible, so you need to make a choice. Within an `async def` (3.5), we cannot use `yield from`, but all we need to do to fix that is replace `yield from` with `await`.

A simple example of single-threaded parallel processing

Parallel processing has many uses: a server taking care of multiple requests at the same time, speeding up heavy tasks, waiting for external resources, and much more. Generic coroutines can help with handling multiple requests and external resources in some cases, but they are still synchronous and therefore limited. With `asyncio`, we can transcend the limitations of generic coroutines and easily handle stalling resources without having to worry about blocking the main thread. Let's see a quick example of how the code does not stall with multiple parallel functions:

```
>>> import asyncio



>>> async def sleeper(delay):
...     await asyncio.sleep(delay)
...     print('Finished sleeper with delay: %d' % delay)



>>> loop = asyncio.get_event_loop()
>>> results = loop.run_until_complete(asyncio.wait((
...     sleeper(1),
...     sleeper(3),
...     sleeper(2),
... )))
Finished sleeper with delay: 1
Finished sleeper with delay: 2
Finished sleeper with delay: 3
```

Even though we started the sleepers with the order of 1, 3, 2, which sleeps for that amount of time, `asyncio.sleep` combined with the `await` statement actually tells Python that it should just continue with a task that needs actual processing at this time. A regular `time.sleep` would actually stall the Python task, meaning they would execute sequentially. This makes it somewhat more obviously transparent what these can be used for, as it handles any type of wait, which we can hand off to `asyncio` instead of keeping the entire Python thread busy. So, instead of `while True: fh.read()`, we can just respond whenever there is new data.

Let's analyze the components used in this example:

- `asyncio.coroutine`: This decorator enables yielding from `async def` coroutines. Unless you are using this syntax, there is no real need for the decorator, but it's a good default if only used as documentation.
- `asyncio.sleep`: This is the asynchronous version of `time.sleep`. The big difference between these two is that `time.sleep` will keep the Python process busy while it is sleeping, whereas `asyncio.sleep` will allow switching to a different task within the event loop. This process is very similar to the workings of task switching in most operating systems.
- `asyncio.get_event_loop`: The default event loop is effectively the `asyncio` task switcher; we'll explain more about these in the next paragraph.
- `asyncio.wait`: This is the coroutine for wrapping a sequence of coroutines or futures and waiting for the results. The wait time is configurable, as is the manner of waiting (first done, all done, or the first exception).

That should explain the basic workings of the example: the `sleeper` function is the asynchronous coroutine, which exits after the given delay. The `wait` function waits for all coroutines to finish before exiting, and the `event` loop is used for switching between the three coroutines.

Concepts of asyncio

The `asyncio` library has several basic concepts, which have to be explained before we venture further into examples and uses. The example shown in the previous paragraph actually used most of them, but a little explanation about the how and the why might still be useful.

The main concepts of `asyncio` are *coroutines* and *event loops*. Within them, there are several helper classes available, such as `Streams`, `Futures`, and `Processes`. The next few paragraphs will explain the basics so that you can understand the implementations in the examples in the later paragraphs.

Futures and tasks

The `asyncio.Future` class is essentially a promise of a result; it returns the results if they are available, and once it receives results, it will pass them along to all the registered callbacks. It maintains a state variable internally, which allows an outside party to mark a future as canceled. The API is very similar to the `concurrent.futures.Future` class, but since they are not fully compatible, make sure you do not confuse the two.

The `Future` class by itself is not that convenient to use though, so that is where `asyncio.Task` comes in. The `Task` class wraps a coroutine and automatically handles the execution, results, and state for you. The coroutine will be executed through the given event loop, or the default event loop if none was given.

The creation of these classes is not something you need to worry about directly. This is because instead of creating the class yourself, the recommended way is through either `asyncio.ensure_future` or `loop.create_task`. The former actually executes `loop.create_task` internally but it is more convenient if you simply want to execute it on the main/default event loop without having to specify it first. The usage is simple enough. To create your own future manually, you simply tell the event loop to execute `create_task` for you. The following example is a bit complicated because of all the setup code but the usage of C should be clear enough. The most important aspect to note is that the event loop should be linked so that the task knows how/where to run:

```
>>> import asyncio



>>> async def sleeper(delay):
...     await asyncio.sleep(delay)
...     print('Finished sleeper with delay: %d' % delay)

# Create an event loop
>>> loop = asyncio.get_event_loop()

# Create the task
>>> result = loop.call_soon(loop.create_task, sleeper(1))

# Make sure the loop stops after 2 seconds
>>> result = loop.call_later(2, loop.stop)

# Start the loop and make it run forever. Or at least until the loop.stop
gets
# called in 2 seconds.
>>> loop.run_forever()
Finished sleeper with delay: 1
```

Now, a little bit about debugging asynchronous functions. Debugging asynchronous functions used to be very difficult if not impossible, as there was no good way to see where and how the functions were stalling. Luckily, that has changed. In the case of the `Task` class, it is as simple as calling `task.get_stack` or `task.print_stack` to see where it is currently. The usage can be as simple as the following:

```
>>> import asyncio


>>> async def stack_printer():
...     for task in asyncio.Task.all_tasks():
...         task.print_stack()

# Create an event loop
>>> loop = asyncio.get_event_loop()

# Create the task
>>> result = loop.run_until_complete(stack_printer())
```

Event loops

The concept of event loops is actually the most important one within `asyncio`. You might have suspected that the coroutines themselves are what everything is about, but without the event loop, they are useless. Event loops function as task switchers, just the way operating systems switch between active tasks on the CPU. Even with multicore processors, there is still a need for a main process to tell the CPU which tasks have to run and which need to wait/sleep for a while. This is exactly what the event loop does: it decides which task to run.

Event loop implementations

So far, we have only seen `asyncio.get_event_loop`, which returns the default event loop with the default event loop policy. Currently, there are two bundled event loop implementations: the `async.SelectorEventLoop` and `async.ProactorEventLoop` implementations. Which of the two is available depends on your operating system. The latter event loop is available only on Windows machines and uses I/O Completion Ports, which is a system that is supposedly faster and more efficient than the `Select` implementation of `asyncio.SelectorEventLoop`. This is something to consider if performance is an issue. The usage is simple enough, luckily:

```
import asyncio


loop = asyncio.ProActorEventLoop()
asyncio.set_event_loop(loop)
```

The alternative event loop is based on selectors, which, since Python 3.4, are available through the `selectors` module in the core Python installation. The `selectors` module was introduced in Python 3.4 to enable easy access to low-level asynchronous I/O operations. Basically, it allows you to open and read from many files by using I/O multiplexing. Since `asyncio` handles all complexities for you, there is generally no need to use the module directly, but the usage is simple enough if you need it. Here's an example of binding a function to the read event (`EVENT_READ`) on the standard input. The code will simply wait until one of the registered files provides new data:

```
import sys
import selectors


def read(fh):
    print('Got input from stdin: %r' % fh.readline())

if __name__ == '__main__':
    # Create the default selector
    selector = selectors.DefaultSelector()

    # Register the read function for the READ event on stdin
    selector.register(sys.stdin, selectors.EVENT_READ, read)

    while True:
        for key, mask in selector.select():
            # The data attribute contains the read function here
            callback = key.data
            # Call it with the fileobj (stdin here)
            callback(key.fileobj)
```

There are several selectors available, such as the traditional `selectors.SelectSelector` (which uses `select.select` internally), but there are also more modern solutions such as `selectors.KqueueSelector`, `selectors.EpollSelector`, and `selectors.DevpollSelector`. Even though it should select the most efficient selector by default, there are cases where the most efficient one is not suitable in some way or another. In those cases, the selector event loop allows you to specify a different selector:

```
import asyncio
import selectors


selector = selectors.SelectSelector()
loop = asyncio.SelectorEventLoop(selector)
asyncio.set_event_loop(loop)
```

It should be noted that the differences between these selectors are generally too small to notice in most real-world applications. The only situation I have come across where such an optimization makes a difference is when building a server that has to handle a lot of simultaneous connections. With "a lot," I am referring to over 100,000 concurrent connections on a single server, which is a problem only a few people on this planet have had to deal with.

Event loop policies

Event loop policies are objects that create and store the actual event loops for you. They have been written with maximum flexibility in mind but are not objects that you often need to modify. The only reason I can think of modifying the event loop policy is if you want to make specific event loops run on specific processors and/or systems, or if you wish to change the default event loop type. Beyond that, it offers more flexibility than most people will ever need. Making your own event loop (`ProActorEventLoop` in this case) the default is simply possible through this code:

```
import asyncio


class ProActorEventLoopPolicy(
        asyncio.events.BaseDefaultEventLoopPolicy):
    _loop_factory = asyncio.SelectorEventLoop


policy = ProActorEventLoopPolicy()
asyncio.set_event_loop_policy(policy)
```

Event loop usage

So far, we have only seen the `loop.run_until_complete` method. Naturally, there are a few others as well. The one you will most likely use most often is `loop.run_forever`. This method, as you might expect, keeps running forever, or at least until `loop.stop` has been run.

So, assuming we have an event loop running forever now, we need to add tasks to it. This is where things get interesting. There are quite a few choices available within the default event loops:

- `call_soon`: Add an item to the end of the (FIFO) queue so that the functions will be executed in the order in which they were inserted.
- `call_soon_threadsafe`: This is the same as `call_soon` except for being thread safe. The `call_soon` method is not thread safe because thread safety requires the usage of the global interpreter lock (GIL), which effectively makes your program single threaded at the moment of thread safety. The performance chapter will explain this more thoroughly.

- `call_later`: Call the function after the given number of seconds. If two jobs would run at the same time, they will run in an undefined order. Note that the delay is a minimum. If the event loop is locked/busy, it can run later.
- `call_at`: Call a function at a specific time related to the output of `loop.time`. Every integer after `loop.time` adds a second.

All of these functions return `asyncio.Handle` objects. These objects allow the cancellation of the task through the `handle.cancel` function as long as it has not been executed yet. Be careful with canceling from other threads, however, as cancellation is not thread safe either. To execute it in a thread-safe way, we have to execute the cancellation function as a task as well: `loop.call_soon_threadsafe(handle.cancel)`. The following is an example usage:

```
>>> import time
>>> import asyncio



>>> t = time.time()

>>> def printer(name):
...     print('Started %s at %.1f' % (name, time.time() - t))
...     time.sleep(0.2)
...     print('Finished %s at %.1f' % (name, time.time() - t))



>>> loop = asyncio.get_event_loop()
>>> result = loop.call_at(loop.time() + .2, printer, 'call_at')
>>> result = loop.call_later(.1, printer, 'call_later')
>>> result = loop.call_soon(printer, 'call_soon')
>>> result = loop.call_soon_threadsafe(printer, 'call_soon_threadsafe')

>>> # Make sure we stop after a second
>>> result = loop.call_later(1, loop.stop)

>>> loop.run_forever()
Started call_soon at 0.0
Finished call_soon at 0.2
Started call_soon_threadsafe at 0.2
```

```
Finished call_soon_threadsafe at 0.4
Started call_later at 0.4
Finished call_later at 0.6
Started call_at at 0.6
Finished call_at at 0.8
```

You might be wondering why we are not using the coroutine decorator here. The reason is that the loop won't allow running of coroutines directly. To run a coroutine through these call functions, we need to make sure that it is wrapped in an `asyncio.Task`. As we have seen in the previous paragraph, this is easy enough—luckily:

```
>>> import time
>>> import asyncio


>>> t = time.time()

>>> async def printer(name):
...     print('Started %s at %.1f' % (name, time.time() - t))
...     await asyncio.sleep(0.2)
...     print('Finished %s at %.1f' % (name, time.time() - t))


>>> loop = asyncio.get_event_loop()

>>> result = loop.call_at(
...     loop.time() + .2, loop.create_task, printer('call_at'))
>>> result = loop.call_later(.1, loop.create_task,
...     printer('call_later'))
>>> result = loop.call_soon(loop.create_task,
...     printer('call_soon'))

>>> result = loop.call_soon_threadsafe(
...     loop.create_task, printer('call_soon_threadsafe'))

>>> # Make sure we stop after a second
>>> result = loop.call_later(1, loop.stop)
```

```
>>> loop.run_forever()
Started call_soon at 0.0
Started call_soon_threadsafe at 0.0
Started call_later at 0.1
Started call_at at 0.2
Finished call_soon at 0.2
Finished call_soon_threadsafe at 0.2
Finished call_later at 0.3
Finished call_at at 0.4
```

These call methods might appear slightly different but the internals actually boil down to two queues that are implemented through `heapq`. The `loop._scheduled` is used for scheduled operations and `loop._ready` is for immediate execution. When the `_run_once` method is called (the `run_forever` method wraps this method in a `while True` loop), the loop will first try to process all items in the `loop._ready` heap with the specific loop implementation (for example, `SelectorEventLoop`). Once everything in `loop._ready` is processed, the loop will continue to move items from the `loop._scheduled` heap to the `loop._ready` heap if they are due.

Both `call_soon` and `call_soon_threadsafe` write to the `loop._ready` heap. And the `call_later` method is simply a wrapper for `call_at` with the current value of `asyncio.time` added to the scheduled time, which writes to the `loop._scheduled` heap.

The result of this method of processing is that everything added through the `call_soon*` methods will always execute after everything that is added through the `call_at/call_later` methods.

As for the `ensure_futures` function, it will call `loop.create_task` internally to wrap the coroutine in a `Task` object, which is, of course, a subclass of a `Future` object. If you need to extend the `Task` class for some reason, that is easily possible through the `loop.set_task_factory` method.

Depending on the type of event loop, there are actually many other methods for creating connections, file handlers, and more. Those will be explained by example in later paragraphs, since they have less to do with the event loop and are more about programming with coroutines.

Processes

So far, we have simply executed specifically asynchronous Python functions, but some things are a tad more difficult to run asynchronously within Python. For example, let's assume we have a long-running external application that we wish to run. The `subprocess` module would be the standard approach for running external applications, and it works quite well. With a bit of care, one could even make sure that these do not block the main thread by polling the output. That still requires polling, however. Yet, won't events be better so that we can do other things while we are waiting for the results? Luckily, this is easily arranged through `asyncio.Process`. Similar to the `Future` and `Task` classes, this class is meant to be created through the event loop. In terms of usage, the class is very similar to the `subprocess.Popen` class, except that the functions have been made asynchronous. This results in the removal of the polling function, of course.

First, let's look at the traditional sequential version:

```
>>> import time
>>> import subprocess
>>>
>>>
>>> t = time.time()
>>>
>>>
>>> def process_sleeper():
...     print('Started sleep at %.1f' % (time.time() - t))
...     process = subprocess.Popen(['sleep', '0.1'])
...     process.wait()
...     print('Finished sleep at %.1f' % (time.time() - t))
...
>>>
>>> for i in range(3):
...     process_sleeper()
Started sleep at 0.0
Finished sleep at 0.1
Started sleep at 0.1
Finished sleep at 0.2
Started sleep at 0.2
Finished sleep at 0.3
```

Since everything is executed sequentially, it takes three times the 0.1 seconds that the sleep command is sleeping. So, instead of waiting for all of them at the same time, let's run them in parallel this time:

```
>>> import time
>>> import subprocess


>>> t = time.time()


>>> def process_sleeper():
...     print('Started sleep at %.1f' % (time.time() - t))
...     return subprocess.Popen(['sleep', '0.1'])
...
>>>
>>> processes = []
>>> for i in range(5):
...     processes.append(process_sleeper())
Started sleep at 0.0
Started sleep at 0.0
Started sleep at 0.0
Started sleep at 0.0
Started sleep at 0.0

>>> for process in processes:
...     returncode = process.wait()
...     print('Finished sleep at %.1f' % (time.time() - t))
Finished sleep at 0.1
Finished sleep at 0.1
Finished sleep at 0.1
Finished sleep at 0.1
Finished sleep at 0.1
```

While this looks a lot better in terms of runtime, our program structure is a bit messy now. We needed two loops, one to start the processes and one to measure the finish time. Moreover, we had to move the print statement outside of the function, which is generally not desirable either. This time, we will try the `asyncio` version:

```
>>> import time
>>> import asyncio


>>> t = time.time()


>>> async def async_process_sleeper():
...     print('Started sleep at %.1f' % (time.time() - t))
...     process = await asyncio.create_subprocess_exec('sleep', '0.1')
...     await process.wait()
...     print('Finished sleep at %.1f' % (time.time() - t))


>>> loop = asyncio.get_event_loop()
>>> for i in range(5):
...     task = loop.create_task(async_process_sleeper())

>>> future = loop.call_later(.5, loop.stop)

>>> loop.run_forever()
Started sleep at 0.0
Started sleep at 0.0
Started sleep at 0.0
Started sleep at 0.0
Started sleep at 0.0
Finished sleep at 0.1
Finished sleep at 0.1
Finished sleep at 0.1
Finished sleep at 0.1
Finished sleep at 0.1
```

As you can see, it is easy to run multiple applications at the same time this way. But that is the easy part; the difficult part with processes is interactive input and output. The `asyncio` module has several measures to make it easier, but it can still be difficult when actually working with the results. Here's an example of calling the Python interpreter, executing some code, and exiting again:

```
import asyncio


async def run_script():
    process = await asyncio.create_subprocess_shell(
        'python3',
        stdout=asyncio.subprocess.PIPE,
        stdin=asyncio.subprocess.PIPE,
    )

    # Write a simple Python script to the interpreter
    process.stdin.write(b'\n'.join((
        b'import math',
        b'x = 2 ** 8',
        b'y = math.sqrt(x)',
        b'z = math.sqrt(y)',
        b'print("x: %d" % x)',
        b'print("y: %d" % y)',
        b'print("z: %d" % z)',
        b'for i in range(int(z)):',
        b'    print("i: %d" % i)',
    )))
    # Make sure the stdin is flushed asynchronously
    await process.stdin.drain()
    # And send the end of file so the Python interpreter will
    # start processing the input. Without this the process will
    # stall forever.
    process.stdin.write_eof()

    # Fetch the lines from the stdout asynchronously
    async for out in process.stdout:
        # Decode the output from bytes and strip the whitespace
        # (newline) at the right
        print(out.decode('utf-8').rstrip())

    # Wait for the process to exit
```

```
        await process.wait()


    if __name__ == '__main__':
        loop = asyncio.get_event_loop()
        loop.run_until_complete(run_script())
        loop.close()
```

The code is simple enough, but there are a few parts of this code that are not obvious to us and yet required to function. While the creation of the subprocess and the writing code is quite obvious, you might be wondering about the `process.stdin.write_eof()` line. The problem here is buffering. To improve performance, most programs will buffer input and output by default. In the case of the Python program, the result is that unless we send the **end of file** (**eof**), the program will keep waiting for more input. An alternative solution would be to close the `stdin` stream or somehow communicate with the Python program that we will not send any more input. However, it is certainly something to take into consideration. Another option is to use `yield` from `process.stdin.drain()`, but that only takes care of the sending side of the code; the receiving side might still be waiting for more input. Let's see the output though:

```
# python3 processes.py
x: 256
y: 16
z: 4
i: 0
i: 1
i: 2
i: 3
```

With this implementation, we still need a loop to get all the results from the `stdout` stream. Unfortunately, the `asyncio.StreamReader` (which `process.stdout` is) class does not support the `async for` syntax yet. If it did, a simple `async for out in process.stdout` would have worked. A simple `yield from process.stdout.read()` would have worked as well, but reading per line is generally more convenient to use.

If possible, I recommend that you abstain from using `stdin` to send data to subprocesses and instead use some network, pipe, or file communication. As we will see in the next paragraphs, these are much more convenient to handle.

Asynchronous servers and clients

One of the most common reason for stalling scripts and applications is the usage of remote resources. With `asyncio`, at least a large portion of that is easily fixable. Fetching multiple remote resources and serving to multiple clients is quite a bit easier and more lightweight than it used to be. While both multithreading and multiprocessing can be used for these cases as well, `asyncio` is a much lighter alternative and it is actually easier to manage. There are two main methods of creating clients and servers. The coroutine way is to use `asyncio.open_connection` and `asyncio.start_server`. The class-based approach requires you to inherit the `asyncio.Protocol` class. While these are essentially the same thing, the workings are slightly different.

Basic echo server

The basic client and server versions are simple enough to write. The `asyncio` module takes care of all the low-level connection handling, leaving us with only the requirement of connecting the correct methods. For the server, we need a method to handle the incoming connections, and for the client, we need a function to create connections. And to illustrate what is happening and at which point in time, we will add a dedicated print function that prints both the time since the server process was started and the given arguments:

```
import time
import sys
import asyncio


HOST = '127.0.0.1'
PORT = 1234


start_time = time.time()


def printer(start_time, *args, **kwargs):
    '''Simple function to print a message prefixed with the
    time relative to the given start_time'''
    print('%.1f' % (time.time() - start_time), *args, **kwargs)


async def handle_connection(reader, writer):
    client_address = writer.get_extra_info('peername')
    printer(start_time, 'Client connected', client_address)

    # Send over the server start time to get consistent
    # timestamps
    writer.write(b'%.2f\n' % start_time)
    await writer.drain()
```

```
        repetitions = int((await reader.readline()))
        printer(start_time, 'Started sending to', client_address)

        for i in range(repetitions):
            message = 'client: %r, %d\n' % (client_address, i)
            printer(start_time, message, end='')
            writer.write(message.encode())
            await writer.drain()

        printer(start_time, 'Finished sending to', client_address)
        writer.close()


    async def create_connection(repetitions):
        reader, writer = await asyncio.open_connection(
            host=HOST, port=PORT)

        start_time = float((await reader.readline()))

        writer.write(repetitions.encode() + b'\n')
        await writer.drain()

        async for line in reader:
            # Sleeping a little to emulate processing time and make
            # it easier to add more simultaneous clients
            await asyncio.sleep(1)

            printer(start_time, 'Got line: ', line.decode(),
                    end='')

        writer.close()


    if __name__ == '__main__':
        loop = asyncio.get_event_loop()

        if sys.argv[1] == 'server':
            server = asyncio.start_server(
                handle_connection,
                host=HOST,
                port=PORT,
            )
            running_server = loop.run_until_complete(server)

            try:
                result = loop.call_later(5, loop.stop)
                loop.run_forever()
            except KeyboardInterrupt:
                pass
```

```
            running_server.close()
            loop.run_until_complete(running_server.wait_closed())
        elif sys.argv[1] == 'client':
            loop.run_until_complete(create_connection(sys.argv[2]))

        loop.close()
```

Now we will run the server and two simultaneous clients. Since these run in parallel, the server output is a bit strange, of course. Because of that, we synchronize the start time from the server to the clients and prefix all print statements with the number of seconds since the server was started.

The server:

```
# python3 simple_connections.py server
0.4 Client connected ('127.0.0.1', 59990)
0.4 Started sending to ('127.0.0.1', 59990)
0.4 client: ('127.0.0.1', 59990), 0
0.4 client: ('127.0.0.1', 59990), 1
0.4 client: ('127.0.0.1', 59990), 2
0.4 Finished sending to ('127.0.0.1', 59990)
2.0 Client connected ('127.0.0.1', 59991)
2.0 Started sending to ('127.0.0.1', 59991)
2.0 client: ('127.0.0.1', 59991), 0
2.0 client: ('127.0.0.1', 59991), 1
2.0 Finished sending to ('127.0.0.1', 59991)
```

The first client:

```
# python3 simple_connections.py client 3
1.4 Got line:  client: ('127.0.0.1', 59990), 0
2.4 Got line:  client: ('127.0.0.1', 59990), 1
3.4 Got line:  client: ('127.0.0.1', 59990), 2
```

The second client:

```
# python3 simple_connections.py client 2
3.0 Got line:  client: ('127.0.0.1', 59991), 0
4.0 Got line:  client: ('127.0.0.1', 59991), 1
```

Since both the input and output have buffers, we need to manually drain the input after writing and use `yield from` when reading the output from the other party. That is exactly the reason that communication with regular external processes is more difficult than network interaction. The standard input for processes is more focused towards user input than computer input, which makes it less convenient to use.

If you wish to use `reader.read(BUFFER)` instead of `reader.readline()`, that's also possible. Just note that you need to specifically separate the data because it might accidently get appended otherwise. All write operations write to the same buffer, resulting in one long return stream. On the other hand, trying to write without a new line (`\n`) for `reader.readline()` to recognize will cause the client to wait forever.

Summary

In this chapter, we saw how to use asynchronous I/O in Python using `asyncio`. For many scenarios, the `asyncio` module is still a bit raw and unfinished, but there should not be any obstacles in using it. Creating a fully functional server/client setup is still a tad complicated, but the most obvious use of `asyncio` is the handling of basic network I/O such as database connections and external resources such as websites. Especially, the latter takes only a few lines to implement with the use of `asyncio`, removing some very important bottlenecks from your code.

The point of this chapter is understanding how to tell Python to wait for results in the background instead of simply waiting or polling for them as usual. In *Chapter 13, Multiprocessing – When a Single CPU Core Is Not Enough* you will learn about multiprocessing, which is also an option for handling stalling resources. However, the goal of multiprocessing is actually to use multiple processors instead of handling stalling resources. When it comes to potentially slow external resources, I recommend that you always use `asyncio`, if at all possible.

When building utilities based on the `asyncio` library, make sure you search for premade libraries to solve your problems, as many of them are currently being developed. While writing this chapter, Python 3.5 was not officially out yet, so the odds are that a lot more documentation and libraries using the `async/await` syntax will pop up soon. To make sure you do not repeat work that others have done, search the Internet thoroughly before writing your own code extending on `asyncio`.

The next chapter will explain a completely different topic—the construction of classes using metaclasses. Regular classes are created using the type class, but now we will see how we can extend and modify the default behavior to make a class do pretty much anything we want. Metaclasses even make it possible to have automatically registering plugins and add features to classes in a very magical way—in short, how to customize not just the class instances but the class definitions themselves.

8
Metaclasses – Making Classes (Not Instances) Smarter

The previous chapters have already shown us how to modify classes and functions using decorators. But that's not the only option to modify or extend a class. An even more advanced technique of modifying your classes before creation is the usage of **metaclasses**. The name already gives a hint to what it could be; a metaclass is a class containing meta information about a class.

The basic premise of a metaclass is a class that generates another class for you at definition time, so generally you wouldn't use it to change the class instances but only the class definitions. By changing the class definitions, it is possible to automatically add some properties to a class, validate whether certain properties are set, change inheritance, automatically register the class at a manager, and do many other things.

Although metaclasses are generally considered to be a more powerful technique than (class) decorators, effectively they don't differ too much in possibilities. The choice usually comes down to either convenience or personal preference.

The following topics are covered in this chapter:

- Basic dynamic class creation
- Metaclasses with arguments
- Internals of class creation, the order of operations
- Abstract base classes, examples and inner workings
- Automatic plugin system using metaclasses
- Storing definition order of class attributes

Dynamically creating classes

Metaclasses are the factories that create new classes in Python. In fact, even though you may not be aware of it, Python will always execute the `type` metaclass whenever you create a class.

When creating classes in a procedural way, the `type` metaclass is used as a function. This function takes three arguments: `name`, `bases`, and `dict`. The name will become the `__name__` attribute, the `bases` is the list of inherited base classes and will be stored in `__bases__` and `dict` is the namespace dictionary that contains all variables and will be stored in `__dict__`.

It should be noted that the `type()` function has another use as well. Given the arguments documented earlier, it creates a class given those specifications. Given a single argument with the instance of a class, it will return the class as well but from the instance. Your next question might be, "What happens if I call `type()` on a class definition instead of a class instance?" Well, that returns the metaclass for the class which is `type` by default.

Let's clarify this using a few examples:

```
>>> class Spam(object):
>>>     eggs = 'my eggs'


>>> Spam = type('Spam', (object,), dict(eggs='my eggs'))
```

The preceding two definitions of `Spam` are completely identical; they both create a class with an instantiated property of `eggs` and `object` as a base. Let's test if this actually works as you would expect:

```
>>> class Spam(object):
...     eggs = 'my eggs'


>>> spam = Spam()
>>> spam.eggs
'my eggs'
>>> type(spam)
<class '…Spam'>
>>> type(Spam)
<class 'type'>
```

```
>>> Spam = type('Spam', (object,), dict(eggs='my eggs'))


>>> spam = Spam()
>>> spam.eggs
'my eggs'
>>> type(spam)
<class '...Spam'>
>>> type(Spam)
<class 'type'>
```

As expected, the results for the two are the same. When creating a class, Python silently adds the `type` metaclass and `custom` metaclasses are simply classes that inherit `type`. A simple class definition has a silent metaclass making a simple definition such as:

```
class Spam(object):
    pass
```

Essentially identical to:

```
class Spam(object, metaclass=type):
    pass
```

This raises the question that if every class is created by a (silent) metaclass, what is the metaclass of `type`? This is actually a recursive definition; the metaclass of `type` is `type`. This is the essence of what a custom metaclass is: a class that inherits type to allow class modification without needing to modify the class definition itself.

A basic metaclass

Since metaclasses can modify any class attribute, you can do absolutely anything you wish. Before we continue with more advanced metaclasses, let's look at a basic example:

```
# The metaclass definition, note the inheritance of type instead
# of object
>>> class MetaSpam(type):
...
...     # Notice how the __new__ method has the same arguments
...     # as the type function we used earlier?
...     def __new__(metaclass, name, bases, namespace):
...         name = 'SpamCreatedByMeta'
```

```
...         bases = (int,) + bases
...         namespace['eggs'] = 1
...         return type.__new__(metaclass, name, bases, namespace)


# First, the regular Spam:
>>> class Spam(object):
...     pass

>>> Spam.__name__
'Spam'
>>> issubclass(Spam, int)
False
>>> Spam.eggs
Traceback (most recent call last):
    ...
AttributeError: type object 'Spam' has no attribute 'eggs'


# Now the meta-Spam
>>> class Spam(object, metaclass=MetaSpam):
...     pass

>>> Spam.__name__
'SpamCreatedByMeta'
>>> issubclass(Spam, int)
True
>>> Spam.eggs
1
```

As you can see, everything about the class definition can easily be modified using metaclasses. This makes it both a very powerful and a very dangerous tool, as you can easily cause very unexpected behavior.

Arguments to metaclasses

The possibility of adding arguments to a metaclass is a little-known feature, but very useful nonetheless. In many cases, simply adding attributes or methods to a class definition is enough to detect what to do, but there are cases where it is useful to be more specific.

```
>>> class MetaWithArguments(type):
...     def __init__(metaclass, name, bases, namespace, **kwargs):
...         # The kwargs should not be passed on to the
...         # type.__init__
...         type.__init__(metaclass, name, bases, namespace)
...
...     def __new__(metaclass, name, bases, namespace, **kwargs):
...         for k, v in kwargs.items():
...             namespace.setdefault(k, v)
...
...         return type.__new__(metaclass, name, bases, namespace)


>>> class WithArgument(metaclass=MetaWithArguments, spam='eggs'):
...     pass


>>> with_argument = WithArgument()
>>> with_argument.spam
'eggs'
```

This simplistic example may not be useful but the possibilities are. The only thing you need to keep in mind is that both the `__new__` and `__init__` methods need to be extended for this to work.

Accessing metaclass attributes through classes

When using metaclasses, it might be confusing to note that the class actually does more than simply construct the class, it actually inherits the class during the creation. To illustrate:

```
>>> class Meta(type):
...
...     @property
```

```
...     def spam(cls):
...         return 'Spam property of %r' % cls
...
...     def eggs(self):
...         return 'Eggs method of %r' % self


>>> class SomeClass(metaclass=Meta):
...     pass

>>> SomeClass.spam
"Spam property of <class '...SomeClass'>"
>>> SomeClass().spam
Traceback (most recent call last):
    ...
AttributeError: 'SomeClass' object has no attribute 'spam'

>>> SomeClass.eggs()
"Eggs method of <class '...SomeClass'>"
>>> SomeClass().eggs()
Traceback (most recent call last):
    ...
AttributeError: 'SomeClass' object has no attribute 'eggs'
```

As can be seen in the preceding example, these methods are only available for the `class` objects and not the instances. The `spam` attribute and the `eggs` method are not accessible through the instance while they are accessible through the class. I personally don't see any useful cases for this behavior but it is definitely noteworthy.

Abstract classes using collections.abc

The abstract base classes module is one of the most useful and most used examples of metaclasses in Python, as it makes it easy to ensure that a class adheres to a certain interface without a lot of manual checks. We have already seen some examples of abstract base classes in the previous chapters, but now we will look at the inner workings of these and the more advanced features, such as custom ABCs.

Internal workings of the abstract classes

First, let's demonstrate the usage of the regular abstract base class:

```
>>> import abc

>>> class Spam(metaclass=abc.ABCMeta):
...
...     @abc.abstractmethod
...     def some_method(self):
...         raise NotImplemented()


>>> class Eggs(Spam):
...     def some_new_method(self):
...         pass


>>> eggs = Eggs()
Traceback (most recent call last):
    ...
TypeError: Can't instantiate abstract class Eggs with abstract
methods some_method


>>> class Bacon(Spam):
...     def some_method():
...         pass


>>> bacon = Bacon()
```

As you can see, the abstract base class blocks us from instantiating the classes until all the abstract methods have been inherited. In addition to the regular methods, `property`, `staticmethod`, and `classmethod` are also supported.

```
>>> import abc

>>> class Spam(object, metaclass=abc.ABCMeta):
...     @property
```

```
...     @abc.abstractmethod
...     def some_property(self):
...         raise NotImplemented()
...
...     @classmethod
...     @abc.abstractmethod
...     def some_classmethod(cls):
...         raise NotImplemented()
...
...     @staticmethod
...     @abc.abstractmethod
...     def some_staticmethod():
...         raise NotImplemented()
...
...     @abc.abstractmethod
...     def some_method():
...         raise NotImplemented()
```

So what does Python do internally? You could, of course, read the `abc.py` source code but I think a simple explanation would be better.

First, `abc.abstractmethod` sets the `__isabstractmethod__` property on the function to `True`. So if you don't want to use the decorator, you can simply emulate the behavior by doing something along the lines of:

```
some_method.__isabstractmethod__ = True
```

After that, the `abc.ABCMeta` metaclass walks through all the items in a namespace and looks for objects where the `__isabstractmethod__` attribute evaluates to `True`. In addition to that, it walks through all bases and checks the `__abstractmethods__` set for every base class, in case the class inherits an `abstract` class. All the items where `__isabstractmethod__` still evaluates to `True` get added to the `__abstractmethods__` set which is stored in the class as `frozenset`.

Note that we don't use `abc.abstractproperty`, `abc.abstractclassmethod`, and `abc.abstractstaticmethod`. Since Python 3.3 these have been deprecated as the `classmethod`, `staticmethod`, and `property` decorators are recognized by `abc.abstractmethod` so a simple `property` decorator followed by a `abc.abstractmethod` is recognized as well. Take care when ordering the decorators; `abc.abstractmethod` needs to be the innermost decorator for this to work properly.

The next question now is about where the actual checks come in; the checks to see if the classes are completely implemented. This actually functions through a few Python internals:

```
>>> class AbstractMeta(type):
...     def __new__(metaclass, name, bases, namespace):
...         cls = super().__new__(metaclass, name, bases, namespace)
...         cls.__abstractmethods__ = frozenset(('something',))
...         return cls


>>> class Spam(metaclass=AbstractMeta):
...     pass

>>> eggs = Spam()
Traceback (most recent call last):
    ...
TypeError: Can't instantiate abstract class Spam with ...
```

We can easily emulate the same behavior with a `metaclass` ourselves, but it should be noted that `abc.ABCMeta` actually does more, which we will demonstrate in the next section. To mimic the behavior of the built-in abstract base class support, take a look at the following example:

```
>>> import functools


>>> class AbstractMeta(type):
...     def __new__(metaclass, name, bases, namespace):
...         # Create the class instance
...         cls = super().__new__(metaclass, name, bases, namespace)
...
...         # Collect all local methods marked as abstract
...         abstracts = set()
...         for k, v in namespace.items():
...             if getattr(v, '__abstract__', False):
...                 abstracts.add(k)
...
...         # Look for abstract methods in the base classes and add
```

```
...         # them to the list of abstracts
...         for base in bases:
...             for k in getattr(base, '__abstracts__', ()):
...                 v = getattr(cls, k, None)
...                 if getattr(v, '__abstract__', False):
...                     abstracts.add(k)
...
...         # store the abstracts in a frozenset so they cannot be
...         # modified
...         cls.__abstracts__ = frozenset(abstracts)
...
...         # Decorate the __new__ function to check if all abstract
...         # functions were implemented
...         original_new = cls.__new__
...         @functools.wraps(original_new)
...         def new(self, *args, **kwargs):
...             for k in self.__abstracts__:
...                 v = getattr(self, k)
...                 if getattr(v, '__abstract__', False):
...                     raise RuntimeError(
...                         '%r is not implemented' % k)
...
...             return original_new(self, *args, **kwargs)
...
...         cls.__new__ = new
...         return cls


>>> def abstractmethod(function):
...     function.__abstract__ = True
...     return function


>>> class Spam(metaclass=AbstractMeta):
...     @abstractmethod
...     def some_method(self):
```

```
...         pass

# Instantiating the function, we can see that it functions as the
# regular ABCMeta does
>>> eggs = Spam()
Traceback (most recent call last):
    ...
RuntimeError: 'some_method' is not implemented
```

The actual implementation is a bit more complicated as it still needs to take care of the old style classes and the `property`, `classmethod`, and `staticmethod` types of methods. Additionally, it features caching, but this code covers the most useful part of the implementation. One of the most important tricks to note here is that the actual check is executed by decorating the `__new__` function of the actual class. This method is only executed once within a class so we can avoid the overhead of these checks for multiple instantiations.

The actual implementation of the abstract methods can be found by looking for `__isabstractmethod__` in the Python source code in the following files: `Objects/descrobject.c`, `Objects/funcobject.c`, and `Objects/object.c`. The Python part of the implementation can be found in `Lib/abc.py`.

Custom type checks

Defining your own interfaces using abstract base classes is great, of course. But it can also be very convenient to tell Python what your class actually resembles and what kind of types are similar. For that, `abc.ABCMeta` offers a register function which allows you to specify which types are similar. For example, a custom list that sees the list type as similar:

```
>>> import abc

>>> class CustomList(abc.ABC):
...     'This class implements a list-like interface'
...     pass

>>> CustomList.register(list)
<class 'list'>
```

```
>>> issubclass(list, CustomList)
True
>>> isinstance([], CustomList)
True
>>> issubclass(CustomList, list)
False
>>> isinstance(CustomList(), list)
False
```

As demonstrated with the last four lines, this is a one-way relationship. The other way around would generally be easy enough to realize through inheriting list, but that won't work in this case. `abc.ABCMeta` refuses to create inheritance cycles.

```
>>> import abc


>>> class CustomList(abc.ABC, list):
...     'This class implements a list-like interface'
...     pass


>>> CustomList.register(list)
Traceback (most recent call last):
    ...
RuntimeError: Refusing to create an inheritance cycle
```

To be able to handle cases like these, there is another useful feature in `abc.ABCMeta`. When subclassing `abc.ABCMeta`, the `__subclasshook__` method can be extended to customize the behavior of `issubclass` and with that, `isinstance`.

```
>>> import abc


>>> class UniversalClass(abc.ABC):
...     @classmethod
...     def __subclasshook__(cls, subclass):
...         return True



>>> issubclass(list, UniversalClass)
True
>>> issubclass(bool, UniversalClass)
```

```
True
>>> isinstance(True, UniversalClass)
True
>>> issubclass(UniversalClass, bool)
False
```

The `__subclasshook__` should return `True`, `False`, or `NotImplemented`, which would result in `issubclass` returning `True`, `False`, or the usual behavior when `NotImplemented` is raised.

Using abc.ABC before Python 3.4

The `abc.ABC` class we have used in this paragraph is only available in Python versions 3.4 and higher, but it's trivial to implement it in older versions. It's little more than syntactic sugar for `metaclass=abc.ABCMeta`. To implement it yourself, you can simply use the following snippet:

```
import abc

class ABC(metaclass=abc.ABCMeta):
    pass
```

Automatically registering a plugin system

One of the most common uses of metaclasses is to have classes automatically register themselves as plugins/handlers. Examples of these can be seen in many projects, such as web frameworks. Those codebases are too extensive to usefully explain here though. Hence, we'll show a simpler example showing the power of metaclasses as a self-registering `plugin` system:

```
>>> import abc



>>> class Plugins(abc.ABCMeta):
...     plugins = dict()
...
...     def __new__(metaclass, name, bases, namespace):
...         cls = abc.ABCMeta.__new__(metaclass, name, bases,
...                                   namespace)
...         if isinstance(cls.name, str):
...             metaclass.plugins[cls.name] = cls
```

```
...         return cls
...
...     @classmethod
...     def get(cls, name):
...         return cls.plugins[name]


>>> class PluginBase(metaclass=Plugins):
...     @property
...     @abc.abstractmethod
...     def name(self):
...         raise NotImplemented()


>>> class SpamPlugin(PluginBase):
...     name = 'spam'


>>> class EggsPlugin(PluginBase):
...     name = 'eggs'


>>> Plugins.get('spam')
<class '...SpamPlugin'>
>>> Plugins.plugins
{'spam': <class '...SpamPlugin'>,
 'eggs': <class '...EggsPlugin'>}
```

This example is a tad simplistic of course, but it's the basis for many plugin systems. Which is a very important thing to note while implementing systems like these; however, while metaclasses run at definition time, the module still needs to be imported to work. There are several options to do this; loading on-demand through the `get` method has my vote as that also doesn't add load time if the plugin is not used.

The following examples will use the following file structure to get reproducible results. All files will be contained in a plugins directory.

The `__init__.py` file is used to create shortcuts, so a simple import plugins will result in having `plugins.Plugins` available, instead of requiring importing `plugins.base` explicitly.

```
# plugins/__init__.py
from .base import Plugin
from .base import Plugins

__all__ = ['Plugin', 'Plugins']
```

The `base.py` file containing the `Plugins` collection and the `Plugin` base class:

```
# plugins/base.py
import abc


class Plugins(abc.ABCMeta):
    plugins = dict()

    def __new__(metaclass, name, bases, namespace):
        cls = abc.ABCMeta.__new__(
            metaclass, name, bases, namespace)
        if isinstance(cls.name, str):
            metaclass.plugins[cls.name] = cls
        return cls

    @classmethod
    def get(cls, name):
        return cls.plugins[name]


class Plugin(metaclass=Plugins):
    @property
    @abc.abstractmethod
    def name(self):
        raise NotImplemented()
```

And two simple plugins, `spam.py`:

```
from . import base


class Spam(base.Plugin):
    name = 'spam'
```

And `eggs.py`:

```
from . import base


class Eggs(base.Plugin):
    name = 'eggs'
```

Importing plugins on-demand

The first of the solutions for the import problem is simply taking care of it in the `get` method of the `Plugins` metaclass. Whenever the plugin is not found in the registry, it should automatically load the module from the `plugins` directory.

The advantages of this approach are that not only the plugins don't explicitly need to be preloaded but also that the plugins are only loaded when the need is there. Unused plugins are not touched, so this method can help in reducing your applications' load times.

The downside is that the code will not be run or tested, so it might be completely broken and you won't know about it until it is finally loaded. Solutions for this problem will be covered in the testing chapter, *Chapter 10, Testing and Logging – Preparing for Bugs*. The other problem is that if the code self-registers itself into other parts of an application then that code won't be executed either.

Modifying the `Plugins.get` method, we get the following:

```
import abc
import importlib


class Plugins(abc.ABCMeta):
    plugins = dict()

    def __new__(metaclass, name, bases, namespace):
        cls = abc.ABCMeta.__new__(
            metaclass, name, bases, namespace)
        if isinstance(cls.name, str):
            metaclass.plugins[cls.name] = cls
        return cls

    @classmethod
    def get(cls, name):
        if name not in cls.plugins:
            print('Loading plugins from plugins.%s' % name)
            importlib.import_module('plugins.%s' % name)
        return cls.plugins[name]
```

This results in the following when executing:

```
>>> import plugins
>>> plugins.Plugins.get('spam')
Loading plugins from plugins.spam
<class 'plugins.spam.Spam'>

>>> plugins.Plugins.get('spam')
<class 'plugins.spam.Spam'>
```

As you can see, this approach only results in running `import` once. The second time, the plugin will be available in the plugins dictionary so no loading will be necessary.

Importing plugins through configuration

While only loading the needed plugins is generally a better idea, there is something to be said to preload the plugins you will likely need. As explicit is better than implicit, an explicit list of plugins to load is generally a good solution. The added advantages of this method are that firstly you are able to make the registration a bit more advanced as you are guaranteed that it is run and secondly you can load plugins from multiple packages.

Instead of importing in the `get` method, we will add a `load` method this time; a `load` method that imports all the given module names:

```
import abc
import importlib


class Plugins(abc.ABCMeta):
    plugins = dict()

    def __new__(metaclass, name, bases, namespace):
        cls = abc.ABCMeta.__new__(
            metaclass, name, bases, namespace)
        if isinstance(cls.name, str):
            metaclass.plugins[cls.name] = cls
        return cls

    @classmethod
    def get(cls, name):
        return cls.plugins[name]
```

```
        @classmethod
        def load(cls, *plugin_modules):
            for plugin_module in plugin_modules:
                plugin = importlib.import_module(plugin_module)
```

Which can be called using the following code:

```
>>> import plugins


>>> plugins.Plugins.load(
...     'plugins.spam',
...     'plugins.eggs',
... )


>>> plugins.Plugins.get('spam')
<class 'plugins.spam.Spam'>
```

A fairly simple and straightforward system to load the plugins based on settings, this can easily be combined with any type of settings system to fill the `load` method.

Importing plugins through the file system

Whenever possible, it is best to avoid having systems depend on automatic detection of modules on a filesystem as it goes directly against `PEP8`. Specifically, "explicit is better than implicit". While these systems can work fine in specific cases, they often make debugging much more difficult. Similar automatic import systems in Django have caused me a fair share of headaches as they tend to obfuscate the errors. Having said that, automatic plugin loading based on all the files in a plugins directory is still a possibility warranting a demonstration.

```
    import os
    import re
    import abc
    import importlib

    MODULE_NAME_RE = re.compile('[a-z][a-z0-9_]*', re.IGNORECASE)

    class Plugins(abc.ABCMeta):
        plugins = dict()

        def __new__(metaclass, name, bases, namespace):
            cls = abc.ABCMeta.__new__(
                metaclass, name, bases, namespace)
```

```
            if isinstance(cls.name, str):
                metaclass.plugins[cls.name] = cls
            return cls

        @classmethod
        def get(cls, name):
            return cls.plugins[name]

        @classmethod
        def load_directory(cls, module, directory):
            for file_ in os.listdir(directory):
                name, ext = os.path.splitext(file_)
                full_path = os.path.join(directory, file_)
                import_path = [module]
                if os.path.isdir(full_path):
                    import_path.append(file_)
                elif ext == '.py' and MODULE_NAME_RE.match(name):
                    import_path.append(name)
                else:
                    # Ignoring non-matching files/directories
                    continue

                plugin = importlib.import_module('.'.join(import_path))

        @classmethod
        def load(cls, **plugin_directories):
            for module, directory in plugin_directories.items():
                cls.load_directory(module, directory)
```

If possible, I would try to avoid using a fully automatic import system as it's very prone to accidental errors and can make debugging more difficult, not to mention that the import order cannot easily be controlled this way. To make this system a bit smarter (even importing packages outside of your Python path), you can create a plugin loader using the abstract base classes in `importlib.abc`. Note that you will most likely still need to list the directories through `os.listdir` or `os.walk` though.

Order of operations when instantiating classes

The order of operations during class instantiation is very important to keep in mind when debugging issues with dynamically created and/or modified classes. The instantiation of a class happens in the following order.

Finding the metaclass

The metaclass comes from either the explicitly given metaclass on the class or `bases`, or by using the default `type` metaclass.

For every class, the class itself and the bases, the first matching of the following will be used:

- Explicitly given metaclass
- Explicit metaclass from bases
- `type()`

Note that if no metaclass is found that is a subtype of all the candidate metaclasses, a `TypeError` will be raised. This scenario is not that likely to occur but certainly a possibility when using multiple inheritance/mixins with metaclasses.

Preparing the namespace

The class namespace is prepared through the metaclass selected previously. If the metaclass has a `__prepare__` method, it will be called `namespace = metaclass.__prepare__(names, bases, **kwargs)`, where `**kwargs` originates from the class definition. If no `__prepare__` method is available, the result will be `namespace = dict()`.

Note that there are multiple ways of achieving custom namespaces, as we saw in the previous paragraph, the `type()` function call also takes a `dict` argument which can be used to alter the namespace as well.

Executing the class body

The body of the class is executed very similarly to normal code execution with one key difference, the separate namespace. Since a class has a separate namespace, which shouldn't pollute the `globals()/locals()` namespaces, it is executed within that context. The resulting call looks something like this: `exec(body, globals(), namespace)` where `namespace` is the previously produced namespace.

Creating the class object (not instance)

Now that we have all the components ready, the actual class object can be produced. This is done through the `class_ = metaclass(name, bases, namespace, **kwargs)` call. This is, as you can see, actually identical to the `type()` call previously discussed. `**kwargs` here are the same as the ones passed to the `__prepare__` method earlier.

It might be useful to note that this is also the object that will be referenced from the `super()` call without arguments.

Executing the class decorators

Now that the class object is actually done already, the class decorators will be executed. Since this is only executed after everything else in the class object has already been constructed, it becomes difficult to modify class attributes, such as which classes are being inherited, and the name of the class. By modifying the `__class__` object you can still modify or overwrite these, but it is, at the very least, more difficult.

Creating the class instance

From the class object produced previously, we can now finally create the actual instances as you normally would with a class. It should be noted that this step and the class decorators steps, unlike the earlier steps, are the only ones that are executed every time you instantiate a class. The steps before these two are only executed once per class definition.

Example

Enough theory! Let's illustrate the creation and instantiation of the class objects so we can check the order of operations:

```
>>> import functools



>>> def decorator(name):
...     def _decorator(cls):
...         @functools.wraps(cls)
...         def __decorator(*args, **kwargs):
...             print('decorator(%s)' % name)
...             return cls(*args, **kwargs)
```

```
...         return __decorator
...     return _decorator


>>> class SpamMeta(type):
...
...     @decorator('SpamMeta.__init__')
...     def __init__(self, name, bases, namespace, **kwargs):
...         print('SpamMeta.__init__()')
...         return type.__init__(self, name, bases, namespace)
...
...     @staticmethod
...     @decorator('SpamMeta.__new__')
...     def __new__(cls, name, bases, namespace, **kwargs):
...         print('SpamMeta.__new__()')
...         return type.__new__(cls, name, bases, namespace)
...
...     @classmethod
...     @decorator('SpamMeta.__prepare__')
...     def __prepare__(cls, names, bases, **kwargs):
...         print('SpamMeta.__prepare__()')
...         namespace = dict(spam=5)
...         return namespace


>>> @decorator('Spam')
... class Spam(metaclass=SpamMeta):
...
...     @decorator('Spam.__init__')
...     def __init__(self, eggs=10):
...         print('Spam.__init__()')
...         self.eggs = eggs
decorator(SpamMeta.__prepare__)
SpamMeta.__prepare__()
decorator(SpamMeta.__new__)
SpamMeta.__new__()
```

```
decorator(SpamMeta.__init__)
SpamMeta.__init__()


# Testing with the class object
>>> spam = Spam
>>> spam.spam
5
>>> spam.eggs
Traceback (most recent call last):
    ...
AttributeError: ... object has no attribute 'eggs'


# Testing with a class instance
>>> spam = Spam()
decorator(Spam)
decorator(Spam.__init__)
Spam.__init__()
>>> spam.spam
5
>>> spam.eggs
10
```

The example clearly shows the creation order of the class:

1. Preparing the namespace through `__prepare__`.
2. Creating the class body using `__new__`.
3. Initializing the metaclass using `__init__` (note that this is not the class `__init__`).
4. Initializing the class through the class decorator.
5. Initializing the class through the class `__init__` function.

One thing we can note from this is that the class decorators are executed each and every time the class is actually instantiated and not before that. This can be both an advantage and a disadvantage of course, but if you wish to build a register of all subclasses, it is definitely more convenient to use a metaclass since the decorator will not register until you instantiate the class.

In addition to this, having the power to modify the namespace before actually creating the class object (not the instance) can be very powerful as well. It can be convenient for sharing a certain scope between several class objects for example, or for easily ensuring that certain items are always available in the scope.

Storing class attributes in definition order

There are cases where the definition order makes a difference. For example, let's assume we are creating a class that represents a CSV (Comma Separated Values) format. The CSV format expects the fields to have a particular order. In some cases this will be indicated by a header but it's still useful to have a consistent field order. Similar systems are using in ORM systems such as SQLAlchemy to store the column order for table definitions and for the input field order within forms in Django.

The classic solution without metaclasses

An easy way to store the order of the fields is by giving the field instances a special `__init__` method which increments for every definition, so the fields have an incrementing index property. This solution can be considered the classic solution as it also works in Python 2.

```
>>> import itertools

>>> class Field(object):
...     counter = itertools.count()
...
...     def __init__(self, name=None):
...         self.name = name
...         self.index = next(Field.counter)
...
...     def __repr__(self):
...         return '<%s[%d] %s>' % (
...             self.__class__.__name__,
...             self.index,
...             self.name,
...         )


>>> class FieldsMeta(type):
```

```
...     def __new__(metaclass, name, bases, namespace):
...         cls = type.__new__(metaclass, name, bases, namespace)
...         fields = []
...         for k, v in namespace.items():
...             if isinstance(v, Field):
...                 fields.append(v)
...                 v.name = v.name or k
...
...         cls.fields = sorted(fields, key=lambda f: f.index)
...         return cls


>>> class Fields(metaclass=FieldsMeta):
...     spam = Field()
...     eggs = Field()

>>> Fields.fields
[<Field[0] spam>, <Field[1] eggs>]

>>> fields = Fields()
>>> fields.eggs.index
1
>>> fields.spam.index
0
>>> fields.fields
[<Field[0] spam>, <Field[1] eggs>]
```

For convenience, and to make things prettier, we have added the `FieldsMeta` class. It is not strictly required here, but it automatically takes care of filling in the name if needed, and adds the `fields` list which contains a sorted list of fields.

Using metaclasses to get a sorted namespace

The previous solution is a bit more straightforward and supports Python 2 as well, but with Python 3 we have more options. As you have seen in the previous paragraphs, since Python 3 we have the `__prepare__` method, which returns the namespace. From the previous chapters you might also remember `collections.OrderedDict`, so let's see what happens when we combine them.

```
>>> import collections

>>> class Field(object):
```

```
...     def __init__(self, name=None):
...         self.name = name
...
...     def __repr__(self):
...         return '<%s %s>' % (
...             self.__class__.__name__,
...             self.name,
...         )


>>> class FieldsMeta(type):
...     @classmethod
...     def __prepare__(metaclass, name, bases):
...         return collections.OrderedDict()
...
...     def __new__(metaclass, name, bases, namespace):
...         cls = type.__new__(metaclass, name, bases, namespace)
...         cls.fields = []
...         for k, v in namespace.items():
...             if isinstance(v, Field):
...                 cls.fields.append(v)
...                 v.name = v.name or k
...
...         return cls


>>> class Fields(metaclass=FieldsMeta):
...     spam = Field()
...     eggs = Field()

>>> Fields.fields
[<Field spam>, <Field eggs>]
>>> fields = Fields()
>>> fields.fields
[<Field spam>, <Field eggs>]
```

As you can see, the fields are indeed in the order we defined them. `Spam` first and `eggs` after that. Since the class namespace is now a `collections.OrderedDict` instance, we know that the order is guaranteed. Instead of the regular not predetermined order of the Python `dict`. This demonstrates how convenient metaclasses can be to extend your classes in a generic way. Another big advantage of metaclasses, instead of a custom `__init__` method, is that the users won't lose the functionality if they forget to call the parent `__init__` method. The metaclass will always be executed, unless a different metaclass is added, that is.

Summary

The Python metaclass system is something every Python programmer uses all the time, perhaps without even knowing about it. Every class should be created through some (subclass of) `type`, which allows for endless customization and magic. Instead of statically defining your class, you can now have it created as you normally would and dynamically add, modify, or remove attributes from your class during definition; very magical but very useful. The magic component, however, is also the reason it should be used with a lot of caution. While metaclasses can be used to make your life much easier, they are also amongst the easiest ways of producing completely incomprehensible code.

Regardless, there are some great use-cases for metaclasses and many libraries such as `SQLAlchemy` and `Django` use metaclasses to make your code work much easier and arguably better. Actually comprehending the magic that is used inside is generally not needed for the usage of these libraries, which makes the cases defendable. The question becomes whether a much better experience for beginners is worth some dark magic internally, and looking at the success of these libraries, I would say yes in this case.

To conclude, when thinking about using metaclasses, keep in mind what Tim Peters once said: "Metaclasses are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't."

Now we will continue with a solution to remove some of the magic that metaclasses generate: documentation. The next chapter will show us how your code can be documented, how that documentation can be tested, and most importantly, how the documentation can be made smarter by annotating types in your documentation.

9

Documentation – How to Use Sphinx and reStructuredText

Documenting code can be both fun and useful! I will admit that many programmers have a strong dislike for documenting code and understandably so. Writing documentation can be a boring job and traditionally only others reap the benefits of that effort. The tools available for Python, however, make it almost trivial to generate useful and up-to-date documentation with little to no effort at all. Generating documentation has actually become so easy that I create and generate documentation before using a Python package. Assuming it wasn't available already, that is.

In addition to simple text documentation explaining what a function does, it is also possible to add metadata, such as type hints. These type hints can be used to make the arguments and return types of a function or class clickable in the documentation. But more importantly, many modern IDEs and editors, such as VIM, have plugins available that parse the type hints and use them for intelligent auto-completion. So if you type `Spam.eggs`, your editor will automatically complete the specific attributes and methods of the eggs object; something that is traditionally only viable with statically typed languages such as Java, C, and C++.

This chapter will explain the types of documentation available in Python and how easily a full set of documentation can be created. With the amazing tools that Python provides, you can have fully functioning documentation within minutes.

Topics covered in this chapter are as follows:

- The reStructuredText syntax
- Setting up documentation using Sphinx
- Sphinx style docstrings
- Google style docstrings
- NumPy style docstrings

The reStructuredText syntax

The **reStructuredText** format (also known as **RST**, **ReST**, or **reST**) was developed in 2002 as a simple language that implements enough markup to be usable, but is simple enough to be readable as plain text. These two features make it readable enough to use in code, yet still versatile enough to generate pretty and useful documentation.

The greatest thing about reStructuredText is that it is very intuitive. Even without knowing anything about the standard, you can easily write documentation in this style without ever knowing that it would be recognized as a language. However, more advanced techniques, such as images and links, do require some explanation.

Next to reStructuredText, there are also languages such as **Markdown** which are quite similar in usage. Within the Python community, reStructuredText has been the standard documentation language for over 10 years, making it the recommended solution.

To easily convert between formats such as reStructuredText and Markdown, use the Pandoc tool, available at `http://pandoc.org/`.

The basic syntax reads just like text and the next few paragraphs will show some of the more advanced features. However, let us start with a simple example demonstrating how simple a reStructuredText file can be.

```
Documentation, how to use Sphinx and reStructuredText
######################################################################

Documenting code can be both fun and useful! ...

Additionally, adding ...

... So that typing `Spam.eggs.` will automatically ...

Topics covered in this chapter are as follows:

 - The reStructuredText syntax
 - Setting up documentation using Sphinx
 - Sphinx style docstrings
 - Google style docstrings
 - NumPy style docstrings
```

```
The reStructuredText syntax
***************************************************************


The reStructuredText format (also known as ...
```

That's how easy it is to convert the text of this chapter so far to reStructuredText. The following paragraphs will cover the following features:

1. Inline markup (italic, bold, code, and links)
2. Lists
3. Headers
4. Advanced links
5. Images
6. Substitutions
7. Blocks containing code, math, and others

Getting started with reStructuredText

To quickly convert a reStructuredText file to HTML, we can use the `docutils` library. The `sphinx` library discussed later in this chapter actually uses the `docutils` library internally, but has some extra features that we won't need initially. To get started, we just need to install `docutils`:

```
pip install docutils
```

After that we can easily convert reStructuredText into PDF, LaTeX, HTML, and other formats. For the examples in this paragraph, we'll use the HTML format which is easily generated using the following command:

```
rst2html.py file.rst file.html
```

The basic components of reStructuredText are roles, which are used for inline modifications of the output and directives to generate markup blocks. Within pure reStructuredText, the directives are the most important, but we will see many uses for the roles in the section about Sphinx.

Inline markup

Inline markup is the markup that is used within a regular line of text. Examples of these are emphasis, in-line code examples, links, images, and bullet lists.

Emphasis, for example, can be added by encapsulating the words between one or two asterisk signs. This sentence for example could add a little bit of `*emphasis*` by adding a single asterisk on both sides or a lot of `**emphasis**` by adding two asterisks at both sides. There are many different inline markup directives so we will list only the most common ones. A full list can always be found through the reStructuredText homepage at `docutils.sourceforge.net`.

Following are some examples:

- Emphasis (italic) text: `*emphasis for this phrase*`.
- Extra emphasis (bold) text: `**extra emphasis for this phrase**`.
- For lists without numbers, a simple dash with spaces after it:

```
- item 1
- item 2
```

The space after the dash is required for reStructuredText to recognize the list.

- For lists with numbers, the number followed by a period and a space:

```
1. item 1
2. item 2
```

- For numbered lists, the period after the number is required.
- Interpreted text: These are domain specific. Within Python documentation, the default role is code which means that surround text with back ticks will convert your code to use code tags. For example, `` `if spam and eggs:` ``.

  Different roles can be set through either a role prefix or suffix depending on your preference. For example, `:math:`E=mc^2`` to show mathematical equations.

- Inline literals: This is formatted with a mono-space font, which makes it ideal for inline code. Just add two back ticks to ``` ``add some code`` ```.
- References: These can be created through a trailing underscore. They can point to headers, links, labels, and more. The next section will cover more about these, but the basic syntax is simply `reference_` or enclosed in back ticks when the reference contains spaces, `` `some reference link`_ ``.
- To escape the preceding characters, the backslash can be used. So if you wish to have an asterisk with emphasis, it's possible to use `*\**`, quite similar to escaping in Python strings.

There are many more available, but these are the ones you will use the most when writing reStructuredText.

Headers

The headers are used to indicate the start of a document, section, chapter, or paragraph. It is therefore the first structure you need in a document. While not strictly needed, its usage is highly recommended as it serves several purposes:

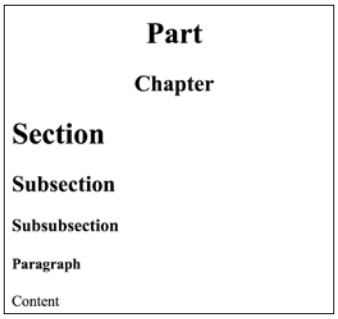
1. The headers are consistently formatted according to their level.
2. Sphinx can generate a Table Of Contents (TOC) tree from the headers.
3. All headers automatically function as labels, which means you can create links towards them.

When creating headers, consistency is one of the few constraints; the character used is fairly arbitrary as is the amount of levels.

Personally, I default to a simple system with a fixed-size header, but I recommend at least following the default of the Python documentation in terms of the parts, chapters, sections, subsections, subsubsections, and paragraphs. Something along the lines of the following:

```
Part
####################################################################

Chapter
********************************************************************

Section
====================================================================

Subsection
--------------------------------------------------------------------

Subsubsection
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Paragraph
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

Content
```

Output:



That is just the common usage of the headers, but the main idea of reStructuredText is that you can use just about anything that feels natural to you, which means that you can use any of the following characters: `= - ` : ' " ~ ^ _ * + # <>`. It also supports both underlines and overlines, so if you prefer that, they are options as well:

```
######################################################################
Part
######################################################################

**********************************************************************
Chapter
**********************************************************************

======================================================================
Section
======================================================================

----------------------------------------------------------------------
Subsection
----------------------------------------------------------------------

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Subsubsection
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Paragraph
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

Content
```

While I try to keep the number of characters fixed to 78 characters as `PEP8` (*Chapter 2, Pythonic Syntax, Common Pitfalls, and Style Guide*) recommends for Python, the number of characters used is mostly arbitrary, but it does have to be at least as long as the text of the header. This allows it to get the following result:

```
Section
=======
```

But not this:

```
Section
====
```

Lists

The reStructuredText format has several styles of lists:

1. Enumerated
2. Bulleted
3. Options
4. Definitions

The simplest forms of lists were already displayed in the introduction section, but it's actually possible to use many different characters, such as letters, Roman numerals, and others, for enumeration. After demonstrating the basic list types, we will continue with the nesting of lists and structures which makes them even more powerful. Care must be taken with the amount of whitespace, as a space too many can cause a structure to be recognized as regular text instead of a structure.

Enumerated list

Enumerated lists are convenient for all sorts of enumerations. The basic premise for enumerated lists is an alphanumeric character followed by a period, a right parenthesis, or parentheses on both sides. Additionally, the `#` character functions as an automatic enumeration. For example:

```
1. With
2. Numbers


a. With
#. letters
```

```
i. Roman
#. numerals

(1) With
(2) Parenthesis
```

The output is perhaps a bit simpler than you would expect. The reason is that it depends on the output format. These were generated with the HTML output format which has no support for parentheses. If you output LaTeX for example, the difference can be made visible. Following is the rendered HTML output:

1. With
2. Numbers

a. With
b. letters

i. Roman
ii. numerals

1. With
2. Parenthesis

Bulleted list

If the order of the list is not relevant and you simply need a list of items without enumeration, then the bulleted list is what you need. To create a simple list using bullets only, the bulleted items need to start with a `*`, `+`, `-`, •, ‣, or ⁃. This list is mostly arbitrary and can be modified by extending Sphinx or Docutils. For example:

```
- dashes
- and more dashes

* asterisk
* stars

+ plus
+ and plus
```

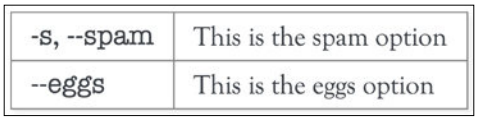
As you can see, with the HTML output again all bullets look identical. When generating documentation as LaTeX (and consecutively, PDF or Postscript), these can differ. Since web-based documentation is by far the most common output format for Sphinx, we default to that output instead. The rendered HTML output is as follows:

Option list

The `option` list is one meant specifically for documenting the command line arguments of a program. The only special thing about the syntax is that the comma-space is recognized as a separator for options.

```
-s, --spam  This is the spam option
--eggs      This is the eggs option
```

Following is the output:

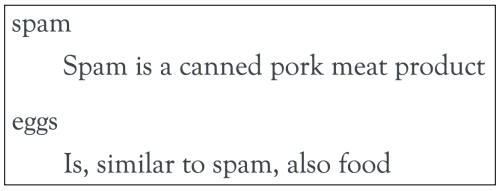


Definition list

The definition list is a bit more obscure than the other types of lists, since the actual structure consists of whitespace only. It's therefore pretty straightforward to use, but not always as easy to identify in a file.

```
spam
    Spam is a canned pork meat product
eggs
    Is, similar to spam, also food
```

Following is the output:

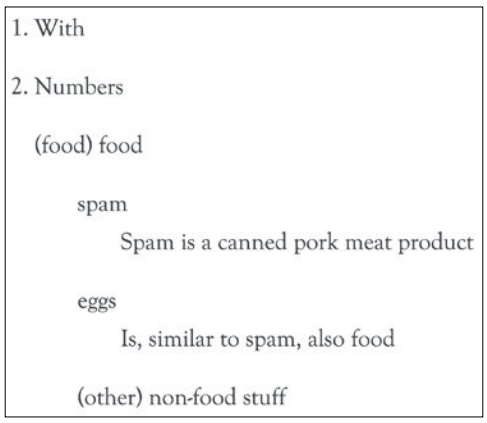


Nested lists

Nesting items is actually not limited to lists and can be done with multiple types of blocks, but the idea is the same. Just be careful to keep the indenting at the correct level. If you don't, it either won't be recognized as a separate level or you will get an error.

```
1. With
2. Numbers

   (food) food

    spam
        Spam is a canned pork meat product

    eggs
        Is, similar to spam, also food

    (other) non-food stuff
```

Following is the output:



Links, references, and labels

There are many types of links supported in reStructuredText, the simplest of which is just a link with the protocol such as `http://python.org`, which will automatically be recognized by most parsers. However, custom labels are also an option by using the interpreted text syntax we saw earlier: `` `Python <http://python.org>`_ ``.

Both of these are nice for simple links, which won't be repeated too often, but generally it's more convenient to attach labels to links so they can be reused and don't clog up the text too much.

For example, refer to the following:

```
The switch to reStructuredText and Sphinx was made with the
`Python 2.6 <https://docs.python.org/whatsnew/2.6.html>`_
release.
```

Now compare it with the following:

```
The switch to reStructuredText and Sphinx was made with the
`python 2.6`_ release.


.. _`Python 2.6`: https://docs.python.org/whatsnew/2.6.html
```

The output is as follows:

Using labels, you can easily have a list of references at a designated location without making the actual text harder to read. These labels can be used for more than external links however; similar to the `GOTO` statements found in older programming languages, you can create labels and refer to them from other parts of the documentation:

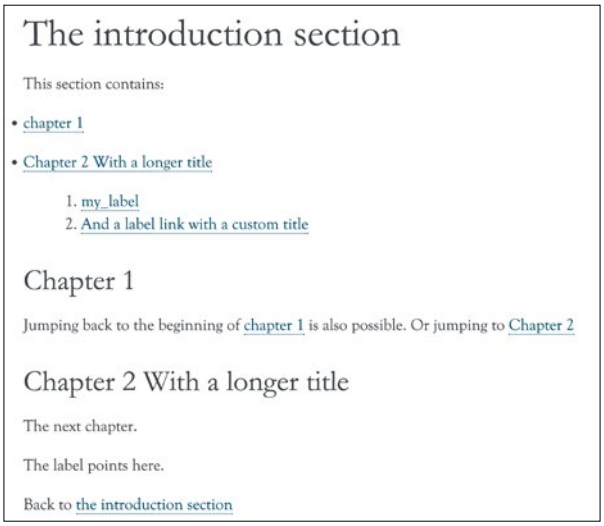
```
.. _label:
```

Within HTML or PDF output, this can be used to create a clickable link from anywhere in the text using the underscore links. Creating a clickable link to the label is as simple as having `label_` in the text. Note that reStructuredText ignores case differences so both uppercase and lowercase links work just fine. Even though it's not likely to make this mistake, having the same label in a single document with only case differences results in an error to make sure duplicates never occur.

The usage of references in conjunction with the headers works in a very natural way; you can just refer to them as you normally would and add an underscore to make it a link:

```
The introduction section
========================================================================

This section contains:

- `chapter 1`_
- :ref:`chapter2`

  1. my_label_

  2. `And a label link with a custom title <my_label>`_

Chapter 1
------------------------------------------------------------------

Jumping back to the beginning of `chapter 1`_ is also possible.
Or jumping to :ref:`Chapter 2 <chapter2>`

.. _chapter2:

Chapter 2 With a longer title
------------------------------------------------------------------

The next chapter.

.. _my_label:

The label points here.

Back to `the introduction section`_
```

The output is as follows:



Images

The image directive looks very similar to the label syntax. They're actually a bit different but the pattern is quite similar. The image directive is just one of the many directives that is supported by reStructuredText. We will see more about that later on when we cover Sphinx and reStructuredText extensions. For the time being, it is enough to know that the directives start with two periods followed by a space, the name of the directive, and two colons:

```
.. name_of_directive::
```

In the case of the image, the directive is called `image` of course:

```
.. image:: python.png
```

Scaled output as the actual image is much larger:

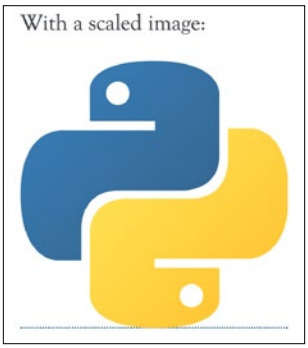
Note the double colon after the directives.

But how about specifying the size and other properties? The image directive has many other options (as do most other directives) which can be used: `http://docutils.sourceforge.net/docs/ref/rst/directives.html#images`, they are mostly fairly obvious however. To specify the width and height or the scale (in percent) of the image:

```
.. image:: python.png
   :width: 150
   :height: 100

.. image:: python.png
   :scale: 10
```

Following is the output:

With a scaled image:



The `scale` option uses the `width` and `height` options if available and falls back to the PIL (Python Imaging Library) or Pillow library to detect the image. If both width/height and PIL/Pillow are not available, the `scale` option will be ignored silently.

In addition to the `image` directive, there is also the `figure` directive. The difference is that `figure` adds a caption to the image. Beyond that, the usage is the same as `image`:

```
.. figure:: python.png
   :scale: 10

   The Python logo
```

The output is as follows:

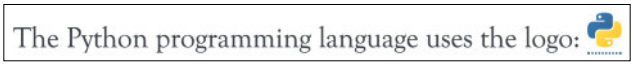
Substitutions

When writing documentation, it often happens that constructs are being repeated, the links have their own labelling system but there are more ways within reStructuredText. The substitution definitions make it possible to shorten directives so they can easily be re-used.

Let's assume we have a logo that we use quite often within a bit of text. Instead of typing the entire `.. image:: <url>` it would be very handy to have a shorthand to make it easier. That's where the substitutions are very useful:

```
.. |python| image:: python.png
   :scale: 1

The Python programming language uses the logo: |python|
```

The output is as follows:



These substitutions can be used with many directives, though they are particularly useful for outputting a variable in many places of a document. For example:

```
.. |author| replace:: Rick van Hattem

This book was written by |author|
```

Following is the output:

Blocks, code, math, comments, and quotes

When writing documentation, a common scenario is the need for blocks that contain different type of content, explanations with mathematical formulas, code examples, and more. The usage of these directives is similar to the image directive. Following is an example of a code block:

```
.. code:: python

   def spam(*args):
       print('spam got args', args)
```

The output is as follows:

```
def spam(*args):
    print('spam got args', args)
```

Or math using LaTeX syntax, the fundamental theorem of calculus:

```
.. math::

    \int_a^b f(x)\,dx = F(b) - F(a)
```

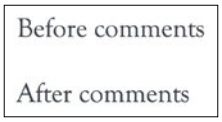
Following is the output:

$$\int_a^b f(x)\,dx = F(b) - F(a)$$

Commenting a bunch of text/commands is easily achieved by using the "empty" directive followed by an indent:

```
Before comments

.. Everything here will be commented

   And this as well

   .. code:: python

      def even_this_code_sample():
          pass  # Will be commented

After comments
```

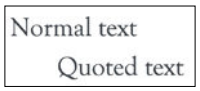
The output is as follows:



The simplest ones are the block quotes. A block quote requires nothing but just a simple bit of indentation.

```
Normal text

    Quoted text
```

The output is as follows:



Conclusion

reStructuredText is both a very simple and a very extensive language; a large portion of the syntax comes naturally when writing plain-text notes. A full guide to all the intricacies, however, could fill a separate book. The previous demonstrations should have given enough of an introduction to do at least 90 percent of the work you will need when documenting your projects. Beyond that, Sphinx will help a lot as we will see in the next sections.

The Sphinx documentation generator

The Sphinx documentation generator was created in 2008 for the Python 2.6 release to replace the old LaTeX documentation for Python. It's a generator that makes it almost trivial to generate documentation for programming projects, but even outside of the programming world it can be easily used. Within programming projects, there is specific support for the following domains (programming languages):

- Python
- C
- C++
- Javascript
- reStructuredText

Outside of these languages, there are extensions available for many other languages such as CoffeeScript, MATLAB, PHP, Ruby Lisp, Go, and Scala. And if you're simply looking for snippet code highlighting, the Pygments highlighter which is used internally supports over 120 languages and is easily extendible for new languages if needed.

The most important advantage of Sphinx is that almost everything can be automatically generated from your source code. So the documentation is always up to date.

Getting started with Sphinx

First of all, we have to make sure we install Sphinx. Even though the Python core documentation is written using Sphinx, it is still a separately maintained project and must be installed separately. Luckily, that's easy enough using pip:

```
pip install sphinx
```

After installing Sphinx, there are two ways of getting started with a project, the `sphinx-quickstart` script and the `sphinx-apidoc` script. If you want to create and customize an entire Sphinx project then `sphinx-quickstart` may be best as it assists you in configuring a fully featured Sphinx project. If you simply want API documentation for an existing project then `sphinx-apidoc` might be better suited since it takes a single command and no further input to create a project.

In the end, both are valid options for creating Sphinx projects and personally I usually end up generating the initial configuration using `sphinx-quickstart` and call the `sphinx-apidoc` command every time I add a Python module to add the new module. Since `sphinx-apidoc` does not overwrite any files by default, it is a safe operation.

Using sphinx-quickstart

The `sphinx-quickstart` script interactively asks you about the most important decisions in your Sphinx project. No need to worry if you've accidently made a typo however. Most of the configuration is stored in the `conf.py` directory so it's easy enough to edit the configuration later in case you still want to enable a certain module.

Usage is easy enough, as a default I would recommend using the following settings. The output uses the following conventions:

- Inline comments start with `#`
- User input lines start with `>`

- Cropped output is indicated with . . . and all questions skipped in between use the default settings

```
# sphinx-quickstart
Welcome to the Sphinx 1.3.3 quickstart utility.

...

Enter the root path for documentation.
> Root path for the documentation [.]: docs

...

The project name will occur in several places in the built documentation.
> Project name: Mastering Python
> Author name(s): Rick van Hattem

# As version you might want to start below 1.0 or add an extra digit
# but I would recommend leaving the default and modify the
# configuration file instead. Just make it import from the Python
# package instead. An example can be found in the numpy-stl package:
# https://github.com/WoLpH/numpy-stl/blob/develop/docs/conf.py
...
> Project version: 1.0
> Project release [1.0]:

...

# Enabling the epub builder can be useful for people using e-readers to
# read the documentation.
Sphinx can also add configuration for epub output:
> Do you want to use the epub builder (y/n) [n]: y

...
# Autodoc is required to document the code, definitely recommended to
# enable
> autodoc: automatically insert docstrings from
  modules (y/n) [n]: y
```

```
# With the doctest feature we can run tests embedded in the
# documentation. This is meant for doctests in the .rst files.
> doctest: automatically test code snippets in
  doctest blocks (y/n) [n]: y

# Intersphinx enables linking between Sphinx documentation sets
# allowing for links to external documentation. After enabling this
# you can make str link to the regular Python documentation about str
# for example.
> intersphinx: link between Sphinx documentation
  of different projects (y/n) [n]: y
...
# Mathjax enables LaTeX style mathematical rendering, not strictly
# needed but very useful for rendering equations.
> mathjax: include math, rendered in the browser
  by MathJax (y/n) [n]: y
...
> viewcode: include links to the source code of
  documented Python objects (y/n) [n]: y

...

Creating file docs/conf.py.
Creating file docs/index.rst.
Creating file docs/Makefile.
Creating file docs/make.bat.

Finished: An initial directory structure has been created.
```

You should now populate your master file `docs/index.rst` and create other documentation source files. Use the Makefile to build the docs, like so:

```
   make builder
where "builder" is one of the supported builders, e.g. html, latex or
linkcheck.
```

After running this, we should have a `docs` directory containing the Sphinx project. Let's see what the command actually created for us:

```
# find docs
docs
docs/_build
docs/_static
docs/_templates
docs/conf.py
docs/index.rst
docs/make.bat
docs/Makefile
```

The `_build`, `_static`, and `_templates` directories are initially empty and can be ignored for now. The `_build` directory is used to output the generated documentation whereas the `_static` directory can be used to easily include custom CSS files and such. The `_templates` directory makes it possible to style the HTML output to your liking as well. Examples of these can be found in the Sphinx Git repository at `https://github.com/sphinx-doc/sphinx/tree/master/sphinx/themes`.

`Makefile` and `make.bat` can be used to generate the documentation output. `Makefile` can be used for any operating system that supports the make utility and `make.bat` is there to support Windows systems out of the box. Now let's look at the `index.rst` source:

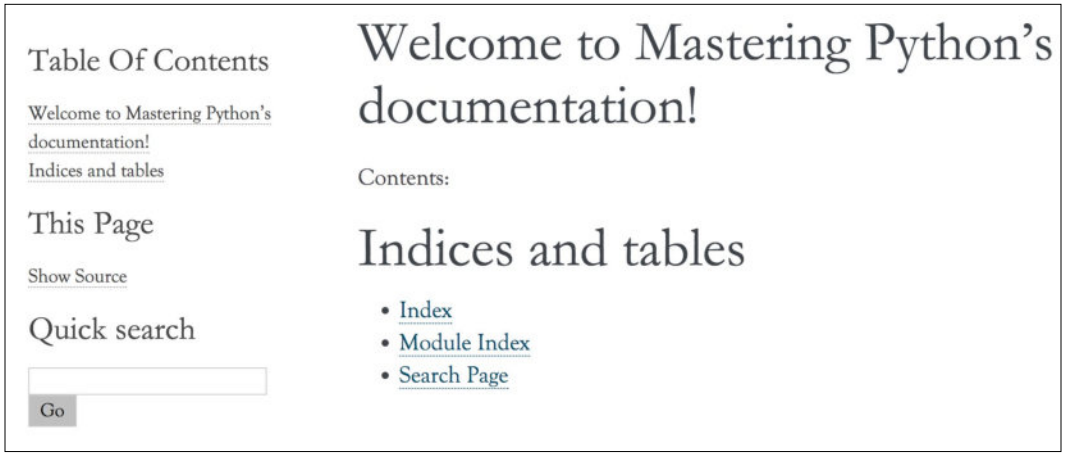
```
Welcome to Mastering Python's documentation!
============================================

Contents:

.. toctree::
   :maxdepth: 2



Indices and tables
==================

* :ref:`genindex`
* :ref:`modindex`
* :ref:`search`
```

We see the document title as expected, followed by `toctree` (table of contents tree; more about that later in this chapter), and the links to the indices and search. `toctree` automatically generates a tree out of the headers of all available documentation pages. The indices and tables are automatically generated Sphinx pages, which are very useful but nothing we need to worry about in terms of settings.

Now it's time to generate the `HTML` output:

```
cd docs
make html
```

The `make` html command generates the documentation for you and the result is placed in `_build/html/`. Just open `index.html` in your browser to see the results. You should have something looking similar to the following now:



With just that single command and by answering a few questions, we now have a documentation project with an index, search, and table of contents on all the pages.

In addition to the HTML output, there are quite a few other formats supported by default, although some require external libraries to actually work:

```
# make
Please use `make <target>' where <target> is one of
  html       to make standalone HTML files
  dirhtml    to make HTML files named index.html in directories
  singlehtml to make a single large HTML file
  pickle     to make pickle files
  json       to make JSON files
```

```
  htmlhelp   to make HTML files and a HTML help project
  qthelp     to make HTML files and a qthelp project
  applehelp  to make an Apple Help Book
  devhelp    to make HTML files and a Devhelp project
  epub       to make an epub
  latex      to make LaTeX files, you can set PAPER=a4 or ...
  latexpdf   to make LaTeX files and run them through pdflatex
  latexpdfja to make LaTeX files and run them through platex/...
  text       to make text files
  man        to make manual pages
  texinfo    to make Texinfo files
  info       to make Texinfo files and run them through makeinfo
  gettext    to make PO message catalogs
  changes    to make an overview of all changed/added/deprecate...
  xml        to make Docutils-native XML files
  pseudoxml  to make pseudoxml-XML files for display purposes
  linkcheck  to check all external links for integrity
  doctest    to run all doctests embedded in the documentation
  coverage   to run coverage check of the documentation
```

Using sphinx-apidoc

The `sphinx-apidoc` command is generally used together with `sphinx-quickstart`. It is possible to generate an entire project with the `--full` parameter but it's generally a better idea to generate the entire project using `sphinx-quickstart` and simply add the API documentation using `sphinx-apidoc`. To properly demonstrate the `sphinx-apidoc` command, we need some Python files, so we'll create two files within a project called `h09`.

The first one is `h09/spam.py` containing a class called `Spam` with some methods:

```
class Spam(object):
    def __init__(self, arg, *args, **kwargs):
        pass

    def regular_method(self, arg):
        pass

    @classmethod
```

```
    def decorated_method(self, arg):
        pass

    def _hidden_method(self):
        pass
```

Next we have `h09/eggs.py` containing a `Eggs` class that inherits `Spam`:

```
import spam


class Eggs(spam.Spam):
    def regular_method(self):
        '''This regular method overrides
        :meth:`spam.Spam.regular_method`
        '''
        pass
```

Now that we have our source files, it's time to generate the actual API documentation:

```
# sphinx-apidoc h09 -o docs
Creating file docs/eggs.rst.
Creating file docs/spam.rst.
Creating file docs/modules.rst.
```

This alone is not enough to include the API in the documentation. It needs to be added to `toctree`. Luckily, that's as simple as adding modules to `toctree` in the `index.rst` file to look something like this:

```
.. toctree::
   :maxdepth: 2

   modules
```

The `toctree` directive is discussed in further detail later in this chapter.

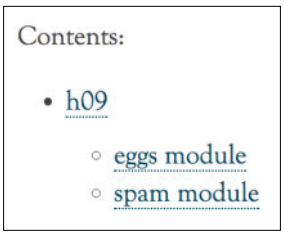
We also have to make sure that the modules can be imported, otherwise Sphinx won't be able to read the Python files. To do that, we simply add the `h09` directory to `sys.path`; this can be put anywhere in the `conf.py` file:

```
import os
sys.path.insert(0, os.path.join(os.path.abspath('..'), 'h09'))
```

Now it's time to generate the documentation again:

```
cd docs
make html
```

Open the `docs/_build/index.html` file again. For the sake of brevity, the repeated parts of the document will be omitted from the screenshots. The cropped output is as follows:

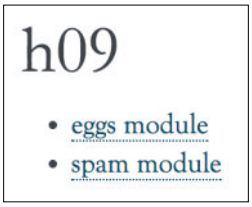


But it actually generated quite a bit more. When running the `sphinx-apidoc` command, it looks at all the Python modules in the specified directory recursively and generates a `rst` file for each of them. After generating all those separate files, it adds all those to a file called `modules.rst` which makes it easy to add them to your documentation.

The `modules.rst` file is really straight to the point; nothing more than a list of modules with the package name as the title really:

```
h09
===

.. toctree::
   :maxdepth: 4

   eggs
   spam
```

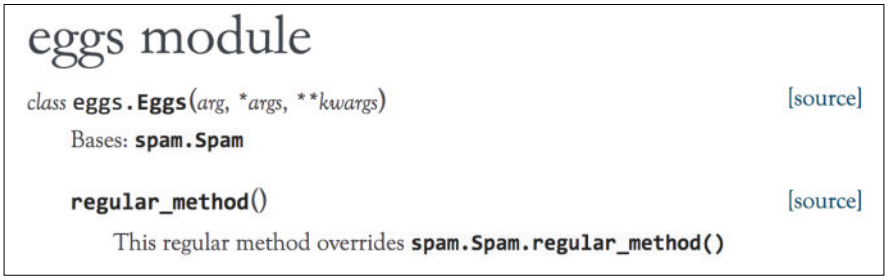
The output is as follows:



`spam.rst` and `eggs.rst` are equally simple, but more important in terms of customization. Within those files it adds the automodule directive which imports the Python module and lists the methods. The methods that are listed can be configured and by default we already get pretty useful output:

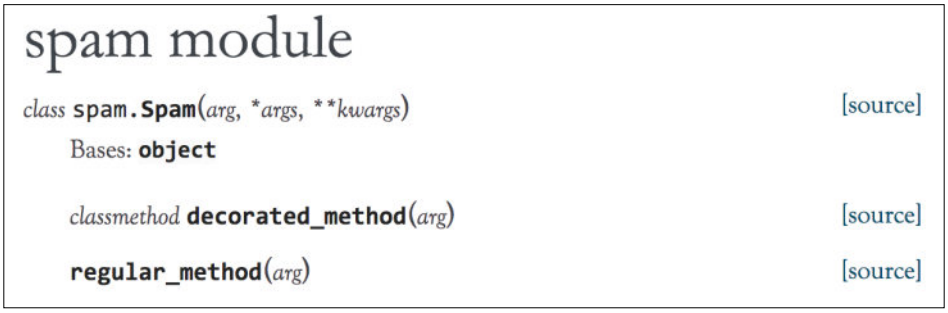
```
eggs module
===========


.. automodule:: eggs
    :members:
    :undoc-members:
    :show-inheritance:
```

Following is the output:



Pretty, isn't it? And all that can be generated from most Python projects with virtually no effort whatsoever. The nice things about this is that the documentation we added to `Eggs.regular_method` is immediately added here, the inherited base (`spam.Spam`) is a clickable link to the `spam.Spam` documentation page, and the `:func:` role makes `spam.Spam.regular_method` immediately clickable as well.

The output for the spam module is similar:



New files won't be added to your docs automatically. It is safe to rerun the `sphinx-apidoc` command to add the new files but it won't update your existing files. Even though the `--force` option can be used to force overwriting the files, within existing files I recommend manually editing them instead. As we will see in the next sections, there are quite a few reasons to manually modify the generated files after.

Sphinx directives

Sphinx adds a few directives on top of the default ones in reStructuredText and an easy API to add new directives yourself. Most of them are generally not that relevant to modify but, as one would expect, Sphinx has pretty good documentation in case you need to know more about them. There are a few very commonly used ones which we will discus however.

The table of contents tree directive (toctree)

This is one of the most important directives in Sphinx; it generates `toctree` (table of contents tree). The `toctree` directive has a couple of options but the most important one is probably `maxdepth` which specifies how deep the tree needs to go. The top level of `toctree` has to be specified manually by specifying the files to be read, but beyond that every level within a document (section, chapter, paragraph, and so on) can be another level in `toctree`, depending on the depth of course. Even though the `maxdepth` option is optional, without it all the available levels will be shown, which is usually more than required. In most cases a `maxdepth` of 2 is a good default value which makes the basic example look like this:

```
.. toctree::
   :maxdepth: 2
```

The items in `toctree` are the `.rst` files in the same directory without the extension. This can include subdirectories, in which case the directories are separated with a `.` (period):

```
.. toctree::
   :maxdepth: 2

   module.a
   module.b
   module.c
```

Another very useful option is the `glob` option. It tells `toctree` to use the `glob` module in Python to automatically add all the documents matching a pattern. By simply adding a directory with a `glob` pattern, you can add all the files in that directory. This makes the `toctree` we had before as simple as:

```
.. toctree::
   :maxdepth: 2
   :glob:

   module.*
```

If for some reason the document title is not as you would have liked, you can easily change the title to something customized:

```
.. toctree::
   :maxdepth: 2

   The A module <module.a>
```

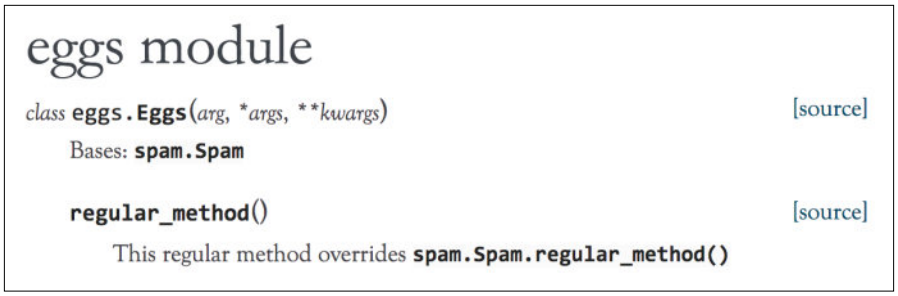
Autodoc, documenting Python modules, classes, and functions

The most powerful feature of Sphinx is the possibility of automatically documenting your modules, classes, and functions. The `sphinx-apidoc` command has already generated some of these for us, so let's use those files for the `Spam` and `Eggs` classes to extend the documentation a bit.

The original result from `sphinx-apidoc` was:

```
eggs module
===========

.. automodule:: eggs
    :members:
    :undoc-members:
    :show-inheritance:
```

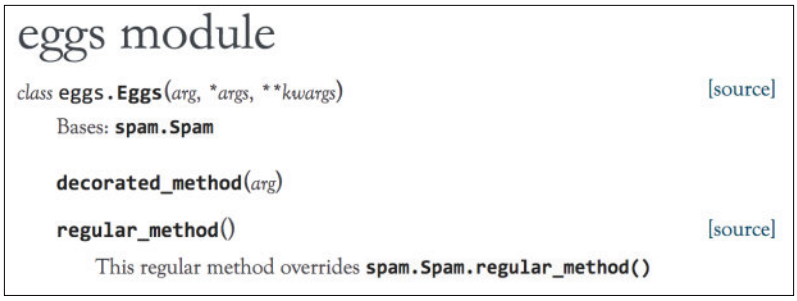
This renders as:



The `Eggs` class has only a single function right now. We can of course click towards the parent class with ease, but in many cases it's useful to see all available functions in the class. So let's add all the functions that are inherited from `Spam` as well:

```
eggs module
===========

.. automodule:: eggs
    :members:
    :undoc-members:
    :show-inheritance:
    :inherited-members:
```

The output is as follows:

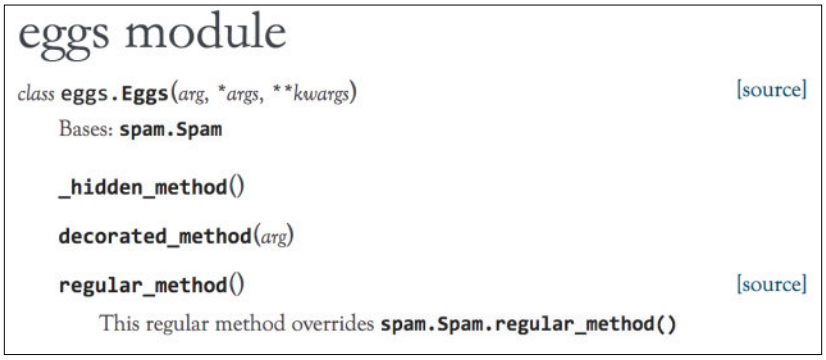


Much more useful already, but we are still missing the hidden method. Let's add the private members as well:

```
eggs module
===========
```

```
.. automodule:: eggs
    :members:
    :undoc-members:
    :show-inheritance:
    :inherited-members:
    :private-members:
```

Following is the output:



Now all the methods are shown, but what about the `members` option? Without the `members` option or the `*-members` options, no functions will be visible anymore.

`show-inheritance` is useful if you want to have the `Bases: ...` section so it is possible to click to the parent class.

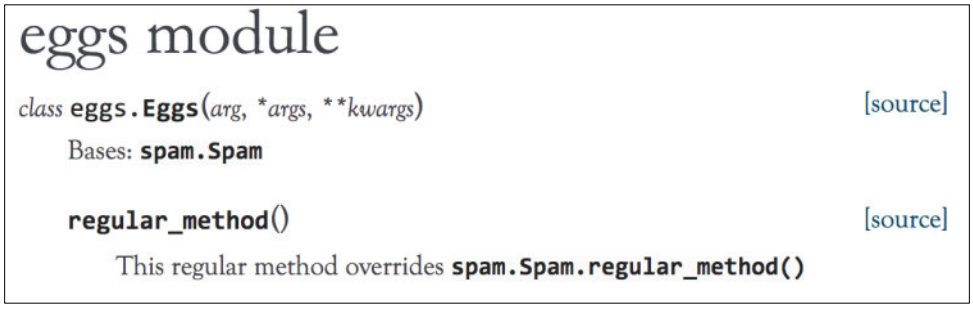
Naturally, it is also possible to create classes manually. While this has little practical use, it does show the internal structure of Python classes within Sphinx.

There is a practical case however, if you are dynamically creating classes then `autodoc` will not always be able to document correctly and some additional help is required. There is more however, while it's generally not that useful as you're doing double work. In some cases, the `autodoc` extension won't be able to correctly identify the members of your class. This is true in case of dynamic class/function generation, for example. For such cases, it can be useful to add some manual documentation to the module/class/function:

```
eggs module
===========

.. automodule:: eggs
    :members:
    :undoc-members:
```

```
    :show-inheritance:

    .. class:: NonExistingClass
       This class doesn't actually exist, but it's in the documentation
now.

       .. method:: non_existing_function()

          And this function does not exist either.
```

Following is the output:



If at all possible, I would avoid this usage though. The biggest benefit of Sphinx is that it can automatically generate a large portion of your docs for you. By manually documenting, you may produce the one thing that's worse than no documentation, that is incorrect documentation. These statements are mainly useful for meta-documentation; documenting how a class might look instead of an actual example.

Sphinx roles

We have seen Sphinx directives, which are separate blocks. Now we will discuss Sphinx roles, which can be used in-line. A role allows you to tell Sphinx how to parse some input. Examples of these roles are links, math, code, and markup. But the most important ones are the roles within the Sphinx domains for referencing other classes, even for external projects. Within Sphinx, the default domain is the Python one so a role such as `:py:meth:` can be used as `:meth:` as well. These roles are really useful to link to different packages, modules, classes, methods, and other objects. The basic usage is simple enough. To link to a class, use the following:

```
Spam: :class:`spam.Spam`
```

The output is:

The same goes for just about any other object, functions, exceptions, attributes, and so on. The Sphinx documentation offers a list of supported objects: `http://sphinx-doc.org/domains.html#cross-referencing-python-objects`.

One of the nicer features of Sphinx is that this is actually possible across projects as well, adding a reference to the `int` object in the standard Python documentation is easily possible using `:obj:`int``. And adding references to your own projects on other sites is fairly trivial as well. Perhaps you remember the `intersphinx` question from the `sphinx-quickstart` script:

```
> intersphinx: link between Sphinx documentation
  of different projects (y/n) [n]: y
```

That's what makes cross-referencing between external Sphinx documentation and your local one possible. With `intersphinx` you can add links between projects with virtually no effort whatsoever. The standard `intersphinx_mapping` in `conf.py` is a bit limited:

```
intersphinx_mapping = {'https://docs.python.org/': None}
```

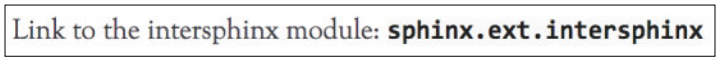
However, it can easily be extended to other documentation sites:

```
intersphinx_mapping = {
    'https://docs.python.org/': None,
    'sphinx': ('http://sphinx-doc.org/', None),
}
```

Now we can easily link to the documentation on the Sphinx homepage:

```
Link to the intersphinx module: :mod:`sphinx.ext.intersphinx`
```

Following is the output:



This links to `http://www.sphinx-doc.org/en/stable/ext/intersphinx.html`.

Documenting code

There are currently three different documentation styles supported by Sphinx: the original Sphinx style and the more recent NumPy and Google styles. The differences between them are mainly in style but it's actually slightly more than that.

The Sphinx style was developed using a bunch of reStructuredText roles, a very effective method but when used a lot it can be detrimental for readability. You can probably tell what the following does but it's not the nicest syntax:

```
:param amount: The amount of eggs to return
:type amount: int
```

The Google style was (as the name suggests) developed by Google. The goal was to have a simple/readable format which works both as in-code documentation and parse able for Sphinx. In my opinion, this comes closer to the original idea of reStructuredText, a format that's very close to how you would document instinctively. This example has the same meaning as the Sphinx style example shown earlier:

```
Args:
    amount (int): The amount of eggs to return
```

The NumPy style was created specifically for the NumPy project. The NumPy project has many functions with a huge amount of documentation and generally a lot of documentation per argument. It is slightly more verbose than the Google format but quite easy to read as well:

```
Parameters
----------
amount : int
    The amount of eggs to return
```

In the future, with the Python 3.5 type hint annotations, at least the argument type part of these syntaxes might become useless. For the time being, Sphinx has no specific support for the annotations yet, so explicit type hinting through the docs must be used. But perhaps we can use the following soon:

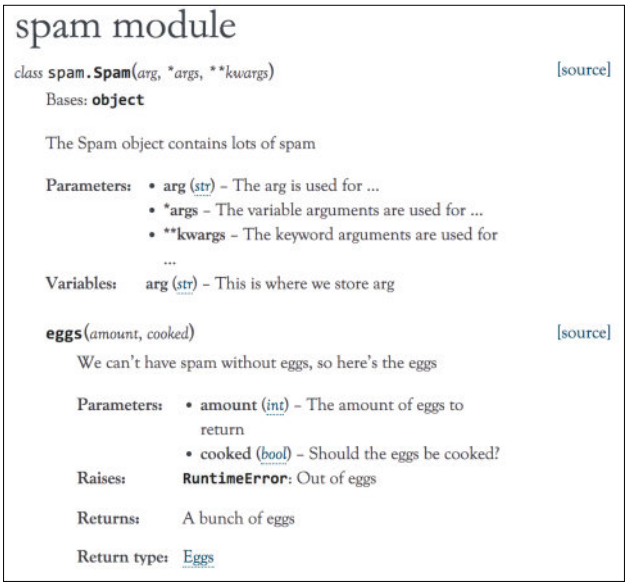
```
def eggs(amount: int):
    pass
```

Documenting a class with the Sphinx style

First of all, let's look at the traditional style, the Sphinx style. While it's easy to understand what all the parameters mean, it's a bit verbose and not all that readable. Nonetheless, it's pretty clear and definitely not a bad style to use:

```
class Spam(object):
    '''
    The Spam object contains lots of spam

    :param arg: The arg is used for ...
    :type arg: str
    :param `*args`: The variable arguments are used for ...
    :param `**kwargs`: The keyword arguments are used for ...
    :ivar arg: This is where we store arg
    :vartype arg: str
    '''
    def __init__(self, arg, *args, **kwargs):
        self.arg = arg

    def eggs(self, amount, cooked):
        '''We can't have spam without eggs, so here's the eggs

        :param amount: The amount of eggs to return
        :type amount: int
        :param bool cooked: Should the eggs be cooked?
        :raises: :class:`RuntimeError`: Out of eggs

        :returns: A bunch of eggs
        :rtype: Eggs
        '''
        pass
```

Following is the output:



spam module

class **spam.Spam**(*arg*, **args*, ***kwargs*) [source]

Bases: **object**

The Spam object contains lots of spam

Parameters:
- **arg** (str) – The arg is used for ...
- ***args** – The variable arguments are used for ...
- ****kwargs** – The keyword arguments are used for ...

Variables: **arg** (str) – This is where we store arg

eggs(*amount*, *cooked*) [source]

We can't have spam without eggs, so here's the eggs

Parameters:
- **amount** (int) – The amount of eggs to return
- **cooked** (*bool*) – Should the eggs be cooked?

Raises: **RuntimeError**: Out of eggs

Returns: A bunch of eggs

Return type: Eggs

This is a very useful output indeed with documented functions, classes, and arguments. And more importantly, the types are documented as well, resulting in a clickable link towards the actual type. An added advantage of specifying the type is that many editors understand the documentation and will provide auto-completion based on the given types.

To explain what's actually happening here, Sphinx has a few roles within the docstrings that offer hints as to what we are documenting.

The `param` role paired with a name sets the documentation for the parameter with that name. The `type` role paired with a name tells Sphinx the data type of the parameter. Both the roles are optional and the parameter simply won't have any added documentation if they are omitted, but the `param` role is always required for any documentation to show. Simply adding the `type` role without the `param` role will result in no output whatsoever, so take note to always pair them.

The `returns` role is similar to the `param` role with regards to documenting. While the `param` role documents a parameter, the `returns` role documents the returned object. They are slightly different however. Opposed to the `param` role, the `returns` role is not dependent of the `rtype` role or vice versa. They both work independently of each other making it possible to use either or both of the roles.

The `rtype`, as you can expect, tells Sphinx (and several editors) what type of object is returned from the function.

Documenting a class with the Google style

The Google style is just a more legible version of the Sphinx style documentation. It doesn't actually support more or less but it's a lot more intuitive to use. The only thing to keep in mind is that it's a fairly recent feature of Sphinx. With the older versions, you were required to install the `sphinxcontrib-napoleon` package. These days it comes bundled with Sphinx but still needs to be enabled through the `conf.py` file. So, depending on the Sphinx version (Napoleon was added in Sphinx 1.3), you will need to add either `sphinx.ext.napoleon` or `sphinxcontrib.napoleon` to the extensions list in `conf.py`.

Once you have everything configured correctly, we can use both the Google and NumPy style. Here's the Google style version of the `Spam` class:

```
class Spam(object):
    '''
    The Spam object contains lots of spam

    Args:
        arg (str): The arg is used for ...
        *args: The variable arguments are used for ...
        **kwargs: The keyword arguments are used for ...

    Attributes:
        arg (str): This is where we store arg,
    '''
    def __init__(self, arg, *args, **kwargs):
        self.arg = arg

    def eggs(self, amount, cooked):
        '''We can't have spam without eggs, so here's the eggs

        Args:
            amount (int): The amount of eggs to return
            cooked (bool): Should the eggs be cooked?

        Raises:
            RuntimeError: Out of eggs
```

```
        Returns:
            Eggs: A bunch of eggs
        '''
        pass
```

This is easier on the eyes than the Sphinx style and has the same amount of possibilities. For longer argument documentation, it's less than convenient though. Just imagine how a multiline description of `amount` would look. That is why the NumPy style was developed, a lot of documentation for its arguments.

Documenting a class with the NumPy style

The NumPy style is meant for having a lot of documentation. Honestly, most people are too lazy for that, so for most projects it would not be a good fit. If you do plan to have extensive documentation of your functions and all their parameters, the NumPy style might be a good option for you. It's a bit more verbose than the Google style but it's very legible, especially with more detailed documentation. Just remember that, similar to the Google style, this requires the Napoleon extension for Sphinx, so make sure you have Sphinx 1.3 or above installed. Following is the NumPy version of the `Spam` class:

```
class Spam(object):
    '''
    The Spam object contains lots of spam

    Parameters
    ----------
    arg : str
        The arg is used for ...
    *args
        The variable arguments are used for ...
    **kwargs
        The keyword arguments are used for ...

    Attributes
    ----------
    arg : str
        This is where we store arg,
    '''
```

```
    def __init__(self, arg, *args, **kwargs):
        self.arg = arg


    def eggs(self, amount, cooked):
        '''We can't have spam without eggs, so here's the eggs

        Parameters
        ----------
        amount : int
            The amount of eggs to return
        cooked : bool
            Should the eggs be cooked?

        Raises
        ------
        RuntimeError
            Out of eggs

        Returns
        -------
        Eggs
            A bunch of eggs
        '''
        pass
```

While the NumPy style definitely isn't bad, it's just very verbose. This example alone is about 1.5 times as long as the alternatives. So, for longer and more detailed documentation it's a very good choice, but if you're planning to have short documentation anyhow, just use the Google style instead.

Which style to choose

For most projects, the Google style is the best choice since it is readable but not too verbose. If you are planning to use large amounts of documentation per parameter then the NumPy style might be a good option as well.

The only reason to choose the Sphinx style is legacy. Even though the Google style might be more legible, consistency is more important.

Summary

Documentation can help greatly in a project's popularity and bad documentation can kill productivity. I think there are few aspects of a library that have more impact on the usage by third parties than documentation. Thus in many cases, documentation is a more important factor in deciding the usage of a project than the actual code quality. That's why it is very important to always try to have some documentation available.

With Sphinx it is actually easy to generate documentation. With just a few minutes of your time, you can have a fully functioning website with documentation available, or a PDF, or ePub, or one of the many other output formats. There really is no excuse for having no documentation anymore. And even if you don't use the documentation that much yourself, offering type hints to your editor can help a lot in productivity as well. Making your editor smarter should always help in productivity. I for one have added type hints to several projects simply to increase my productivity.

The next chapter will explain how code can be tested in Python and some part of the documentation will return there. Using `doctest`, it is possible to have example code, documentation, and tests in one.

10
Testing and Logging – Preparing for Bugs

When programming, most developers plan a bit and immediately continue writing code. After all, we all expect to write bug-free code! Unfortunately, we don't. At some point, an incorrect assumption, a misinterpretation, or just a silly mistake is bound to happen. Debugging (covered in *Chapter 11*, *Debugging – Solving the Bugs*) will always be required at some point, but there are several methods that you can use to prevent bugs or, at the very least, make it much easier to solve them when they do occur.

To prevent bugs from occurring in the first place, test-driven development or, at the very least, functional/regression/unit tests are very useful. The standard Python installation alone offers several options such as the `doctest`, `unittest`, and `test` modules. The `doctest` module allows you to combine tests with example documentation. The `unittest` module allows you to easily write regression tests. The `test` module is meant for internal usage only, so unless you are planning to modify the Python core, you probably won't need this one.

The test modules we will discuss in this chapter are:

- `doctest`
- `py.test` (and why it's more convenient than `unittest`)
- `unittest.mock`

The `py.test` module has roughly the same purpose as the `unittest` module, but it's much more convenient to use and has a few extra options.

After learning how to avoid the bugs, it's time to take a look at logging so that we can inspect what is happening in our program and why. The logging module in Python is highly configurable and can be adjusted for just about any use case. If you've ever written Java code, you should feel right at home with the `logging` module, as its design is largely based on the `log4j` module and is very similar in both implementation and naming. The latter makes it a bit of an odd module in Python as well, as it is one of the few modules that do not follow the `pep8` naming standards.

This chapter will explain the following topics:

- Combining documentation with tests using `doctest`
- Regression and unit tests using `py.test` and `unittest`
- Testing with fake objects using `unittest.mock`
- Using the `logging` module effectively
- Combining `logging` and `py.test`

Using examples as tests with doctest

The `doctest` module is one of the most useful modules within Python. It allows you to combine documenting your code with tests to make sure that it keeps working as it is supposed to.

A simple doctest example

Let's start with a quick example: a function that squares the input. The following example is a fully functional command-line application, containing not only code but also functioning tests. The first few tests cover how the function is supposed to behave when executing normally, followed by a few tests to demonstrate the expected errors:

```
def square(n):
    '''
    Returns the input number, squared


    >>> square(0)
    0
    >>> square(1)
    1
    >>> square(2)
    4
```

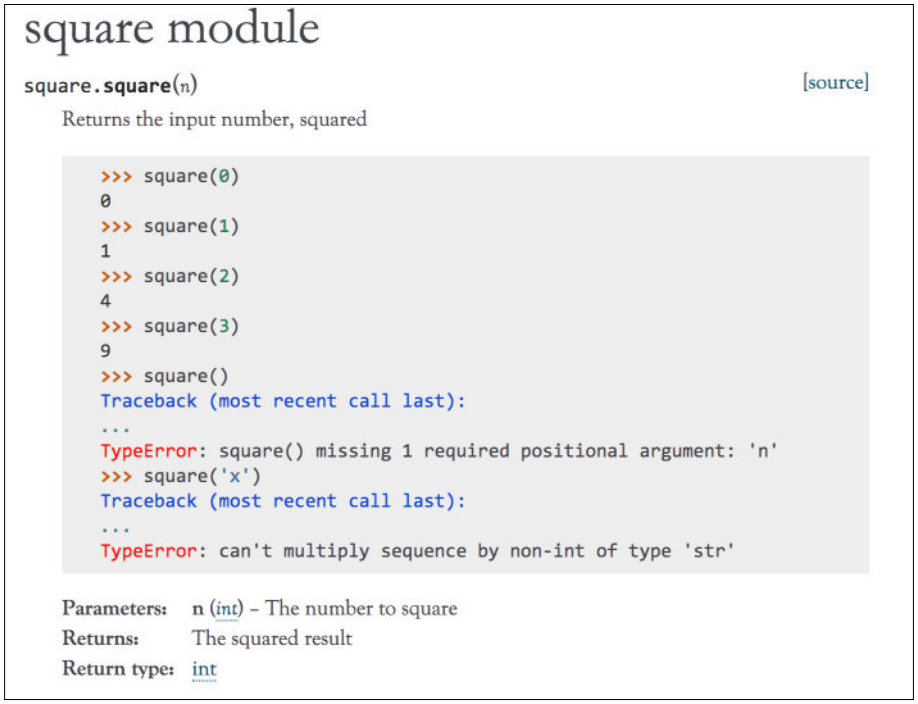
```
    >>> square(3)
    9
    >>> square()
    Traceback (most recent call last):
    ...
    TypeError: square() missing 1 required positional argument: 'n'
    >>> square('x')
    Traceback (most recent call last):
    ...
    TypeError: can't multiply sequence by non-int of type 'str'

    Args:
        n (int): The number to square

    Returns:
        int: The squared result
    '''
    return n * n

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

It can be executed as any Python script, but the regular command won't give any output as all tests are successful. The `doctest.testmod` function takes verbosity parameters, luckily:

```
# python square.py -v
Trying:
    square(0)
Expecting:
    0
ok
Trying:
    square(1)
Expecting:
    1
```

```
ok
Trying:
    square(2)
Expecting:
    4
ok
Trying:
    square(3)
Expecting:
    9
ok
Trying:
    square()
Expecting:
    Traceback (most recent call last):
    ...
    TypeError: square() missing 1 required positional argument: 'n'
ok
Trying:
    square('x')
Expecting:
    Traceback (most recent call last):
    ...
    TypeError: can't multiply sequence by non-int of type 'str'
ok
1 items had no tests:
    __main__
1 items passed all tests:
   6 tests in __main__.square
6 tests in 2 items.
6 passed and 0 failed.
Test passed.
```

Additionally, since it uses the Google syntax (as discussed in *Chapter 9*, *Documentation – How to Use Sphinx and reStructuredText*, the documentation chapter), we can generate pretty documentation using Sphinx:



square module

square.square(n) [source]

Returns the input number, squared

```
>>> square(0)
0
>>> square(1)
1
>>> square(2)
4
>>> square(3)
9
>>> square()
Traceback (most recent call last):
...
TypeError: square() missing 1 required positional argument: 'n'
>>> square('x')
Traceback (most recent call last):
...
TypeError: can't multiply sequence by non-int of type 'str'
```

Parameters: n (*int*) – The number to square
Returns: The squared result
Return type: int

However, the code is not always correct, of course. What would happen if we modify the code so that the tests do not pass anymore?

This time, instead of `n * n`, we use `n ** 2`. Both square a number right? So the results must be identical. Right? These are the types of assumptions that create bugs, and the types of assumptions that are trivial to catch using a few basic tests:

```
def square(n):
    '''
    Returns the input number, squared

    >>> square(0)
    0
    >>> square(1)
    1
    >>> square(2)
```

```
    4
    >>> square(3)
    9
    >>> square()
    Traceback (most recent call last):
    ...
    TypeError: square() missing 1 required positional argument: 'n'
    >>> square('x')
    Traceback (most recent call last):
    ...
    TypeError: can't multiply sequence by non-int of type 'str'

    Args:
        n (int): The number to square

    Returns:
        int: The squared result
    '''
    return n ** 2

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

So let's execute the test again and see what happens this time. For brevity, we will skip the verbosity flag this time:

```
# python square.py
**********************************************************************
File "square.py", line 17, in __main__.square
Failed example:
    square('x')
Expected:
    Traceback (most recent call last):
    ...
    TypeError: can't multiply sequence by non-int of type 'str'
Got:
```

```
    Traceback (most recent call last):
      File "doctest.py", line 1320, in __run
        compileflags, 1), test.globs)
      File "<doctest __main__.square[5]>", line 1, in <module>
        square('x')
      File "square.py", line 28, in square
        return n ** 2
    TypeError: unsupported operand type(s) for ** or pow(): 'str' and
'int'
**********************************************************************
1 items had failures:
   1 of   6 in __main__.square
***Test Failed*** 1 failures.
```

The only modification we made to the code was replacing `n * n` with `n ** 2`, which translates to the power function. Since multiplication is not the same as taking the power of a number, the results are slightly different but similar enough in practice that most programmers wouldn't notice the difference.

The only difference caused by the code change was that we now have a different exception—an innocent mistake, only breaking the tests in this case. But it shows how useful these tests are. When rewriting code, an incorrect assumption is easily made, and that is where tests are most useful—knowing you are breaking code as soon as you break it instead of finding out months later.

Writing doctests

Perhaps, you have noticed from the preceding examples that the syntax is very similar to the regular Python console, and that is exactly the point. The `doctest` input is nothing more than the output of a regular Python shell session. This is what makes testing with this module so intuitive; simply write the code in the Python console and copy the output into a docstring to get tests. Here is an example:

```
# python
>>> from square import square
>>> square(5)
25
>>> square()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: square() missing 1 required positional argument: 'n'
```

That's why this is probably the easiest way to test code. With almost no effort, you can check whether your code is working as you would expect it, add tests, and add documentation at the same time. Simply copy the output from the interpreter to your function or class documentation and you have functioning doctests.

Testing with pure documentation

The docstrings in functions, classes, and modules are usually the most obvious way to add doctests to your code, but they are not the only way. The Sphinx documentation, as we discussed in the previous chapter, also supports the `doctest` module. You might remember that when creating the Sphinx project, we enabled the `doctest` module:

```
> doctest: automatically test code snippets in doctest blocks (y/n) [n]:y
```

This flag enables the `sphinx.ext.doctest` extension in Sphinx, which tells Sphinx to run those tests as well. Since not all the examples in the code are useful, let's see whether we can split them between the ones that are actually useful and the ones that are only relevant for documentation. Moreover, to see the results, we will add an error to the documentation:

square.py

```
def square(n):
    '''
    Returns the input number, squared

    >>> square(2)
    4

    Args:
        n (int): The number to square

    Returns:
        int: The squared result
    '''
    return n * n

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

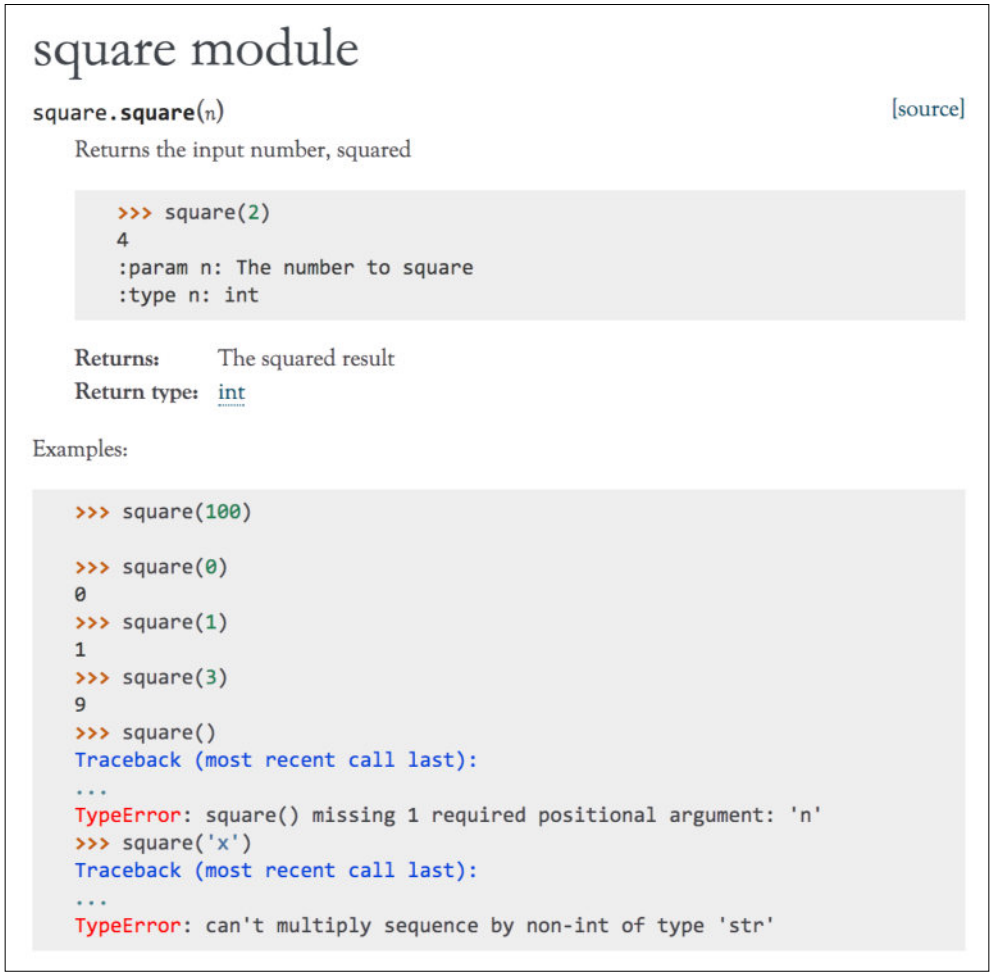
square.rst

```
square module
=============

.. automodule:: square
    :members:
    :undoc-members:
    :show-inheritance:

Examples:

.. testsetup::

    from square import square

.. doctest::

    >>> square(100)

    >>> square(0)
    0
    >>> square(1)
    1
    >>> square(3)
    9
    >>> square()
    Traceback (most recent call last):
    ...
    TypeError: square() missing 1 required positional argument: 'n'
    >>> square('x')
    Traceback (most recent call last):
    ...
    TypeError: can't multiply sequence by non-int of type 'str'
```

Now, it's time to execute the tests. In the case of Sphinx, there is a specific command for this:

```
# make doctest
sphinx-build -b doctest -d _build/doctrees . _build/doctest
Running Sphinx v1.3.3
loading translations [en]... done
loading pickled environment... done
building [mo]: targets for 0 po files that are out of date
building [doctest]: targets for 3 source files that are out of date
updating environment: 0 added, 0 changed, 0 removed
looking for now-outdated files... none found
running tests...

Document: square
----------------
**********************************************************************
File "square.rst", line 16, in default
Failed example:
    square(100)
Expected nothing
Got:
    10000
**********************************************************************
1 items had failures:
   1 of   7 in default
7 tests in 1 items.
6 passed and 1 failed.
***Test Failed*** 1 failures.

Doctest summary
===============
    7 tests
    1 failure in tests
    0 failures in setup code
    0 failures in cleanup code
build finished with problems.
make: *** [doctest] Error 1
```

As expected, we are getting an error for the incomplete `doctest`, but beyond that, all tests executed correctly. To make sure that the tests know what `square` is, we had to add the `testsetup` directive, and this still generates a pretty output:



square module

square.square(n) [source]

Returns the input number, squared

```
>>> square(2)
4
:param n: The number to square
:type n: int
```

Returns: The squared result

Return type: int

Examples:

```
>>> square(100)

>>> square(0)
0
>>> square(1)
1
>>> square(3)
9
>>> square()
Traceback (most recent call last):
...
TypeError: square() missing 1 required positional argument: 'n'
>>> square('x')
Traceback (most recent call last):
...
TypeError: can't multiply sequence by non-int of type 'str'
```

The doctest flags

The `doctest` module features several option flags. They affect how `doctest` processes the tests. These option flags can be passed globally using your test suite, through command-line parameters while running the tests, and through inline commands. For this book, I have globally enabled the following option flags through a `pytest.ini` file (we will cover more about `py.test` later in this chapter):

```
doctest_optionflags = ELLIPSIS NORMALIZE_WHITESPACE
```

Without these option flags, some of the examples in this book will not function properly. This is because they have to be reformatted to fit. The next few paragraphs will cover the following option flags:

- `DONT_ACCEPT_TRUE_FOR_1`
- `NORMALIZE_WHITESPACE`
- `ELLIPSIS`

There are several other option flags available with varying degrees of usefulness, but these are better left to the Python documentation:

`https://docs.python.org/3/library/doctest.html#option-flags`

True and False versus 1 and 0

Having `True` evaluating to `1` and `False` evaluating to `0` is useful in most cases, but it can give unexpected results. To demonstrate the difference, we have these lines:

```
'''
>>> False
0
>>> True
1
>>> False  # doctest: +DONT_ACCEPT_TRUE_FOR_1
0
>>> True  # doctest: +DONT_ACCEPT_TRUE_FOR_1
1
'''


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Here are the results of the `DONT_ACCEPT_TRUE_FOR_1` flag:

```
# python test.py
**********************************************************************
File "test.py", line 6, in __main__
Failed example:
    False  # doctest: +DONT_ACCEPT_TRUE_FOR_1
```

```
Expected:
    0
Got:
    False
**********************************************************************
File "test.py", line 8, in __main__
Failed example:
    True  # doctest: +DONT_ACCEPT_TRUE_FOR_1
Expected:
    1
Got:
    True
**********************************************************************
1 items had failures:
   2 of   4 in __main__
***Test Failed*** 2 failures.
```

As you can see, the `DONT_ACCEPT_TRUE_FOR_1` flag makes `doctest` reject `1` as a valid response for `True` as well as `0` for `False`.

Normalizing whitespace

Since doctests are used for both documentation and test purposes, it is pretty much a requirement to keep them readable. Without normalizing whitespace, this can be tricky, however. Consider the following example:

```
>>> [list(range(5)) for i in range(5)]
[[0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0,
1, 2, 3, 4]]
```

While not all that bad, this output isn't the best for readability. With whitespace normalizing, here is what we can do instead:

```
>>> [list(range(5)) for i in range(5)]  # doctest: +NORMALIZE_WHITESPACE
[[0, 1, 2, 3, 4],
 [0, 1, 2, 3, 4],
 [0, 1, 2, 3, 4],
 [0, 1, 2, 3, 4],
 [0, 1, 2, 3, 4]]
```

Formatting the output in this manner is both more readable and convenient for keeping your line length less.

Ellipsis

The `ELLIPSIS` flag is very useful but also a bit dangerous, as it can easily lead to incorrect matches. It makes `...` match any substring, which is very useful for exceptions but dangerous in other cases:

```
>>> {10: 'a', 20: 'b'}  # doctest: +ELLIPSIS
{...}
>>> [True, 1, 'a']  # doctest: +ELLIPSIS
[...]
>>> True,  # doctest: +ELLIPSIS
(...)
>>> [1, 2, 3, 4]  # doctest: +ELLIPSIS
[1, ..., 4]
>>> [1, 0, 0, 0, 0, 0, 4]  # doctest: +ELLIPSIS
[1, ..., 4]
```

These cases are not too useful in real scenarios, but they demonstrate how the `ELLIPSIS` option flag functions. They also indicate the danger. Both `[1, 2, 3, 4]` and `[1, 0, ... , 4]` match the `[1, ..., 4]` test, which is probably unintentional, so be very careful while using `ELLIPSIS`.

A more useful case is when documenting class instances:

```
>>> class Spam(object):
...     pass
>>> Spam()  # doctest: +ELLIPSIS
<__main__.Spam object at 0x...>
```

Without the `ELLIPSIS` flag, the memory address (the `0x...` part) would never be what you expect. Let's demonstrate an actual run in a normal CPython instance:

```
Failed example:
    Spam()
Expected:
    <__main__.Spam object at 0x...>
Got:
    <__main__.Spam object at 0x10d9ad160>
```

Doctest quirks

The three option flags discussed earlier take care of quite a few quirks found in doctests, but there are several more cases that require care. In these cases, you just need to be a bit careful and work around the limitations of the `doctest` module. The `doctest` module effectively uses the representation string, and those are not always consistent.

The most important cases are floating-point inaccuracies, dictionaries, and random values, such as timers. The following example will fail most of the time because certain types in Python have no consistent ordering and depend on external variables:

```
>>> dict.fromkeys('spam')
{'s': None, 'p': None, 'a': None, 'm': None}
>>> 1./7.
0.14285714285714285


>>> import time
>>> time.time() - time.time()
-9.5367431640625e-07
```

All the problems have several possible solutions, which differ mostly in style and your personal preference.

Testing dictionaries

The problem with dictionaries is that they are internally implemented as hash tables, resulting in an effectively random representation order. Since the `doctest` system requires a representation string that is identical in meaning (save for certain `doctest` flags, of course) to the `docstring`, this does not work. Naturally, there are several workaround options available and all have some advantages and disadvantages.

The first is using the `pprint` library to format it in a pretty way:

```
>>> import pprint
>>> data = dict.fromkeys('spam')
>>> pprint.pprint(data)
{'a': None, 'm': None, 'p': None, 's': None}
```

Since the `pprint` library always sorts the items before outputting, this solves the problem with random representation orders. However, it does require an extra import and function call, which some people prefer to avoid.

Another option is manual sorting of the items:

```
>>> data = dict.fromkeys('spam')
>>> sorted(data.items())
[('a', None), ('m', None), ('p', None), ('s', None)]
```

The downside here is that it is not visible from the output that `data` is a dictionary, which makes the output less readable.

Lastly, comparing the `dict` with a different `dict` comprised of the same elements works as well:

```
>>> data = dict.fromkeys('spam')
>>> data == {'a': None, 'm': None, 'p': None, 's': None}
True
```

A perfectly okay solution, of course! But `True` is not really the clearest output, especially if the comparison doesn't work:

```
Failed example:
    data == {'a': None, 'm': None, 'p': None}
Expected:
    True
Got:
    False
```

On the other hand, the other options presented previously show both the expected value and the returned value correctly:

```
Failed example:
    sorted(data.items())
Expected:
    [('a', None), ('m', None), ('p', None)]
Got:
    [('a', None), ('m', None), ('p', None), ('s', None)]


Failed example:
    pprint.pprint(data)
Expected:
    {'a': None, 'm': None, 'p': None}
Got:
    {'a': None, 'm': None, 'p': None, 's': None}
```

Personally, out of the solutions presented, I would recommend using `pprint`, as I find it the most readable solution, but all the solutions have some merits to them.

Testing floating-point numbers

For the same reason as a floating-point comparison can be problematic (that is, `1/3 == 0.333`), a representation string comparison is also problematic. The easiest solution is to simply add some rounding/clipping to your code, but the `ELLIPSIS` flag is also an option here. Here is a list of several solutions:

```
>>> 1/3  # doctest: +ELLIPSIS
0.333...
>>> '%.3f' % (1/3)
'0.333'
>>> '{:.3f}'.format(1/3)
'0.333'
>>> round(1/3, 3)
0.333
>>> 0.333 < 1/3 < 0.334
True
```

When the `ELLIPSIS` option flag is enabled globally anyhow, that would be the most obvious solution. In other cases, I recommend one of the alternative solutions.

Times and durations

For timings, the problems that you will encounter are quite similar to the floating-point issues. When measuring the duration execution time of a code snippet, there will always be some variation present. That's why the most stable solution for tests, including time, is limiting the precision, although even that is no guarantee. Regardless, the simplest solution checks whether the delta between the two times is smaller than a certain number, as follows:

```
>>> import time
>>> a = time.time()
>>> b = time.time()
>>> (b - a) < 0.01
True
```

For the `timedelta` objects, however, it's slightly more complicated. Yet, this is where the `ELLIPSIS` flag definitely comes in handy again:

```
>>> import datetime
>>> a = datetime.datetime.now()
>>> b = datetime.datetime.now()
>>> str(b - a)  # doctest: +ELLIPSIS
'0:00:00.000...
```

The alternative to the `ELLIPSIS` option flag would be comparing the days, hours, minutes, and microseconds in `timedelta` separately.

In a later paragraph, we will see a completely stable solution for problems like these using mock objects. For doctests, however, that is generally overkill.

Testing with py.test

The `py.test` tool makes it very easy to write tests and run them. There are a few other options such as `nose` and the bundled `unittest` module available, but the `py.test` library offers a very good combination of usability and active development. In the past, I was an avid `nose` user but have since switched to `py.test` as it tends to be easier to use and has better community support, in my experience at least. Regardless, `nose` is still a good choice, and if you're already using it, there is little reason to switch and rewrite all of your tests. When writing tests for a new project, however, `py.test` can be much more convenient.

Now, we will run the doctests from the previously discussed `square.py` file using `py.test`.

First, start by installing `py.test`, of course:

```
pip install pytest
```

Now you can do a test run, so let's give the doctests we have in `square.py` a try:

```
# py.test --doctest-modules -v square.py
========================= test session starts ==========================
platform darwin -- Python 3.5.1, pytest-2.8.2, py-1.4.30, pluggy-0.3.1 --
python3.5
cachedir: .cache
rootdir: code, inifile: pytest.ini
collected 1 items
```

```
square.py::square.square PASSED

==================== 1 passed in 0.02 seconds ======================
```

The difference between the unittest and py.test output

We have the doctests in `square.py`. Let's create a new class called `cube` and create a proper set of tests outside of the code.

First of all, we have the code of `cube.py`, similar to `square.py` but minus the doctests, since we don't need them anymore:

```
def cube(n):
    '''
    Returns the input number, cubed

    Args:
        n (int): The number to cube

    Returns:
        int: The cubed result
    '''
    return n ** 3
```

Now let's start with the `unittest` example, `test_cube.py`:

```
import cube
import unittest


class TestCube(unittest.TestCase):
    def test_0(self):
        self.assertEqual(cube.cube(0), 0)

    def test_1(self):
        self.assertEqual(cube.cube(1), 1)

    def test_2(self):
```

```
        self.assertEqual(cube.cube(2), 8)

    def test_3(self):
        self.assertEqual(cube.cube(3), 27)

    def test_no_arguments(self):
        with self.assertRaises(TypeError):
            cube.cube()

    def test_exception_str(self):
        with self.assertRaises(TypeError):
            cube.cube('x')

if __name__ == '__main__':
    unittest.main()
```

This can be executed by executing the file itself:

```
# python test_cube.py -v
test_0 (__main__.TestCube) ... ok
test_1 (__main__.TestCube) ... ok
test_2 (__main__.TestCube) ... ok
test_3 (__main__.TestCube) ... ok
test_exception_str (__main__.TestCube) ... ok
test_no_arguments (__main__.TestCube) ... ok

----------------------------------------------------------------------
Ran 6 tests in 0.001s

OK
```

Alternatively, it can be done through the module:

```
# python -m unittest -v test_cube.py
test_0 (test_cube.TestCube) ... ok
test_1 (test_cube.TestCube) ... ok
test_2 (test_cube.TestCube) ... ok
test_3 (test_cube.TestCube) ... ok
test_exception_str (test_cube.TestCube) ... ok
```

```
test_no_arguments (test_cube.TestCube) ... ok

----------------------------------------------------------------------
Ran 6 tests in 0.001s

OK
```

This one is through `py.test`:

```
# py.test -v test_cube.py
====================== test session starts ======================
platform darwin -- Python 3.5.1, pytest-2.8.5, py-1.4.31, pluggy-0.3.1 --
python3.5
cachedir: ../.cache
rootdir: code, inifile: pytest.ini
collected 6 items

test_cube.py::TestCube::test_0 PASSED
test_cube.py::TestCube::test_1 PASSED
test_cube.py::TestCube::test_2 PASSED
test_cube.py::TestCube::test_3 PASSED
test_cube.py::TestCube::test_exception_str PASSED
test_cube.py::TestCube::test_no_arguments PASSED

=================== 6 passed in 0.02 seconds ====================
```

We even have `nose`:

```
# nosetests -v test_cube.py
test_0 (test_cube.TestCube) ... ok
test_1 (test_cube.TestCube) ... ok
test_2 (test_cube.TestCube) ... ok
test_3 (test_cube.TestCube) ... ok
test_exception_str (test_cube.TestCube) ... ok
test_no_arguments (test_cube.TestCube) ... ok

----------------------------------------------------------------------
Ran 6 tests in 0.001s

OK
```

As long as all the results are successful, the differences between `unittest` and `py.test` are slim. In the case of `unittest` and `nose`, the results are identical. This time around, however, we are going to break the code to show the difference when it actually matters. Instead of the `cube` code, we will add the `square` code. So returning `n ** 2` instead of `n ** 3` from `square`.

First of all, we have the regular `unittest` output:

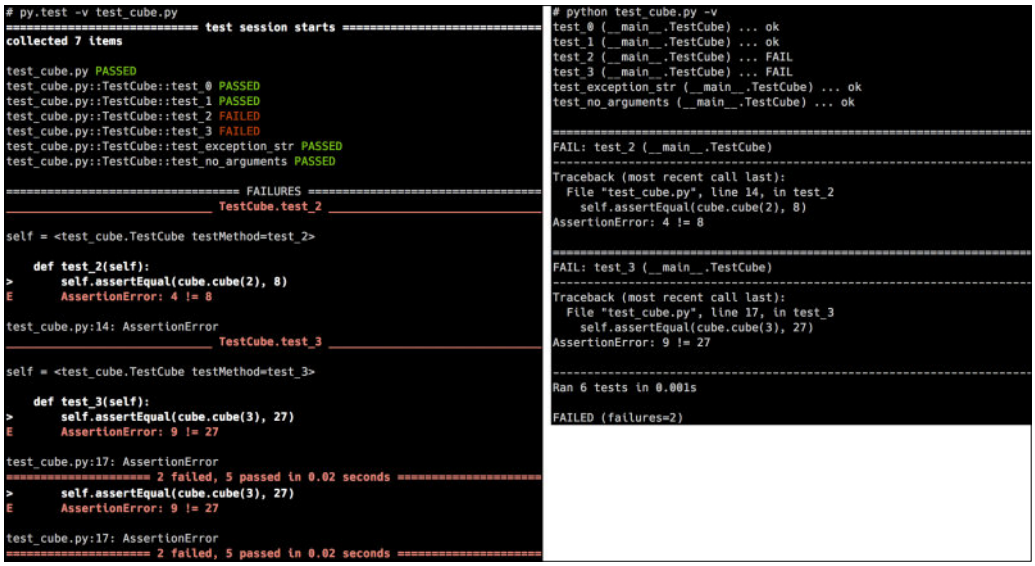
```
# python test_cube.py -v
test_0 (__main__.TestCube) ... ok
test_1 (__main__.TestCube) ... ok
test_2 (__main__.TestCube) ... FAIL
test_3 (__main__.TestCube) ... FAIL
test_exception_str (__main__.TestCube) ... ok
test_no_arguments (__main__.TestCube) ... ok

======================================================================
FAIL: test_2 (__main__.TestCube)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_cube.py", line 13, in test_2
    self.assertEqual(cube.cube(2), 8)
AssertionError: 4 != 8

======================================================================
FAIL: test_3 (__main__.TestCube)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_cube.py", line 16, in test_3
    self.assertEqual(cube.cube(3), 27)
AssertionError: 9 != 27

----------------------------------------------------------------------
Ran 6 tests in 0.001s

FAILED (failures=2)
```

Not all that bad, as per each test returns a nice stack trace that includes the values and everything. Yet, we can observe a small difference here when compared with the `py.test` run:

```
# py.test -v test_cube.py
======================== test session starts ========================
platform darwin -- Python 3.5.1, pytest-2.8.5, py-1.4.31, pluggy-0.3.1 --
python3.5
cachedir: ../.cache
rootdir: code, inifile: pytest.ini
collected 6 items

test_cube.py::TestCube::test_0 PASSED
test_cube.py::TestCube::test_1 PASSED
test_cube.py::TestCube::test_2 FAILED
test_cube.py::TestCube::test_3 FAILED
test_cube.py::TestCube::test_exception_str PASSED
test_cube.py::TestCube::test_no_arguments PASSED

============================== FAILURES ==============================
_____________________________ TestCube.test_2 ______________________________

self = <test_cube.TestCube testMethod=test_2>

    def test_2(self):
>       self.assertEqual(cube.cube(2), 8)
E       AssertionError: 4 != 8

test_cube.py:13: AssertionError
_____________________________ TestCube.test_3 ______________________________

self = <test_cube.TestCube testMethod=test_3>

    def test_3(self):
>       self.assertEqual(cube.cube(3), 27)
E       AssertionError: 9 != 27

test_cube.py:16: AssertionError
================= 2 failed, 4 passed in 0.03 seconds ================
```

In small cases such as these, the difference is not all that apparent, but when testing complicated code with large stack traces, it becomes even more useful. However, for me personally, seeing the surrounding test code is a big advantage. In the example that was just discussed, the `self.assertEqual(...)` line shows the entire test, but in many other cases, you will need more information. The difference between the regular `unittest` module and the `py.test` module is that you can see the entire function with all of the code and the output. Later in this chapter, we will see how powerful this can be when writing more advanced tests.

To truly appreciate the `py.test` output, we need to enable colors as well. The colors depend on your local color schemes, of course, but it's useful to see them side by side at least once, as shown here:



Perhaps you are wondering now, "Is that all?" The only difference between `py.test` and `unittest` is a bit of color and a slightly different output? Well, far from it, there are many other differences, but this alone is enough reason to give it a try.

The difference between unittest and py.test tests

The improved output does help a bit, but the combination of improved output and a much easier way to write tests is what makes `py.test` so useful. There are quite a few methods for making the tests simpler and more legible, and in many cases, you can choose which you prefer. As always, readability counts, so choose wisely and try not to over-engineer the solutions.

Simplifying assertions

Where the unittest library requires the usage of `self.assertEqual` to compare variables, `py.test` uses some magic to allow for simpler tests using regular `assert` statements.

The following test file contains both styles of tests, so they can be compared easily:

```
import cube
import pytest
import unittest


class TestCube(unittest.TestCase):
    def test_0(self):
        self.assertEqual(cube.cube(0), 0)

    def test_1(self):
        self.assertEqual(cube.cube(1), 1)

    def test_2(self):
        self.assertEqual(cube.cube(2), 8)

    def test_3(self):
        self.assertEqual(cube.cube(3), 27)

    def test_no_arguments(self):
        with self.assertRaises(TypeError):
            cube.cube()

    def test_exception_str(self):
        with self.assertRaises(TypeError):
            cube.cube('x')


class TestPyCube(object):
    def test_0(self):
        assert cube.cube(0) == 0

    def test_1(self):
        assert cube.cube(1) == 1

    def test_2(self):
        assert cube.cube(2) == 8
```

```
        def test_3(self):
            assert cube.cube(3) == 27

        def test_no_arguments(self):
            with pytest.raises(TypeError):
                cube.cube()

        def test_exception_str(self):
            with pytest.raises(TypeError):
                cube.cube('x')
```

So what did we do? Well, we simply replaced `self.assertEqual` with `assert ... == ...` and `with self.assertRaises` with `with pytest.raises`. A minor improvement indeed, but the actual benefit is seen in the failure output. The first two use the `unittest` style and the latter two use the `py.test` style:

```
============================== FAILURES ==============================
_____________________________ TestCube.test_2 ______________________________

self = <test_cube.TestCube testMethod=test_2>

    def test_2(self):
>       self.assertEqual(cube.cube(2), 8)
E       AssertionError: 4 != 8

test_cube.py:14: AssertionError
_____________________________ TestCube.test_3 ______________________________

self = <test_cube.TestCube testMethod=test_3>

    def test_3(self):
>       self.assertEqual(cube.cube(3), 27)
E       AssertionError: 9 != 27

test_cube.py:17: AssertionError
_____________________________ TestPyCube.test_2 ____________________________

self = <test_cube.TestPyCube object at 0x107c7bef0>
```

```
    def test_2(self):
>       assert cube.cube(2) == 8
E       assert 4 == 8
E        +  where 4 = <function cube at 0x107bb7c80>(2)
E        +    where <function cube at 0x107bb7c80> = cube.cube

test_cube.py:36: AssertionError
_____________________ TestPyCube.test_3 _____________________

self = <test_cube.TestPyCube object at 0x107c56a90>

    def test_3(self):
>       assert cube.cube(3) == 27
E       assert 9 == 27
E        +  where 9 = <function cube at 0x107bb7c80>(3)
E        +    where <function cube at 0x107bb7c80> = cube.cube

test_cube.py:39: AssertionError
================ 4 failed, 8 passed in 0.05 seconds ================
```

Therefore, in addition to seeing the values that were compared, we can actually see the function that was called and which input parameters it received. With the static numbers that we have here, it may not be that useful, but it is invaluable when using variables, as we'll see in the next paragraphs.

The preceding tests are all stored in a class. With `py.test`, that's completely optional, however. If readability or inheritance makes it useful to encapsulate the tests in a class, then feel free to do so, but as far as `py.test` is concerned, there is no advantage.

The standard `py.test` behavior works for most test cases, but it may not be enough for some custom types. For example, let's say that we have a `Spam` object with a `count` attribute that should be compared with the `count` attribute on another object. This part can easily be achieved by implementing the `__eq__` method on `Spam`, but it does not improve clarity. Since `count` is the attribute that we compare, it would be useful if the tests show `count` when errors are displayed. First is the class with two tests, one working and one broken to demonstrate the regular output:

test_spam.py

```
class Spam(object):
    def __init__(self, count):
```

```
            self.count = count

        def __eq__(self, other):
            return self.count == other.count


    def test_spam_equal_correct():
        a = Spam(5)
        b = Spam(5)

        assert a == b


    def test_spam_equal_broken():
        a = Spam(5)
        b = Spam(10)

        assert a == b
```

And here is the regular `py.test` output:

```
============================== FAILURES ==============================
_________________________ test_spam_equal_broken _________________________

    def test_spam_equal_broken():
        a = Spam(5)
        b = Spam(10)

>       assert a == b
E       assert <test_spam.Spam object at 0x105b484e0> == <test_spam.Spam
object at 0x105b48518>

test_spam.py:20: AssertionError
================ 1 failed, 1 passed in 0.01 seconds ================
```

The default test output is still usable since the function is fairly straightforward, and the value for `count` is visible due to it being available in the constructor. However, it would have been more useful if we could explicitly see the value of `count`. By adding a `pytest_assertrepr_compare` function to the `conftest.py` file, we can modify the behavior of the `assert` statements.

That's a special file for `py.test` that can be used to override or extend `py.test`. Note that this file will automatically be loaded by every test run in that directory, so we need to test the types of both the left-hand side and the right-hand side of the operator. In this case, it's `a` and `b`.

conftest.py

```
import test_spam


def pytest_assertrepr_compare(config, op, left, right):
    left_spam = isinstance(left, test_spam.Spam)
    right_spam = isinstance(right, test_spam.Spam)
    if left_spam and right_spam and op == '==':
        return [
            'Comparing Spam instances:',
            '    counts: %s != %s' % (left.count, right.count),
        ]
```

The preceding function will be used as the output for our test. So when it fails, this time we get our own, slightly more useful, output:

```
============================ FAILURES ============================
_____________________ test_spam_equal_broken _____________________

    def test_spam_equal_broken():
        a = Spam(5)
        b = Spam(10)

>       assert a == b
E       assert Comparing Spam instances:
E            counts: 5 != 10

test_spam.py:20: AssertionError
================ 1 failed, 1 passed in 0.01 seconds ================
```

In this case, we could have easily changed the `__repr__` function of `Spam` as well, but there are many cases where modifying the `py.test` output can be useful. Similar to this, there is specific support for many types, such as sets, dictionaries, and texts.

Parameterizing tests

So far, we have specified every test separately, but we can simplify tests a lot by parameterizing them. Both the square and cube tests were very similar; a certain input gave a certain output. This is something that can easily be verified using a loop, of course, but using a loop in a test has a pretty big downside. It will be executed as a single test. This means that it will fail in its entirety if a single test iteration of the loop fails, and that is a problem. Instead of having an output for every version, you will get it only once, while they actually might be separate bugs. That's where parameters help. You can simply create a list of parameters and the expected data and make it run the test function for every parameter separately:

```
import cube
import pytest


cubes = (
    (0, 0),
    (1, 1),
    (2, 8),
    (3, 27),
)


@pytest.mark.parametrize('n,expected', cubes)
def test_cube(n, expected):
    assert cube.cube(n) == expected
```

This outputs the following, as you might have already expected:

```
============================== FAILURES ==============================
_____________________________ test_cube[2-8] ______________________________

n = 2, expected = 8

    @pytest.mark.parametrize('n,expected', cubes)
    def test_cube(n, expected):
>       assert cube.cube(n) == expected
E       assert 4 == 8
E        +  where 4 = <function cube at 0x106576268>(2)
E        +    where <function cube at 0x106576268> = cube.cube

test_cube.py:15: AssertionError
_____________________________ test_cube[3-27] _____________________________
```

```
n = 3, expected = 27

    @pytest.mark.parametrize('n,expected', cubes)
    def test_cube(n, expected):
>       assert cube.cube(n) == expected
E       assert 9 == 27
E        +  where 9 = <function cube at 0x106576268>(3)
E        +    where <function cube at 0x106576268> = cube.cube

test_cube.py:15: AssertionError
================ 2 failed, 2 passed in 0.02 seconds ================
```

With the parameterized tests, we can see the parameters clearly, which means we can see all inputs and outputs without any extra effort.

Generating the list of tests dynamically at runtime is also possible with a global function. Similar to the `pytest_assertrepr_compare` function that we added to `conftest.py` earlier, we can add a `pytest_generate_tests` function, which generates tests.

Creating the `pytest_generate_tests` function can be useful only to test a subset of options depending on the configuration options. If possible, however, I recommend trying to configure selective tests using fixtures instead, as they are somewhat more explicit. The problem with functions such as `pytest_generate_tests` is that they are global and don't discriminate between specific tests, resulting in strange behavior if you are not expecting that.

Automatic arguments using fixtures

The fixture system is one of the most magical features of `py.test`. It magically executes a fixture function with the same name as your arguments. Because of this, the naming of the arguments becomes very important, as they can easily collide with other fixtures. To prevent collisions, the scope is set to `function` by default. However, `class`, `module`, and `session` are also valid options for the scope. There are several fixtures available by default, some of which you will use often, and others most likely never. A complete list can always be generated with the following command:

```
# py.test --quiet --fixtures
cache
    Return a cache object that can persist state between testing
sessions.

    cache.get(key, default)
```

```
    cache.set(key, value)

    Keys must be a ``/`` separated value, where the first part is usually
the
    name of your plugin or application to avoid clashes with other cache
users.

    Values can be any object handled by the json stdlib module.
capsys
    enables capturing of writes to sys.stdout/sys.stderr and makes
    captured output available via ``capsys.readouterr()`` method calls
    which return a ``(out, err)`` tuple.
capfd
    enables capturing of writes to file descriptors 1 and 2 and makes
    captured output available via ``capfd.readouterr()`` method calls
    which return a ``(out, err)`` tuple.
record_xml_property
    Fixture that adds extra xml properties to the tag for the calling
test.
    The fixture is callable with (name, value), with value being
automatically
    xml-encoded.
monkeypatch
    The returned ``monkeypatch`` funcarg provides these
    helper methods to modify objects, dictionaries or os.environ::

    monkeypatch.setattr(obj, name, value, raising=True)
    monkeypatch.delattr(obj, name, raising=True)
    monkeypatch.setitem(mapping, name, value)
    monkeypatch.delitem(obj, name, raising=True)
    monkeypatch.setenv(name, value, prepend=False)
    monkeypatch.delenv(name, value, raising=True)
    monkeypatch.syspath_prepend(path)
    monkeypatch.chdir(path)

    All modifications will be undone after the requesting
    test function has finished. The ``raising``
```

```
    parameter determines if a KeyError or AttributeError
    will be raised if the set/deletion operation has no target.
pytestconfig
    the pytest config object with access to command line opts.
recwarn
    Return a WarningsRecorder instance that provides these methods:

    * ``pop(category=None)``: return last warning matching the category.
    * ``clear()``: clear list of warnings

    See http://docs.python.org/library/warnings.html for information
    on warning categories.
tmpdir_factory
    Return a TempdirFactory instance for the test session.
tmpdir
    return a temporary directory path object
    which is unique to each test function invocation,
    created as a sub directory of the base temporary
    directory.  The returned object is a `py.path.local`_
    path object.
```

The standard fixtures are quite well documented, but a few examples never hurt. The next paragraphs demonstrate fixture usage.

Cache

The cache fixture is as simple as it is useful; there is a `get` function and a `set` function, and it remains between sessions. This test, for example, will allow five executions and raise an error every time after that. While it is not the most useful and elaborate example, it does show how the `cache` function works:

```
def test_cache(cache):
    counter = cache.get('counter', 0)
    assert counter < 5
    cache.set('counter', counter + 1)
```

The default value (`0` in this case) is required for the `cache.get` function.

The cache can be cleared through the `--cache-clear` command-line parameter, and all caches can be shown through `--cache-show`.

Custom fixtures

Bundled fixtures are quite useful, but within most projects, you will need to create your own fixtures to make things easier. Fixtures make it trivial to repeat code that is needed more often. You are most likely wondering how this is different from a regular function, context wrapper, or something else, but the special thing about fixtures is that they themselves can accept fixtures as well. So, if your function needs the `pytestconfig` variables, it can ask for it without needing to modify the calling functions.

The use cases for fixtures strongly depend on the projects, and because of that, it is difficult to generate a universally useful example, but a theoretical one is of course an option. The basic premise is simple enough, though: a function with the `pytest.fixture` decorator, which returns a value that will be passed along as an argument. Also, the function can take parameters and fixtures just as any test can. The only notable variation is `pytest.yield_fixture`. This fixture variation has one small difference; the actual test will be executed at the `yield` (more than one `yield` results in errors) and the code before/after functions as setup/teardown code. The most basic example of a `fixture` with `yield_fixture` looks like this:

```
import pytest


@pytest.yield_fixture
def some_yield_fixture():
    # Before the function
    yield 'some_value_to_pass_as_parameter'
    # After the function


@pytest.fixture
def some_regular_fixture():
    # Do something here
    return 'some_value_to_pass_as_parameter'
```

These fixtures take no parameters and simply pass a parameter to the `py.test` functions. A more useful example would be setting up a database connection and executing a query in a transaction:

```
import pytest
import sqlite3


@pytest.fixture(params=[':memory:'])
def connection(request):
    return sqlite3.connect(request.param)
```

```
@pytest.yield_fixture
def transaction(connection):
    with connection:
        yield connection


def test_insert(transaction):
    transaction.execute('create table test (id integer)')
    transaction.execute('insert into test values (1), (2), (3)')
```

Naturally, instead of using the `:memory:` database in `sqlite3`, we can use a different database name (or several) as well.

Print statements and logging

Even though print statements are generally not the most optimal way to debug code, I admit that it is still my default method of debugging. This means that when running and trying tests, I will include many print statements. However, let's see what happens when we try this with `py.test`. Here is the testing code:

```
import sys
import logging


def test_print():
    print('Printing to stdout')
    print('Printing to stderr', file=sys.stderr)
    logging.debug('Printing to debug')
    logging.info('Printing to info')
    logging.warning('Printing to warning')
    logging.error('Printing to error')
```

The following is the actual output:

```
# py.test test_print.py -v
======================== test session starts ========================
platform darwin -- Python 3.5.1, pytest-2.8.5, py-1.4.31, pluggy-0.3.1
cachedir: ../.cache
rootdir: code, inifile: pytest.ini
collected 1 items

test_print.py .

===================== 1 passed in 0.01 seconds ======================
```

So, all of our print statements and logging got trashed? Well, not really. In this case, `py.test` assumed that it wouldn't be relevant to you, so it ignored the output. But what about the same test with an error?

```
import sys
import logging


def test_print():
    print('Printing to stdout')
    print('Printing to stderr', file=sys.stderr)
    logging.debug('Printing to debug')
    logging.info('Printing to info')
    logging.warning('Printing to warning')
    logging.error('Printing to error')
    assert False, 'Dying because we can'
```

And the output with the error?

```
============================== FAILURES ==============================
_______________________________ test_print ________________________________

    def test_print():
        print('Printing to stdout')
        print('Printing to stderr', file=sys.stderr)
        logging.debug('Printing to debug')
        logging.info('Printing to info')
        logging.warning('Printing to warning')
        logging.error('Printing to error')
>       assert False, 'Dying because we can'
E       AssertionError: Dying because we can
E       assert False

test_print.py:12: AssertionError
------------------------ Captured stdout call -------------------------
Printing to stdout
------------------------ Captured stderr call -------------------------
Printing to stderr
WARNING:root:Printing to warning
ERROR:root:Printing to error
==================== 1 failed in 0.01 seconds ====================
```

Wow! Do you see that? The `stdout`, `stderr`, and logging with a level of `WARNING` or higher do get output now. `DEBUG` and `INFO` still won't be visible, but we'll see more about that later in this chapter, in the logging section.

Plugins

One of the most powerful features of `py.test` is the plugin system. Within `py.test`, nearly everything can be modified using the available hooks, the result of which is that writing plugins is almost simple. Actually, you already wrote a few plugins in the previous paragraphs without realizing it. By packaging `conftest.py` in a different package or directory, it becomes a `py.test` plugin. We will explain more about packaging in *Chapter 15*, *Packaging – Creating Your Own Libraries or Applications*. Generally, it won't be required to write your own plugin because the odds are that the plugins you seek are already available. A small list of plugins can be found on the `py.test` website at `https://pytest.org/latest/plugins.html`, and a longer list can be found through the Python package index at `https://pypi.python.org/pypi?%3Aaction=search&term=pytest-`.

By default, `py.test` does cover quite a bit of the desirable features, so you can easily do without plugins, but within the packages that I write myself, I generally default to the following list:

- `pytest-cov`
- `pytest-pep8`
- `pytest-flakes`

By using these plugins, it becomes much easier to maintain the code quality of your project. In order to understand why, we will take a closer look at these packages in the following paragraphs.

pytest-cov

Using the `pytest-cov` package, you can see whether your code is properly covered by tests or not. Internally, it uses the `coverage` package to detect how much of the code is being tested. To demonstrate the principle, we will check the coverage of a `cube_root` function.

Make sure you have `pytest-cov` installed:
`pip install pytest-cov`

First of all, let's create a `.coveragerc` file with some useful defaults:

```
[report]
# The test coverage you require, keeping to 100% is not easily
# possible for all projects but it's a good default for new projects.
fail_under = 100

# These functions are generally only needed for debugging and/or
# extra safety so we want to ignore them from the coverage
# requirements
exclude_lines =
    # Make it possible to ignore blocks of code
    pragma: no cover

    # Generally only debug code uses this
    def __repr__

    # If a debug setting is set, skip testing
    if self\.debug:
    if settings.DEBUG

    # Don't worry about safety checks and expected errors
    raise AssertionError
    raise NotImplementedError

    # This code will probably never run so don't complain about that
    if 0:
    if __name__ == .__main__.:
    @abc.abstractmethod

[run]
# Make sure we require that all branches of the code is covered. So
# both the if and the else
branch = True

# No need to test the testing code
omit =
    test_*.py
```

Here is the `cube_root.py` code:

```
def cube_root(n):
    '''
    Returns the cube root of the input number
```

```
    Args:
        n (int): The number to cube root

    Returns:
        int: The cube root result
    '''
    if n >= 0:
        return n ** (1/3)
    else:
        raise ValueError('A number larger than 0 was expected')
```

And the `test_cube_root.py` code:

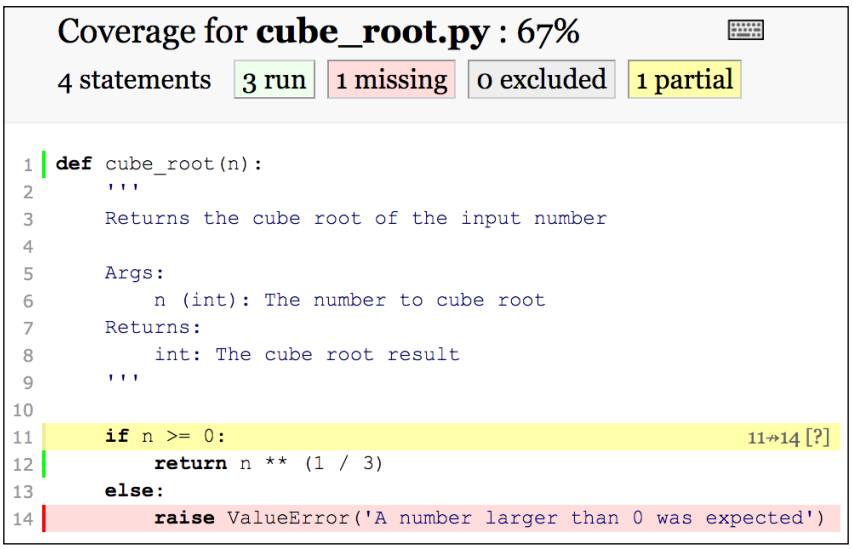
```
import pytest
import cube_root


cubes = (
    (0, 0),
    (1, 1),
    (8, 2),
    (27, 3),
)


@pytest.mark.parametrize('n,expected', cubes)
def test_cube_root(n, expected):
    assert cube_root.cube_root(n) == expected
```

Now let's see what happens when we run this with the `--cov-report=html` parameter:

```
# py.test test_cube_root.py --cov-report=html --cov-report=term-missing
--cov=cube_root.py
======================= test session starts =======================
platform darwin -- Python 3.5.1, pytest-2.8.5, py-1.4.31, pluggy-0.3.1
rootdir: code, inifile: pytest.ini
plugins: cov-2.2.0
collected 4 items

test_cube_root.py ....
--------- coverage: platform darwin, python 3.5.1-final-0 ----------
Name           Stmts   Miss Branch BrPart  Cover   Missing
```

```
-----------------------------------------------------------
cube_root.py       4      1      2      1    67%   14, 11->14
Coverage HTML written to dir htmlcov
Traceback (most recent call last):
...
pytest_cov.plugin.CoverageError: Required test coverage of 100% not
reached. Total coverage: 66.67%
```

What happened here? It looks like we forgot to test some part of the code: line `14` and the branch that goes from line `11` to line `14`. This output isn't all that readable, and that's why we specified the HTML output as well:



Perfect! So now we know. We forgot to test for values smaller than `0`.

The yellow line indicates that only one part of the branch was executed (`(n >= 0) == True`) and not the other (`(n >= 0) == False`), this occurs with `if` statements, loops, and other things where at least one of the branches is not covered. For example, if a loop over an empty array is an impossible scenario, then the test can be partially skipped:

```
#  pragma: no branch
```

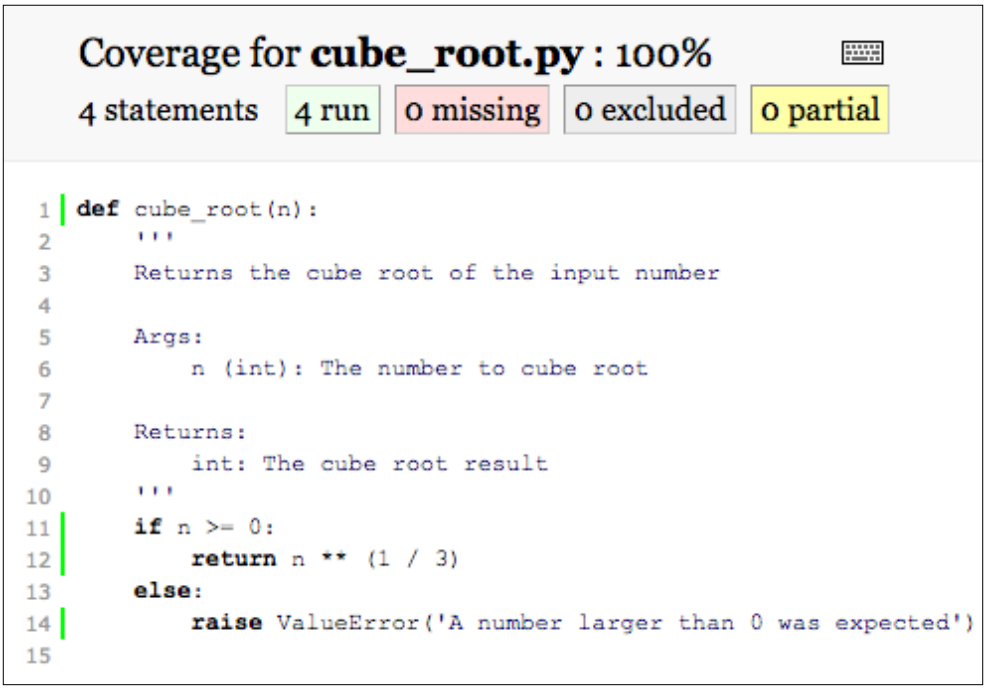
But since we know the problem, that is, the missing test for `ValueError`, let's add the test case:

```
import cube
import pytest


cubes = (
    (0, 0),
    (1, 1),
    (2, 8),
    (3, 27),
)


@pytest.mark.parametrize('n,expected', cubes)
def test_cube(n, expected):
    assert cube.cube(n) == expected


def test_cube_root_below_zero():
    with pytest.raises(ValueError):
        cube_root.cube_root(-1)
```

Then we run the test again:

```
# py.test test_cube_root.py --cov-report=html --cov-report=term-missing
--cov=cube_root.py
======================== test session starts ========================
platform darwin -- Python 3.5.1, pytest-2.8.5, py-1.4.31, pluggy-0.3.1
rootdir: code, inifile: pytest.ini
plugins: cov-2.2.0
collected 5 items

test_cube_root.py .....
---------- coverage: platform darwin, python 3.5.1-final-0 -----------
Name            Stmts   Miss Branch BrPart  Cover   Missing
-------------------------------------------------------------
cube_root.py       4      0      2      0   100%
Coverage HTML written to dir htmlcov

===================== 5 passed in 0.03 seconds ======================
```

Perfect! 100% coverage without a problem, and the HTML output is also exactly what we expect:



But what if the code was slightly different? Instead of raising a `ValueError` for values below `0`, what if we just raise a `NotImplementedError`?

```
def cube_root(n):
    '''
    Returns the cube root of the input number

    Args:
        n (int): The number to cube root

    Returns:
        int: The cube root result
    '''
    if n >= 0:
        return n ** (1 / 3)
    else:
        raise NotImplementedError(
            'A number larger than 0 was expected')
```

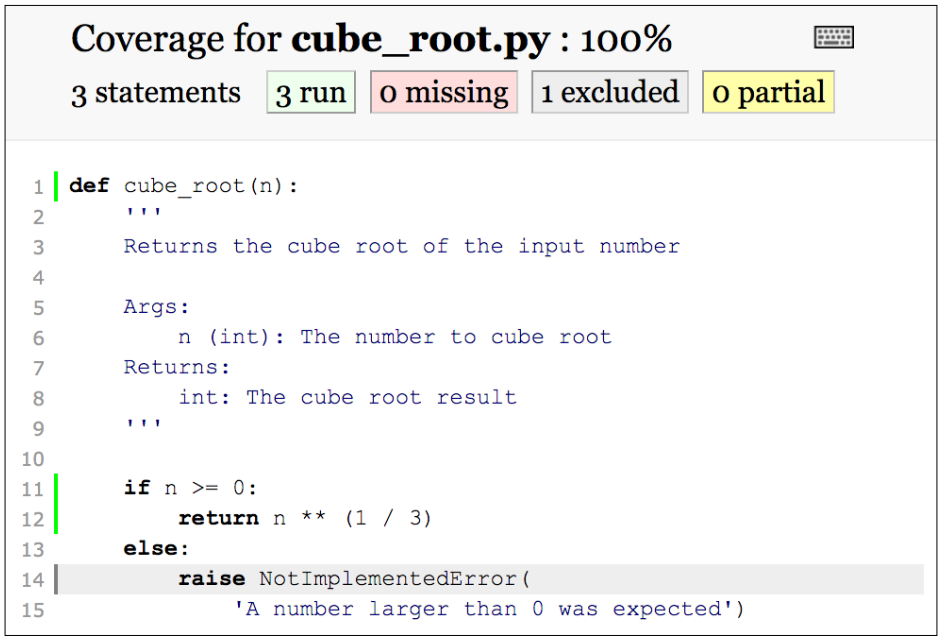
And remove the extra test as well:

```
import cube_root
import pytest


cubes = (
    (0, 0),
    (1, 1),
    (8, 2),
    (27, 3),
)


@pytest.mark.parametrize('n,expected', cubes)
def test_cube_root(n, expected):
    assert cube_root.cube_root(n) == expected
```

Run the test again:

```
# py.test test_cube_root.py --cov-report=html --cov-report=term-missing
--cov=cube_root.py
======================== test session starts ========================
platform darwin -- Python 3.5.1, pytest-2.8.5, py-1.4.31, pluggy-0.3.1
rootdir: code, inifile: pytest.ini
plugins: cov-2.2.0
collected 4 items

test_cube_root.py ....
---------- coverage: platform darwin, python 3.5.1-final-0 -----------
Name            Stmts   Miss Branch BrPart  Cover   Missing
-------------------------------------------------------------
cube_root.py       3      0      0      0   100%
Coverage HTML written to dir htmlcov

===================== 4 passed in 0.03 seconds ======================
```

You might wonder why we get 100% test coverage now though we actually didn't cover `NotImplementedError`. This is because we added `raise NotImplementedError` to the ignore list in the `.coveragerc` file. This also gives us a different result in the HTML output:



Even if we add the test for `NotImplementedError` in the test file, the coverage report will still ignore the line.

pytest-pep8 and pytest-flakes

Pyflakes and pep8 are code quality testing tools that are very useful for making your code readable and pep8 compliant. The `pytest-pep8` and `pytest-flakes` modules automatically execute these checks before running the actual tests. To install them, simply execute this line:

```
# pip install pytest-flakes pytest-pep8
```

After that, you'll be able to run both of them like this:

```
# py.test --flakes --pep8 cube_root.py
======================== test session starts ========================
platform darwin -- Python 3.5.1, pytest-2.8.5, py-1.4.31, pluggy-0.3.1
rootdir: code, inifile: pytest.ini
plugins: cov-2.2.0, flakes-1.0.1, pep8-1.0.6
```

```
collected 2 items

cube_root.py ..

====================== 2 passed in 0.01 seconds ======================
```

Configuring plugins

To make sure that all the plugins get executed and to configure them, simply add the settings to the `pytest.ini` file. The following example can be a reasonable default for development, but for production releases, you will probably want to care of the `UnusedImport` warnings.

pytest.ini:

```
[pytest]
python_files =
    your_project_source/*.py
    tests/*.py

addopts =
    --doctest-modules
    --cov your_project_source
    --cov-report term-missing
    --cov-report html
    --pep8
    --flakes

# W391 is the error about blank lines at the end of a file
pep8ignore =
    *.py W391

# Ignore unused imports
flakes-ignore =
    *.py UnusedImport
```

When debugging to find out why a test is failing, it can be useful to simply look at the first test that fails. The `py.test` module offers both a `-x` flag to stop after the first failure and `--maxfail=n` to stop after *n* failures.

Mock objects

When writing tests, this regularly occurs: you are testing not only your own code but also the interaction with external resources, such as hardware, databases, web hosts, servers, and others. Some of these can be run safely, but certain tests are too slow, too dangerous, or even impossible to run. In those cases, mock objects are your friends; they can be used to fake anything, so you can be certain that your code still returns the expected results without having any variation from external factors.

Using unittest.mock

The `unittest.mock` library provides two base objects, `Mock` and `MagicMock`, to easily mock any external resources. The `Mock` object is just a general generic mock object and `MagicMock` is mostly the same, but it has all the magic methods such as `__contains__` and `__len__` defined. In addition to this, it can make your life even easier. This is because in addition to creating mock objects manually, it is possible to patch objects directly using the `patch` decorator/context manager.

The following function uses `random` to return `True` or `False` given governed by a certain probability distribution. Due to the random nature of a function like this, it is notoriously difficult to test, but not with `unittest.mock`. With the use of `unittest.mock`, it's easy to get repeatable results:

```
from unittest import mock
import random


def bernoulli(p):
    return random.random() > p


@mock.patch('random.random')
def test_bernoulli(mock_random):
    # Test for random value of 0.1
    mock_random.return_value = 0.1
    assert bernoulli(0.0)
    assert not bernoulli(0.1)
    assert mock_random.call_count == 2
```

Wonderful, isn't it? Without having to modify the original code, we can make sure that `random.random` now returns `0.1` instead of some random number. For completeness, the version that uses a context manager is given here:

```
from unittest import mock
import random


def bernoulli(p):
    return random.random() > p


def test_bernoulli():
    with mock.patch('random.random') as mock_random:
        mock_random.return_value = 0.1
        assert bernoulli(0.0)
        assert not bernoulli(0.1)
        assert mock_random.call_count == 2
```

The possibilities with mock objects are nearly endless. They vary from raising exceptions on access to faking entire APIs and returning different results on multiple calls. For example, let's fake deleting a file:

```
import os
from unittest import mock


def delete_file(filename):
    while os.path.exists(filename):
        os.unlink(filename)


@mock.patch('os.path.exists', side_effect=(True, False, False))
@mock.patch('os.unlink')
def test_delete_file(mock_exists, mock_unlink):
    # First try:
    delete_file('some non-existing file')

    # Second try:
    delete_file('some non-existing file')
```

Quite a bit of magic in this example! The `side_effect` parameter tells mock to return those values in that sequence, making sure that the first call to `os.path.exists` returns `True` and the other two return `False`. The `mock.patch` without arguments simply returns a callable that does nothing.

Using py.test monkeypatch

The `monkeypatch` object in `py.test` is a fixture that allows mocking as well. While it may seem useless after seeing the possibilities with `unittest.mock`, in summary, it's not. Some of the functionality does overlap, but while `unittest.mock` focuses on controlling and recording the actions of an object, the `monkeypatch` fixture focuses on simple and temporary environmental changes. Some examples of these are given in the following list:

- Setting and deleting attributes using `monkeypatch.setattr` and `monkeypatch.delattr`
- Setting and deleting dictionary items using `monkeypatch.setitem` and `monkeypatch.delitem`
- Setting and deleting environment variables using `monkeypatch.setenv` and `monkeypatch.delenv`
- Inserting an extra path to `sys.path` before all others using `monkeypatch.syspath_prepend`
- Changing the directory using `monkeypatch.chdir`

To undo all modifications, simply use `monkeypatch.undo`.

For example, let's say that for a certain test, we need to work from a different directory. With mock, your options would be to mock pretty much all file functions, including the `os.path` functions, and even in that case, you will probably forget about a few. So, it's definitely not useful in this case. Another option would be to put the entire test into a `try...finally` block and just do an `os.chdir` before and after the testing code. This is quite a good and safe solution, but it's a bit of extra work, so let's compare the two methods:

```
import os


def test_chdir_monkeypatch(monkeypatch):
    monkeypatch.chdir('/dev')
    assert os.getcwd() == '/dev'
    monkeypatch.chdir('/')
    assert os.getcwd() == '/'


def test_chdir():
    original_directory = os.getcwd()
    try:
        os.chdir('/dev')
```

```
        assert os.getcwd() == '/dev'
        os.chdir('/')
        assert os.getcwd() == '/'
    finally:
        os.chdir(original_directory)
```

They effectively do the same, but one needs four lines of code whereas the other needs eight. All of these can easily be worked around with a few extra lines of code, of course, but the simpler the code is, the fewer mistakes you can make and the more readable it is.

Logging

The Python logging module is one of those modules that are extremely useful, but it tends to be very difficult to use correctly. The result is often that people just disable logging completely and use print statements instead. This is insightful but a waste of the very extensive logging system in Python. If you've written Java code before, you might be familiar with the Log4j Java library. The Python logging module is largely and primarily based on that library.

The most important objects of the logging module are the following:

- **Logger**: the actual logging interface
- **Handler**: This processes the log statements and outputs them
- **Formatter**: This formats the input data into a string
- **Filter**: This allows filtering of certain messages

Within these objects, you can set the logging levels to one of the default levels:

- `CRITICAL: 50`
- `ERROR: 40`
- `WARNING: 30`
- `INFO: 20`
- `DEBUG: 10`
- `NOTSET: 0`

The numbers are the numeric values of these log levels. While you can generally ignore them, the order is obviously important while setting the minimum level. Also, when defining custom levels, you will have to overwrite existing levels if they have the same numeric value.

Configuration

There are several ways to configure the logging system, ranging from pure code to JSON files or even remote configuration. The examples will use parts of the logging module later discussed in this chapter, but the usage of the config system is all that matters here. If you are not interested in the internal workings of the logging module, you should be able to get by with just this paragraph of the logging section.

Basic logging configuration

The most basic logging configuration is, of course, no configuration, but that will not get you much useful output:

```
import logging

logging.debug('debug')
logging.info('info')
logging.warning('warning')
logging.error('error')
logging.critical('critical')
```

With the default log level, you will only see a warning and up:

```
# python log.py
WARNING:root:warning
ERROR:root:error
CRITICAL:root:critical
```

A quick and easy start for a configuration is `basicConfig`. I recommend using this if you just need some quick logging for a script you're writing, but not for a full-blown application. While you can configure pretty much anything you wish, once you get a more complicated setup, there are usually more convenient options. We will talk more about that in later paragraphs, but first, we have a `basicConfig` that configures our logger to display some more information, including the logger name:

```
import logging

log_format = (
    '[%(asctime)s] %(levelname)-8s %(name)-12s %(message)s')

logging.basicConfig(
    filename='debug.log',
    format=log_format,
    level=logging.DEBUG,
)
```

```
formatter = logging.Formatter(log_format)
handler = logging.StreamHandler()
handler.setLevel(logging.WARNING)
handler.setFormatter(formatter)
logging.getLogger().addHandler(handler)
```

We test the code:

```
logging.debug('debug')
logging.info('info')
some_logger = logging.getLogger('some')
some_logger.warning('warning')
some_logger.error('error')
other_logger = some_logger.getChild('other')
other_logger.critical('critical')
```

This will give us the following output on our screen:

```
# python log.py
[2015-12-02 15:56:19,449] WARNING  some         warning
[2015-12-02 15:56:19,449] ERROR    some         error
[2015-12-02 15:56:19,449] CRITICAL some.other   critical
```

And here is the output in the `debug.log` file:

```
[2015-12-02 15:56:19,449] DEBUG    root         debug
[2015-12-02 15:56:19,449] INFO     root         info
[2015-12-02 15:56:19,449] WARNING  some         warning
[2015-12-02 15:56:19,449] ERROR    some         error
[2015-12-02 15:56:19,449] CRITICAL some.other   critical
```

This configuration shows how log outputs can be configured with separate configurations, log levels, and, if you choose so, formatting. It tends to become unreadable though, which is why it's usually a better idea to use `basicConfig` only for simple configurations that don't involve multiple handlers.

Dictionary configuration

The `dictconfig` makes it possible to name all parts so that they can be reused easily, for example, a single formatter for multiple loggers and handlers. So let's rewrite our previous configuration using `dictconfig`:

```
from logging import config

config.dictConfig({
    'version': 1,
```

```
        'formatters': {
            'standard': {
                'format': '[%(asctime)s] %(levelname)-8s '
                '%(name)-12s %(message)s',
            },
        },
        'handlers': {
            'file': {
                'filename': 'debug.log',
                'level': 'DEBUG',
                'class': 'logging.FileHandler',
                'formatter': 'standard',
            },
            'stream': {
                'level': 'WARNING',
                'class': 'logging.StreamHandler',
                'formatter': 'standard',
            },
        },
        'loggers': {
            '': {
                'handlers': ['file', 'stream'],
                'level': 'DEBUG',
            },
        },
    })
```

The nice thing about the dictionary configuration is that it's very easy to extend and/or overwrite the logging configuration. For example, if you want to change the formatter for all of your logging, you can simply change the `standard` formatter or even loop through `handlers`.

JSON configuration

Since `dictconfig` takes any type of dictionary, it is actually quite simple to implement a different type of reader employing JSON or YAML files. This is especially useful as they tend to be a bit friendlier towards non-Python programmers. As opposed to Python files, they are easily readable and writable from outside of Python.

Let's assume that we have a `log_config.json` file such as the following:

```
    {
        "version": 1,
        "formatters": {
            "standard": {
                "format": "[%(asctime)s] %(levelname)-8s %(name)-12s
    %(message)s"
```

```
            }
        },
        "handlers": {
            "file": {
                "filename": "debug.log",
                "level": "DEBUG",
                "class": "logging.FileHandler",
                "formatter": "standard"
            },
            "stream": {
                "level": "WARNING",
                "class": "logging.StreamHandler",
                "formatter": "standard"
            }
        },
        "loggers": {
            "": {
                "handlers": ["file", "stream"],
                "level": "DEBUG"
            }
        }
    }
```

We can simply use this code to read the config:

```
import json
from logging import config

with open('log_config.json') as fh:
    config.dictConfig(json.load(fh))
```

Ini file configuration

The file configuration is probably the most readable format for non-programmers. It uses the `ini-style` configuration format and uses the `configparser` module internally. The downside is that it is perhaps a little verbose, but it is clear enough and makes it easy to combine several configuration files without us having to worry too much about overwriting other configurations. Having said that, if `dictConfig` is an option, then it is most likely a better option. This is because `fileConfig` is slightly limited and awkward at times. Just look at the handlers as an example:

```
[formatters]
keys=standard

[handlers]
```

```
keys=file,stream

[loggers]
keys=root

[formatter_standard]
format=[%(asctime)s] %(levelname)-8s %(name)-12s %(message)s

[handler_file]
level=DEBUG
class=FileHandler
formatter=standard
args=('debug.log',)

[handler_stream]
level=WARNING
class=StreamHandler
formatter=standard
args=(sys.stderr,)

[logger_root]
handlers=file,stream
level=DEBUG
```

Reading the files is extremely easy though:

```
from logging import config

config.fileConfig('log_config.ini')
```

One thing to make note of, however, is that if you look carefully, you will see that this config is slightly different from the other configs. With `fileConfig` you can't just use keyword arguments alone. The `args` is required for both `FileHandler` and `StreamHandler`.

The network configuration

The network configuration is both very convenient and a bit dangerous, because it allows you to configure your logger on the fly while your application/script is still running. The dangerous part is that the config is (partially) read by using the `eval` function, which allows people to potentially execute code within your application remotely. Even though `logging.config.listen` only listens to local connections, it can still be dangerous if you execute the code on a shared/unsafe host.

Luckily, since version Python 3.4, it is possible to add a `verify` parameter, which is a function that will be executed to convert the input into the output. The default is obviously something along the lines of `lambda config: config`, but it can be configured to return just about anything.

To prove this point through an example, we need two scripts. One script will continuously print a few messages to the loggers and the other will change the logging configuration. We will start with the same test code that we had before but keep it running in an endless loop with a `sleep` in between:

```
import time
import logging
from logging import config

listener = config.listen()
listener.start()

try:
    while True:
        logging.debug('debug')
        logging.info('info')
        some_logger = logging.getLogger('some')
        some_logger.warning('warning')
        some_logger.error('error')
        other_logger = some_logger.getChild('other')
        other_logger.critical('critical')

        time.sleep(5)

except KeyboardInterrupt:
    # Stop listening and finish the listening thread
    logging.config.stopListening()
    listener.join()
```

Now comes the code that will send the configuration file:

```
import struct
import socket
from logging import config

with open('log_config.ini') as fh:
    data = fh.read()

# Open the socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
# Connect to the server
sock.connect(('127.0.0.1', config.DEFAULT_LOGGING_CONFIG_PORT))
# Send the magic logging packet
sock.send(struct.pack('>L', len(data)))
# Send the config
sock.send(data)
# And close the connection again
sock.close()
```

Next, let's see the output. After the first execution of the loop, we will execute the second script to read the logging configuration:

```
# python log_networkconfig.py
WARNING:some:warning
ERROR:some:error
CRITICAL:some.other:critical
```

You might be wondering where the rest of the output is. There is none. The `debug.log` file has been filled with messages like these, however:

```
[2015-12-03 12:32:38,894] DEBUG    root         debug
[2015-12-03 12:32:38,894] INFO     root         info
```

So what happened? This is where we see the pitfalls of custom loggers and configuration after using the loggers. The `logging.config.listen` function will modify the root logger as requested, but since the other loggers (`some` and `some.other`) weren't specified, they weren't modified. We modify the configuration to include them, as follows:

```
[formatters]
keys=standard

[handlers]
keys=file,stream

[loggers]
keys=root,some

[formatter_standard]
format=[%(asctime)s] %(levelname)-8s %(name)-12s %(message)s

[handler_file]
level=DEBUG
class=FileHandler
formatter=standard
```

```
args=('debug.log',)

[handler_stream]
level=WARNING
class=StreamHandler
formatter=standard
args=(sys.stderr,)

[logger_root]
handlers=file,stream
level=DEBUG

[logger_some]
level=DEBUG
qualname=some
handlers=
```

Now it works as expected:

```
# python log_networkconfig.py
WARNING:some:warning
ERROR:some:error
CRITICAL:some.other:critical
[2015-12-03 12:42:05,621] WARNING  some         warning
[2015-12-03 12:42:05,622] ERROR    some         error
[2015-12-03 12:42:05,622] CRITICAL some.other   critical
```

You will probably notice that we didn't add any handlers to the `some` logger. That's because the handler is already present—at the root level. However, without manually telling the logging module that the logger is there, it won't send it to the handler anymore. This is not problematic generally, but it's a dangerous pitfall when modifying logging configurations at runtime.

An alternative way to configure it without having this propagation issue is by disabling propagation altogether, but that will create an entirely new logger and will forget any configuration added to the root. So, if you have a handler for the error level at the root that gets sent to your error reporting system, it won't arrive anymore. In this case, however, the config is slightly clearer:

```
[logger_some]
handlers=file,stream
level=DEBUG
qualname=some
propagate=0
```

Logger

The main object that you will be using all the time with the `logging` module is the `Logger` object. This object contains all the APIs that you will need to do the actual logging. Most are simple enough but some require attention.

First of all, loggers inherit the parent settings by default. As we have seen previously with the propagate setting, by default, all settings will propagate from the parent. This is really useful when incorporating loggers within your files. Assuming your modules are using sane names and import paths, I recommend the following style of naming your loggers:

```
import logging


logger = logging.getLogger(__name__)


class Spam(object):
    def __init__(self, count):
        self.logger = logger.getChild(self.__class__.__name__)
```

By using this style, your loggers will get names such as `main_module.sub_module.ClassName`. Not only does this make your logs easier to read, but also it is easily possible to enable or disable logging per module with the propagation of log settings. To create a new log file that logs everything from `main_module.sub_module`, we can simply do this:

```
import logging

logger = logging.getLogger('main_module.sub_module')
logger.addHandler(logging.FileHandler('sub_module.log'))
```

Alternatively, you can configure it using your chosen configuration option, of course. The relevant point is that with sub-loggers, you have very fine-grained control over your loggers.

This includes increasing the log level:

```
import logging

logger = logging.getLogger('main_module.sub_module')
logger.setLevel(logging.DEBUG)
```

Usage

The usage of the `Logger` object is mostly identical to that of the bare `logging` module, but `Logger` actually supports a bit more. This is because the bare `logging` module just calls the functions on the root logger. It has a few very useful properties, although most of these are undocumented in the library:

- `Propagate`: Whether to pass events to this logger or to the handlers of the parent loggers. Without this, a log message to `main_module.sub_module` won't be logged by `main_module`.

  The `handle` method will keep looking for parent handlers as long as those loggers have `propagate` set to `true`, which is the default.

- `Filters`: These are the filters attached to the logger. They can be added through `addFilter` and `removeFilter`, To see whether a message will be filtered, the `filter` method can be used.
- `Disabled`: By setting this property, it's possible to disable a certain logger. The regular API only allows disabling of all loggers below a certain level. This offers some fine-grained control.
- `Handlers`: These are the handlers attached to the logger. They can be added through `addHandler` and `removeHandler`. The existence of any (inherited) handlers can be checked through the `hasHandlers` function.
- `Level`: This is really an internal one as it simply has a numeric value and not a name. But beyond that, it doesn't take inheritance into account, so it's better to avoid the property and use the `getEffectiveLevel` function instead. To check whether the setting is enabled for a `DEBUG` for example, you can simply do `logger.isEnabledFor(logging.DEBUG)`. Setting the property is possible through the `setLevel` function, of course.
- `Name`: As this property's name says, it is very useful for your own reference, of course.

Now that you know about the properties, it is time to discuss the logging functions themselves. The functions you will use most often are the `log`, `debug`, `info`, `warning`, `error`, and `critical` log functions. They can be used quite simply, but they support string formatting as well, which is very useful:

```
import logging

logger = logging.getLogger()
exception = 'Oops...'
logger.error('Some horrible error: %r', exception)
```

You might wonder why we don't simply use the regular string formatting with `%` or `string.format` instead. The reason is that when parameters are used instead of preformatted strings, the handler gets them as parameters. The result is that you can group log messages by the original string, which is what tools such as sentry (`https://github.com/getsentry/sentry`) use.

There is more to it, however. In terms of parameters, `*args` are only for string formatting, but it's possible to add extra parameters to a log object using the `extra` keyword parameter:

```
import logging

logger = logging.getLogger()
logger.error('simple error', extra=dict(spam='some spam'))
```

These `extra` parameters can be used in the logging formatter to display extra information just like the standard formatting options:

```
import logging

logging.basicConfig(format='%(spam)s: %(message)s')
logger = logging.getLogger()
logger.error('the message', extra=dict(spam='some spam'))
```

This results in the following:

```
# python test_spam.py
some spam: the message
```

However, one of the most useful features is the support for exceptions:

```
import logging

logger = logging.getLogger()

try:
    raise RuntimeError('Not enough spam')
except:
    logger.exception('Got an exception')

logger.error('And an error')
```

This results in a stack trace for the exception, but it will not kill the code:

```
# python test_spam.py
Got an exception
Traceback (most recent call last):
  File "test_spam.py", line 6, in <module>
    raise RuntimeError('Not enough spam')
RuntimeError: Not enough spam
And an error
```

Summary

This chapter showed us how to write `doctests`, make use of the shortcuts provided by `py.test`, and use the `logging` module. With testing, there is never a one-size-fits-all solution. While the `doctest` system is very useful in many cases for providing both documentation and tests at the same time, in many functions, there are edge cases that simply don't matter for documentation, but still need to be tested. This is where regular unit tests come in and where `py.test` helps a lot.

Because the `py.test` library is always evolving, this chapter cannot fully cover everything you will need, but it should provide you with enough of a basis to be able to use it effectively and extend it where needed.

The logging module is extremely useful but it's also a pain if configured incorrectly. Unfortunately, the right configuration can be a bit obscure when multiple modules are trying to configure logging simultaneously. The usage of the logging system should be clear enough for most of the common use cases now, and as long as you keep the `propagate` parameter in check, you should be fine when implementing a logging system.

Next up is debugging, where testing helps prevent bugs. We will see how to solve them effectively. In addition, the logging that we added in this chapter will help a lot in that area.

11
Debugging – Solving the Bugs

The previous chapter showed you how to add logging and tests to your code, but no matter how many tests you have, you will always have bugs. The biggest problem is always user input, as it is simply impossible to test all possible inputs, implying that at one point, we will need to debug the code.

There are many debugging techniques, and most certainly, you have already used a few of them. Within this chapter, we are going to focus on print/trace debugging and interactive debugging.

Debugging using print statements, stack traces, and logging is one of the most versatile methods to work with, and it is most likely the first type of debugging you've ever used. Even a `print 'Hello world'` can be considered this type, as the output will show you that your code is being executed correctly. There is obviously no point in explaining how and where to place print statements to debug your code, but there are quite a few nice tricks using decorators and other Python modules that render this type of debugging a lot more useful, such as `faulthandler`.

Interactive debugging is a more complicated debugging method. It allows you to debug a program while it's still running. Using this method, it's even possible to change variables while the application is running and pause the application at any point desired. The downside is that it requires some knowledge about the debugger commands to be really useful.

To summarize, we will cover the following topics:

- Debugging using `print`, `trace`, `logging`, and `faulthandler`
- Interactive debugging using `pdb`

Non-interactive debugging

The most basic form of debugging is adding a simple print statement into your code to see what is still working and what isn't. This is useful in a variety of cases and likely to help solve most of your issues. Later in this chapter, we will show some interactive debugging methods, but those are not always suitable. Interactive debugging tends to become difficult or even impossible in multithreaded environments, while on a closed-off remote server, you might need a different solution as well. Both methods have their merits, but I personally opt for non-interactive debugging 90% of the time since a simple print/log statement is usually enough to analyze the cause of a problem.

A basic example of this (I've been known to do similar) with a generator can be as follows:

```
>>> def spam_generator():
...     print('a')
...     yield 'spam'
...     print('b')
...     yield 'spam!'
...     print('c')
...     yield 'SPAM!'
...     print('d')

>>> generator = spam_generator()

>>> next(generator)
a
'spam'

>>> next(generator)
b
'spam!'
```

This shows exactly where the code does, and consequently, does not reach. Without this example, you might have expected the first print to come immediately after the `spam_generator()` call, since it's a generator. However, the execution completely stalls until we `yield` an item. Assuming you would have some setup code before the first `yield`, it won't run until `next` is actually called.

Although this is one of the simplest ways to debug functions using print statements, it's definitely not the best way. We can start by making an auto-print function that automatically increments the letter:

```
>>> import string


>>> def print_character():
...     i = 0
...     while True:
...         print('Letter: %r' % string.ascii_letters[i])
...         i = (i + 1) % len(string.ascii_letters)
...         yield
>>> # Always initialize
>>> print_character = print_character()


>>> next(print_character)
Letter: 'a'
>>> next(print_character)
Letter: 'b'
>>> next(print_character)
Letter: 'c'
```

While the print statement generator is slightly better than bare print statements, it doesn't help that much yet. It would be much more useful to see which lines were actually executed while running the code. We can do this manually using `inspect.currentframe`, but there is no need for hacking. Python has you covered with some dedicated tools.

Inspecting your script using trace

Simple print statements are useful in a lot of cases since you can easily incorporate print statements in nearly every application. It does not matter whether it's remote or local, threaded or using multiprocessing. It works almost everywhere, making it the most universal solution available, in addition to logging that is. The general solution is often not the best solution, however. There are better solutions available for the most common scenarios. One of them is the `trace` module. It offers you a way to trace every execution, relationships between functions, and a few others.

To demonstrate, we will use our previous code but without print statements:

```
def eggs_generator():
    yield 'eggs'
    yield 'EGGS!'


def spam_generator():
    yield 'spam'
    yield 'spam!'
    yield 'SPAM!'

generator = spam_generator()
print(next(generator))
print(next(generator))

generator = eggs_generator()
print(next(generator))
```

We will execute it with the trace module:

```
# python3 -m trace --trace --timing tracing.py
 --- modulename: tracing, funcname: <module>
0.00 tracing.py(1): def eggs_generator():
0.00 tracing.py(6): def spam_generator():
0.00 tracing.py(11): generator = spam_generator()
0.00 tracing.py(12): print(next(generator))
 --- modulename: tracing, funcname: spam_generator
0.00 tracing.py(7):     yield 'spam'
spam
0.00 tracing.py(13): print(next(generator))
 --- modulename: tracing, funcname: spam_generator
0.00 tracing.py(8):     yield 'spam!'
spam!
0.00 tracing.py(15): generator = eggs_generator()
 --- modulename: tracing, funcname: spam_generator
0.00 tracing.py(16): print(next(generator))
 --- modulename: tracing, funcname: eggs_generator
0.00 tracing.py(2):     yield 'eggs'
eggs
 --- modulename: trace, funcname: _unsettrace
0.00 trace.py(77):         sys.settrace(None)
```

Quite nice, isn't it? It shows you exactly which line is being executed with function names and, more importantly, which line was caused by which statement (or statements). Additionally, it shows you at what time it was executed relative to the start time of the program. This is due to the `--timing` flag.

As you might expect, this output is a bit too verbose to be universally useful. In spite of the fact that you can opt to ignore specific modules and directories by using command-line parameters, it is still too verbose in many cases. So let's go for the next solution—a context manager. The preceding output has already revealed some of the `trace` internals. The last line shows a `sys.settrace` call, which is exactly what we need for manual tracing:

```
import- sys
import trace as trace_module
import contextlib


@contextlib.contextmanager
def trace(count=False, trace=True, timing=True):
    tracer = trace_module.Trace(
        count=count, trace=trace, timing=timing)
    sys.settrace(tracer.globaltrace)
    yield tracer
    sys.settrace(None)

    result = tracer.results()
    result.write_results(show_missing=False, summary=True)


def eggs_generator():
    yield 'eggs'
    yield 'EGGS!'


def spam_generator():
    yield 'spam'
    yield 'spam!'
    yield 'SPAM!'


with trace():
    generator = spam_generator()
    print(next(generator))
    print(next(generator))

generator = eggs_generator()
print(next(generator))
```

When executed as a regular Python file, this returns:

```
# python3 tracing.py
 --- modulename: tracing, funcname: spam_generator
0.00 tracing.py(24):     yield 'spam'
spam
 --- modulename: tracing, funcname: spam_generator
0.00 tracing.py(25):     yield 'spam!'
spam!
 --- modulename: contextlib, funcname: __exit__
0.00 contextlib.py(64):          if type is None:
0.00 contextlib.py(65):              try:
0.00 contextlib.py(66):                  next(self.gen)
 --- modulename: tracing, funcname: trace
0.00 tracing.py(12):     sys.settrace(None)
```

This code immediately reveals what the trace code does internally as well: it uses `sys.settrace` to tell the Python interpreter where to send every statement that is being executed. Given this, it's obviously trivial to write the function as a decorator, but I'll leave that as an exercise to you if you need it.

Another take-away from this is that you can easily add extra filters to your trace function by wrapping `tracer.globaltrace`. The function takes the following parameters (from the standard Python documentation):

Parameter	Description
`Call`	A function is called (or some other code block entered). The global trace function is called; `arg` is `None`. The return value specifies the local trace function.
`Line`	The interpreter is about to execute a new line of code or re-execute the condition of a loop. The local trace function is called; `arg` is `None`. The return value specifies the new local trace function. See `Objects/lnotab_notes.txt` for a detailed explanation of how this works.
`return`	A function (or another code block) is about to return. The local trace function is called; `arg` is the value that will be returned or `None` if the event is caused by an exception being raised. The trace function's return value is ignored.
`exception`	This means an exception has occurred. The local trace function is called; `arg` is a tuple (`exception`, `value`, `traceback`). The return value specifies the new local trace function.

Parameter	Description
`c_call`	A C function is about to be called. This may be an extension function or a built-in function. The `arg` is the C function object.
`c_return`	A C function has returned, and `arg` is the C function object.
`c_exception`	A C function has raised an exception, and `arg` is the C function object.

As you must have expected, with a simple filter function, you can easily make sure that only specific functions will be returned, instead of the long list you would normally get. You really shouldn't underestimate the amount of data generated by tracing code with a few imports. The preceding context manager code gives over 300 lines of output.

Debugging using logging

In *Chapter 10*, *Testing and Logging – Preparing for Bugs*, our chapter about testing and logging, we saw how to create custom loggers, set the levels for them, and add handlers to specific levels. We are going to use the `logging.DEBUG` level to log now, which is nothing special by itself, but with a few decorators, we can add some very useful debug-only code.

Whenever I'm debugging, I always find it very useful to know the input and output for a function. The basic version with a decorator is simple enough to write; just print the `args` and `kwargs` and you are done. The following example goes a little further. By using the `inspect` module, we can retrieve the default arguments as well, making it possible to show all arguments with the argument names and values in all cases, even if the argument was not specified:

```
import pprint
import inspect
import logging
import functools


logging.basicConfig(level=logging.DEBUG)


def debug(function):
    @functools.wraps(function)
    def _debug(*args, **kwargs):
        try:
            result = function(*args, **kwargs)
        finally:
            # Extract the signature from the function
            signature = inspect.signature(function)
```

```
            # Fill the arguments
            arguments = signature.bind(*args, **kwargs)
            # NOTE: This only works for Python 3.5 and up!
            arguments.apply_defaults()

            logging.debug('%s(%s): %s' % (
                function.__qualname__,
                ', '.join('%s=%r' % (k, v) for k, v in
                          arguments.arguments.items()),
                pprint.pformat(result),
            ))

        return _debug


    @debug
    def spam(a, b=123):
        return 'some spam'

    spam(1)
    spam(1, 456)
    spam(b=1, a=456)
```

The following output is returned:

```
# python3 logged.py
DEBUG:root:spam(a=1, b=123): 'some spam'
DEBUG:root:spam(a=1, b=456): 'some spam'
DEBUG:root:spam(a=456, b=1): 'some spam'
```

Very nice of course, as we have a clear sight of when the function is called, which parameters were used, and what is returned. However, this is something you will probably only execute when you are actively debugging your code. You can also make the regular `logging.debug` statements in your code quite a bit more useful by adding a debug-specific logger, which shows more information. Simply replace the logging config of the preceding example with this:

```
    import logging


    log_format = (
        '[%(relativeCreated)d %(levelname)s] '
        '%(pathname)s:%(lineno)d:%(funcName)s: %(message)s'
    )
    logging.basicConfig(level=logging.DEBUG, format=log_format)
```

Then your result will be something like this:

```
# time python3 logged.py
[0 DEBUG] logged.py:31:_debug: spam(a=1, b=123): 'some spam'
[0 DEBUG] logged.py:31:_debug: spam(a=1, b=456): 'some spam'
[0 DEBUG] logged.py:31:_debug: spam(a=456, b=1): 'some spam'
python3 logged.py  0.04s user 0.01s system 96% cpu 0.048 total
```

It shows the time relative to the start of the application in milliseconds and the log level. This is followed by an identification block that shows the filename, line number, and function name that originated the logs. Of course, there is a message at the end.

Showing call stack without exceptions

When looking at how and why a piece of code is being run, it's often useful to see the entire stack trace. Simply raising an exception is, of course, an option. However, that will kill the current code execution, which is generally not something we are looking for. This is where the `traceback` module comes in handy. With just a few simple lines, we get a full (or limited, if you prefer) stack list:

```
import traceback


class Spam(object):

    def run(self):
        print('Before stack print')
        traceback.print_stack()
        print('After stack print')


class Eggs(Spam):
    pass


if __name__ == '__main__':
    eggs = Eggs()
    eggs.run()
```

This results in the following:

```
# python3 traceback_test.py
Before stack print
```

```
  File "traceback_test.py", line 18, in <module>
    eggs.run()
  File "traceback_test.py", line 8, in run
    traceback.print_stack()
After stack print
```

As you can see, the traceback simply prints without any exceptions. The `traceback` module actually has quite a few other methods for printing tracebacks based on exceptions and such, but you probably won't need them often. The most useful one is probably the `limit` parameter; this parameter allows you to limit the stack trace to the useful part. For example, if you've added this code using a decorator or helper function, you probably have no need to include those in the stack trace. That's where the `limit` parameter helps:

```
import traceback


class Spam(object):

    def run(self):
        print('Before stack print')
        traceback.print_stack(limit=-1)
        print('After stack print')


class Eggs(Spam):
    pass


if __name__ == '__main__':
    eggs = Eggs()
    eggs.run()
```

This results in the following:

```
# python3 traceback_test.py
Before stack print
  File "traceback_test.py", line 18, in <module>
    eggs.run()
After stack print
```

As you can see, the `print_stack` function itself has now been hidden from the stack trace, which makes everything a bit cleaner.

The negative limit support was added in Python 3.5. Before that, only positive limits were supported.

Debugging asyncio

The `asyncio` module has a few special provisions to make debugging somewhat easier. Given the asynchronous nature of functions within `asyncio`, this is a very welcome feat. While debugging of multithreaded/multiprocessing functions or classes can be difficult—since concurrent classes can easily change environment variables in parallel—with `asyncio`, it's just as difficult if not more.

Within most Linux/Unix/Mac shell sessions, environment variables can be set using it as a prefix:

```
SOME_ENVIRONMENT_VARIABLE=value python3 script.py
```

Also, it can be configured for the current shell session using `export`:

```
export SOME_ENVIRONMENT_VARIABLE=value
```

The current value can be fetched using the following line:

```
echo $SOME_ENVIRONMENT_VARIABLE
```

On Windows, you can configure an environment variable for your local shell session using the `set` command:

```
set SOME_ENVIRONMENT_VARIABLE=value
```

The current value can be fetched using this line:

```
set SOME_ENVIRONMENT_VARIABLE
```

When enabling the debug mode using the `PYTHONASYNCIODEBUG` environment setting the `asyncio` module will check whether every defined coroutine is actually run:

```
import asyncio


@asyncio.coroutine
def printer():
    print('This is a coroutine')

printer()
```

This results in an error for the printer coroutine, which is never yielded here:

```
# PYTHONASYNCIODEBUG=1 python3 asyncio_test.py
<CoroWrapper printer() running, defined at asyncio_test.py:4, created at
asyncio_test.py:8> was never yielded from
Coroutine object created at (most recent call last):
  File "asyncio_test.py", line 8, in <module>
    printer()
```

Additionally, the `event` loop has some log messages by default:

```
import asyncio
import logging


logging.basicConfig(level=logging.DEBUG)
loop = asyncio.get_event_loop()
```

This results in debug messages such as the following:

```
# PYTHONASYNCIODEBUG=1 python3 asyncio_test.py
DEBUG:asyncio:Using selector: KqueueSelector
DEBUG:asyncio:Close <_UnixSelectorEventLoop running=False closed=False
debug=True>
```

You might wonder why we are using the `PYTHONASYNCIODEBUG` flag instead of `loop.set_debug(True)`. The reason is that there are cases where this won't work because debugging is enabled too late. For example, when trying that with the preceding `printer()`, you will see that you won't get any errors when using `loop.set_debug(True)` alone.

When enabling debugging, the following will change:

- Coroutines that have not been yielded (as can be seen in the preceding lines) will raise an exception.
- Calling coroutines from the "wrong" thread raises an exception.
- The execution time of the selector will be logged.
- Slow callbacks (more than 100 ms) will be logged. This timeout can be modified through `loop.slow_callback_duration`.
- Warnings will be raised when resources are not closed properly.
- Tasks that were destroyed before execution will be logged.

Handling crashes using faulthandler

The `faulthandler` module helps when debugging really low-level crashes, that is, crashes that should only be possible when using low-level access to memory, such as C extensions.

For example, here's a bit of code that will cause your Python interpreter to crash:

```
import ctypes

# Get memory address 0, your kernel shouldn't allow this:
ctypes.string_at(0)
```

It results in something similar to the following:

```
# python faulthandler_test.py
zsh: segmentation fault  python faulthandler_test.py
```

That's quite an ugly response of course and gives you no possibility to handle the error. Just in case you are wondering, having a `try/except` structure won't help you in these cases either. The following code will crash exactly in the same way:

```
import ctypes

try:
    # Get memory address 0, your kernel shouldn't allow this:
    ctypes.string_at(0)
except Exception as e:
    print('Got exception:', e)
```

This is where the `faulthandler` module helps. It will still cause your interpreter to crash, but at least you will see a proper error message raised, so it's a good default if you (or any of the sublibraries) have any interaction with raw memory:

```
import ctypes
import faulthandler

faulthandler.enable()

# Get memory address 0, your kernel shouldn't allow this:
ctypes.string_at(0)
```

It results in something along these lines:

```
# python faulthandler_test.py
Fatal Python error: Segmentation fault
```

```
Current thread 0x00007fff79171300 (most recent call first):
  File "ctypes/__init__.py", line 491 in string_at
  File "faulthandler_test.py", line 7 in <module>
zsh: segmentation fault  python faulthandler_test.py
```

Obviously, it's not desirable to have a Python application exit in this manner as the code won't exit with a normal cleanup. Resources won't be closed cleanly and your exit handler won't be called. If you somehow need to catch this behavior, your best bet is to wrap the Python executable in a separate script.

Interactive debugging

Now that we have discussed basic debugging methods that will always work, we will look at interactive debugging for some more advanced debugging techniques. The previous debugging methods made variables and stacks visible through modifying the code and/or foresight. This time around, we will look at a slightly smarter method, which constitutes doing the same thing interactively, but once the need arises.

Console on demand

When testing some Python code, you may have used the interactive console a couple of times, since it's a simple yet effective tool for testing your Python code. What you might not have known is that it is actually simple to start your own shell from within your code. So, whenever you want to drop into a regular shell from a specific point in your code, that's easily possible:

```
import code


def spam():
    eggs = 123
    print('The begin of spam')
    code.interact(banner='', local=locals())
    print('The end of spam')
    print('The value of eggs: %s' % eggs)


if __name__ == '__main__':
    spam()
```

When executing that, we will drop into an interactive console halfway:

```
# python3 test_code.py
The begin of spam
>>> eggs
123
>>> eggs = 456
>>>
The end of spam
The value of eggs: 123
```

To exit this console, we can use ^*d* (*Ctrl* + *d*) on Linux/Mac systems and ^`z` (*Ctrl* + *Z*) on Windows systems.

One important thing to note here is that the scope is not shared between the two. Even though we passed along `locals()` to share the local variables for convenience, this relation is not bidirectional. The result is that even though we set `eggs` to `456` in the interactive session, it does not carry over to the outside function. You can modify variables in the outside scope through direct manipulation (for example, setting the properties) if you wish, but all variables declared locally will remain local.

Debugging using pdb

When it comes to actually debugging code, the regular interactive console just isn't suited. With a bit of effort, you can make it work, but it's just not all that convenient for debugging since you can only see the current scope and can't jump around the stack easily. With `pdb` (Python debugger), this is easily possible. So let's look at a simple example of using `pdb`:

```
import pdb


def spam():
    eggs = 123
    print('The begin of spam')
    pdb.set_trace()
    print('The end of spam')
    print('The value of eggs: %s' % eggs)


if __name__ == '__main__':
    spam()
```

This example is pretty much identical to the one in the previous paragraph, except that this time we end up in the `pdb` console instead of a regular interactive console. So let's give the interactive debugger a try:

```
# python3 test_pdb.py
The begin of spam
> test_pdb.py(8)spam()
-> print('The end of spam')
(Pdb) eggs
123
(Pdb) eggs = 456
(Pdb) continue
The end of spam
The value of eggs: 456
```

As you can see, we've actually modified the value of `eggs` now. In this case, we used the full `continue` command, but all the `pdb` commands have short versions as well. So, using `c` instead of `continue` gives the same result. Just typing `eggs` (or any other variable) will show the contents and setting the variable will simply set it, just as we would expect from an interactive session.

To get started with `pdb`, first of all, a list of the most useful (full) commands with shorthands is shown here:

Command	Explanation
`h(elp)`	This shows the list of commands (this list).
`h(elp) command`	This shows the help for the given command.
`w(here)`	Current stack trace with an arrow at the current frame.
`d(own)`	Move down/to a newer frame in the stack.
`u(p)`	Move up/to an older frame in the stack.
`s(tep)`	Execute the current line and stop as soon as possible.
`n(ext)`	Execute the current line and stop at the next line within the current function.
`r(eturn)`	Continue execution until the function returns.
`c(ont(inue))`	Continue execution up to the next breakpoint.
`l(ist) [first[, last]]`	List the lines of source code (by default, 11 lines) around the current line.
`ll | longlist`	List all of the source code for the current function or frame.
`source expression`	List the source code for the given object. This is similar to longlist.

Command	Explanation
`a(rgs)`	Print the arguments for the current function.
`pp expression`	Pretty-print the given expression.
`b(reak)`	Show the list of breakpoints.
`b(reak) [filename:] lineno`	Place a breakpoint at the given line number and, optionally, file.
`b(reak) function[, condition]`	Place a breakpoint at the given function. The condition is an expression that must evaluate to `True` for the breakpoint to work.
`cl(ear) [filename:] lineno`	Clear the breakpoint (or breakpoints) at this line.
`cl(ear) breakpoint [breakpoint ...]`	Clear the breakpoint (or breakpoints) with these numbers.
`Command`	List all defined commands.
`command breakpoint`	Specify a list of commands to execute whenever the given breakpoint is encountered. The list is ended using the `end` command.
`Alias`	List all aliases.
`alias name command`	Create an alias. The command can be any valid Python expression, so you can do the following to print all properties for an object: **`alias pd pp %1.__dict__`**
`unalias name`	Remove an alias.
`! statement`	Execute the statement at the current point in the stack. Normally the `!` sign is not needed, but this can be useful if there are collisions with debugger commands. For example, try `b = 123`.
`Interact`	Open an interactive session similar to the previous paragraph. Note that variables set within that local scope will not be transferred.

Breakpoints

It's quite a long list, but you will probably use most of these quite regularly. To highlight one of the options shown in the preceding table, let's demonstrate the setting and use of breakpoints:

```
import pdb


def spam():
```

```
        print('The begin of spam')
        print('The end of spam')


    if __name__ == '__main__':
        pdb.set_trace()
        spam()
```

So far, nothing new has happened, but let's now open the interactive debugging session, as follows:

```
# python3 test_pdb.py
> test_pdb.py(11)<module>()
-> while True:
(Pdb) source spam  # View the source of spam
  4     def spam():
  5         print('The begin of spam')
  6         print('The end of spam')

(Pdb) b 5  # Add a breakpoint to line 5
Breakpoint 1 at test_pdb.py:5

(Pdb) w  # Where shows the current line
> test_pdb.py(11)<module>()
-> while True:

(Pdb) c  # Continue (until the next breakpoint or exception)
> test_pdb.py(5)spam()
-> print('The begin of spam')

(Pdb) w  # Where again
  test_pdb.py(12)<module>()
-> spam()
> test_pdb.py(5)spam()
-> print('The begin of spam')

(Pdb) ll  # List the lines of the current function
  4     def spam():
  5 B->     print('The begin of spam')
```

```
  6         print('The end of spam')

(Pdb) b  # Show the breakpoints
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at test_pdb.py:5
        breakpoint already hit 1 time

(Pdb) cl 1  # Clear breakpoint 1
Deleted breakpoint 1 at test_pdb.py:5
```

That was a lot of output, but it's actually not as complex as it seems:

1. First, we used the `source spam` command to see the source for the `spam` function.
2. After that, we knew the line number of the first `print` statement, which we used to place a breakpoint (`b 5`) at line 5.
3. To check whether we were still at the right position, we used the `w` command.
4. Since the breakpoint was set, we used `c` to continue up to the next breakpoint.
5. Having stopped at the breakpoint at line 5, we used `w` again to confirm that.
6. Listing the code of the current function using `ll`.
7. Listing the breakpoints using `b`.
8. Removing the breakpoint again using `cl 1` with the breakpoint number from the previous command.

It all seems a bit complicated in the beginning, but you'll see that it's actually a very convenient way of debugging once you've tried a few times.

To make it even better, this time we will execute the breakpoint only when `eggs = 3`. The code is pretty much the same, although we need a variable in this case:

```
import pdb


def spam(eggs):
    print('eggs:', eggs)


if __name__ == '__main__':
    pdb.set_trace()
    for i in range(5):
        spam(i)
```

Now, let's execute the code and make sure that it only breaks at certain times:

```
# python3 test_breakpoint.py
> test_breakpoint.py(10)<module>()
-> for i in range(5):
(Pdb) source spam
  4     def spam(eggs):
  5         print('eggs:', eggs)
(Pdb) b 5, eggs == 3  # Add a breakpoint to line 5 whenever eggs=3
Breakpoint 1 at test_breakpoint.py:5
(Pdb) c  # Continue
eggs: 0
eggs: 1
eggs: 2
> test_breakpoint.py(5)spam()
-> print('eggs:', eggs)
(Pdb) a  # Show function arguments
eggs = 3
(Pdb) c  # Continue
eggs: 3
eggs: 4
```

To list what we have done:

1. First, using `source` spam, we looked for the line number.
2. After that, we placed a breakpoint with the `eggs == 3` condition.
3. Then we continued execution using `c`. As you can see, the values `0`, `1`, and `2` are printed as normal.
4. The breakpoint was reached at value `3`. To verify this we used `a` to see the function arguments.
5. And we continued to execute the rest of the code.

Catching exceptions

All of these have been manual calls to the `pdb.set_trace()` function, but in general, you are just running your application and not really expecting issues. This is where exception catching can be very handy. In addition to importing `pdb` yourself, you can run scripts through `pdb` as a module as well. Let's examine this bit of code, which dies as soon as it reaches zero division:

```
print('This still works')
1/0
print('We shouldnt reach this code')
```

If we run it using the `pdb` parameter, we can end up in the Python Debugger whenever it crashes:

```
# python3 -m pdb test_zero.py
> test_zero.py(1)<module>()
-> print('This still works')
(Pdb) w  # Where
  bdb.py(431)run()
-> exec(cmd, globals, locals)
  <string>(1)<module>()
> test_zero.py(1)<module>()
-> print('This still works')
(Pdb) s  # Step into the next statement
This still works
> test_zero.py(2)<module>()
-> 1/0
(Pdb) c  # Continue
Traceback (most recent call last):
  File "pdb.py", line 1661, in main
    pdb._runscript(mainpyfile)
  File "pdb.py", line 1542, in _runscript
    self.run(statement)
  File "bdb.py", line 431, in run
    exec(cmd, globals, locals)
  File "<string>", line 1, in <module>
  File "test_zero.py", line 2, in <module>
    1/0
ZeroDivisionError: division by zero
Uncaught exception. Entering post mortem debugging
Running 'cont' or 'step' will restart the program
> test_zero.py(2)<module>()
-> 1/0
```

A useful little trick within `pdb` is to use the *Enter* button, which, by default, will execute the previously executed command again. This is very useful when stepping through the program.

Commands

The `commands` command is a little complicated but very useful. It allows you to execute commands whenever a specific breakpoint is encountered. To illustrate this, let's start from a simple example again:

```
import pdb


def spam(eggs):
    print('eggs:', eggs)


if __name__ == '__main__':
    pdb.set_trace()
    for i in range(5):
        spam(i)
```

The code is simple enough, so now we'll add the breakpoint and the commands, as follows:

```
# python3 test_breakpoint.py
> test_breakpoint.py(10)<module>()
-> for i in range(3):
(Pdb) b spam  # Add a breakpoint to function spam
Breakpoint 1 at test_breakpoint.py:4
(Pdb) commands 1  # Add a command to breakpoint 1
(com) print('The value of eggs: %s' % eggs)
(com) end  # End the entering of the commands
(Pdb) c  # Continue
The value of eggs: 0
> test_breakpoint.py(5)spam()
-> print('eggs:', eggs)
(Pdb) c  # Continue
eggs: 0
The value of eggs: 1
> test_breakpoint.py(5)spam()
-> print('eggs:', eggs)
(Pdb) cl 1  # Clear breakpoint 1
Deleted breakpoint 1 at test_breakpoint.py:4
(Pdb) c  # Continue
eggs: 1
eggs: 2
```

As you can see, we can easily add commands to the breakpoint. After removing the breakpoint, these commands obviously won't be executed anymore.

Debugging using ipdb

While the generic Python console is useful, it can be a little rough around the edges. The IPython console offers a whole new world of extra features, which make it a much nicer console to work with. One of those features is a more convenient debugger.

First, make sure you have `ipdb` installed:

```
pip install ipdb
```

Next, let's try the debugger again with our previous script. The only small change is that we now import `ipdb` instead of `pdb`:

```
import ipdb


def spam(eggs):
    print('eggs:', eggs)


if __name__ == '__main__':
    ipdb.set_trace()
    for i in range(3):
        spam(i)
```

Then we execute it:

```
# python3 test_ipdb.py
> test_ipdb.py(10)<module>()
      9     ipdb.set_trace()
---> 10     for i in range(3):
     11         spam(i)

ipdb> b spam  # Set a breakpoint
Breakpoint 1 at test_ipdb.py:4
ipdb> c  # Continue (until exception or breakpoint)
> test_ipdb.py(5)spam()
1     4 def spam(eggs):
----> 5     print('eggs:', eggs)
      6
```

```
ipdb> a  # Show the arguments
eggs = 0
ipdb> c  # Continue
eggs: 0
> test_ipdb.py(5)spam()
1     4 def spam(eggs):
----> 5     print('eggs:', eggs)
      6

ipdb>   # Repeat the previous command, so continue again
eggs: 1
> test_ipdb.py(5)spam()
1     4 def spam(eggs):
----> 5     print('eggs:', eggs)
      6

ipdb> cl 1  # Remove breakpoint 1
Deleted breakpoint 1 at test_ipdb.py:4
ipdb> c  # Continue
eggs: 2
```

The commands are all the same, but the output is just a tad more legible in my opinion. The actual version also includes syntax highlighting, which makes the output even easier to follow.

In short, you can just replace `pdb` with `ipdb` in most situations to simply get a more intuitive debugger. But I will give you the recommendation as well, to the `ipdb` context manager:

```
import ipdb


with ipdb.launch_ipdb_on_exception():
    main()
```

This is as convenient as it looks. It simply hooks `ipdb` into your exceptions so that you can easily debug whenever needed. Combine that with a debug flag to your application to easily allow debugging when needed.

Other debuggers

`pdb` and `ipdb` are just two of the large number of debuggers available for Python. Some of the currently noteworthy debuggers are as follows:

- `pudb`: This offers a a full-screen command-line debugger
- `pdbpp`: This hooks into the regular `pdb`
- `rpdb2`: This is a remote debugger that allows hooking into running (remote) applications
- `Werkzeug`: This is a web-based debugger that allows debugging of web applications while they are running

There are many others, of course, and there isn't a single one that's the absolute best. As is the case with all tools, they all have their advantages and their fallacies, and the one that is best for your current purpose can be properly decided only by you. Chances are that your current Python IDE already has an integrated debugger.

Debugging services

In addition to debugging when you encounter a problem, there are times when you simply need to keep track of errors for later debugging. Especially when working with remote servers, these can be invaluable to detect when and how a Python process is malfunctioning. Additionally, these services offer grouping of errors as well, making them far more useful than a simple e-mail-on-exception type of script, which can quickly spam your inbox.

A nice open source solution for keeping track of errors is `sentry`. If you need a full-fletched solution that offers performance tracking as well, then Opbeat and Newrelic are very nice solutions; they offer both free and paid versions. Note that all of these also support tracking of other languages, such as JavaScript.

Summary

This chapter explained a few different debugging techniques and gotchas. There is, of course, much more that can be said about debugging, but I hope you have acquired a nice vantage point for debugging your Python code now. Interactive debugging techniques are very useful for single-threaded applications and locations where interactive sessions are available. But since that's not always the case, we also discussed some non-interactive options.

Here's an overview of all the points discussed in this chapter:

- Non-interactive debugging using:
  - `print`
  - `logging`
  - `trace`
  - `traceback`
  - `asyncio`
  - `faulthandler`
- Interactive debugging using both `pdb` and `ipdb`

In the next chapter, we will see how to monitor and improve both CPU and memory performance, as well as finding and fixing memory leaks.

12
Performance – Tracking and Reducing Your Memory and CPU Usage

Before we talk about performance, there is a quote by *Donald Knuth* you need to consider first:

> "*The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.*"

Donald Knuth is often called the father of algorithm analysis. His book series, *The Art of Computer Programming*, can be considered the Bible of all fundamental algorithms.

As long as you pick the correct data structures with the right algorithms, performance should not be something to worry about. That does not mean you should ignore performance entirely, but just make sure you pick the right battles and optimize only when it is actually needed. Micro/premature optimizations can definitely be fun, but only very rarely useful.

We have seen the performance characteristics of many data structures in *Chapter 2*, *Pythonic Syntax, Common Pitfalls, and Style Guide*, already, so we won't discuss that, but we will show you how performance can be measured and how problems can be detected. There are cases where micro optimizations make a difference, but you won't know until you measure the performance.

Within this chapter, we will cover:

- Profiling CPU usage
- Profiling memory usage
- Learning how to correctly compare performance metrics
- Optimizing performance
- Finding and fixing memory leaks

What is performance?

Performance is a very broad term. It has many different meanings and in many cases it is defined incorrectly. You have probably heard statements similar to "Language X is faster than Python". However, that statement is inherently wrong. Python is neither fast nor slow; Python is a programming language and a language has no performance metrics whatsoever. If one were to say that the CPython interpreter is faster or slower than interpreter Y for language X, that would be possible. The performance characteristics of code can vary greatly between different interpreters. Just take a look at this small test:

```
# python3 -m timeit '"".join(str(i) for i in range(10000))'
100 loops, best of 3: 2.91 msec per loop
# python2 -m timeit '"".join(str(i) for i in range(10000))'
100 loops, best of 3: 2.13 msec per loop
# pypy -m timeit '"".join(str(i) for i in range(10000))'
1000 loops, best of 3: 677 usec per loop
```

Three different interpreters with all vastly different performance! All are Python but the interpreters obviously vary. Looking at this benchmark, you might be tempted to drop the CPython interpreter completely and only use Pypy. The danger with benchmarks such as these is that they rarely offer any meaningful results. For this limited example, the Pypy interpreter was about four times faster than the CPython3 interpreter, but that has no relevance whatsoever for the general case. The only conclusion that can safely be drawn here is that this specific version of the Pypy interpreter is more than four times faster than this specific version of CPython3 for this exact test. For any other test and interpreter version the results could be vastly different.

Timeit – comparing code snippet performance

Before we can start improving performance, we need a reliable method to measure it. Python has a really nice module (`timeit`) with the specific purpose of measuring execution times of bits of code. It executes a bit of code many times to make sure there is as little variation as possible and to make the measurement fairly clean. It's very useful if you want to compare a few code snippets. Following are example executions:

```
# python3 -m timeit 'x=[]; [x.insert(0, i) for i in range(10000)]'
10 loops, best of 3: 30.2 msec per loop
# python3 -m timeit 'x=[]; [x.append(i) for i in range(10000)]'
1000 loops, best of 3: 1.01 msec per loop
# python3 -m timeit 'x=[i for i in range(10000)]'
1000 loops, best of 3: 381 usec per loop
# python3 -m timeit 'x=list(range(10000))'
10000 loops, best of 3: 212 usec per loop
```

These few examples demonstrate the performance difference between `list.insert`, `list.append`, a list comprehension, and the `list` function. But more importantly, it demonstrates how to use the `timeit` command. Naturally, the command can be used with regular scripts as well, but the `timeit` module only accepts statements as strings to execute which is a bit of an annoyance. Luckily, you can easily work around that by wrapping your code in a function and just timing that function:

```
import timeit


def test_list():
    return list(range(10000))


def test_list_comprehension():
    return [i for i in range(10000)]


def test_append():
    x = []
    for i in range(10000):
        x.append(i)

    return x
```

```
    def test_insert():
        x = []
        for i in range(10000):
            x.insert(0, i)

        return x


    def benchmark(function, number=100, repeat=10):
        # Measure the execution times
        times = timeit.repeat(function, number=number, globals=globals())
        # The repeat function gives `repeat` results so we take the min()
        # and divide it by the number of runs
        time = min(times) / number
        print('%d loops, best of %d: %9.6fs :: %s' % (
            number, repeat, time, function))


    if __name__ == '__main__':
        benchmark('test_list()')
        benchmark('test_list_comprehension()')
        benchmark('test_append()')
        benchmark('test_insert()')
```

When executing this, you will get something along the following lines:

```
# python3 test_timeit.py
100 loops, best of 10:  0.000238s :: test_list()
100 loops, best of 10:  0.000407s :: test_list_comprehension()
100 loops, best of 10:  0.000838s :: test_append()
100 loops, best of 10:  0.031795s :: test_insert()
```

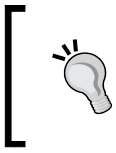
As you may have noticed, this script is still a bit basic. While the regular version keeps trying until it reaches 0.2 seconds or more, this script just has a fixed number of executions. Unfortunately, the `timeit` module wasn't entirely written with re-use in mind, so besides calling `timeit.main()` from your script there is not much you can do to re-use that logic.

Personally, I recommend using IPython instead, as it makes measurements much easier:

```
# ipython3
In [1]: import test_timeit
In [2]: %timeit test_timeit.test_list()
1000 loops, best of 3: 255 µs per loop
```

```
In [3]: %timeit test_timeit.test_list_comprehension()
1000 loops, best of 3: 430 µs per loop
In [4]: %timeit test_timeit.test_append()
1000 loops, best of 3: 934 µs per loop
In [5]: %timeit test_timeit.test_insert()
10 loops, best of 3: 31.6 ms per loop
```

In this case, IPython automatically takes care of the string wrapping and passing of `globals()`. Still, this is all very limited and useful only for comparing multiple methods of doing the same thing. When it comes to full Python applications, there are more methods available.



To view the source of both IPython functions and regular modules, entering `object??` in the IPython shell returns the source. In this case just enter `timeit??` to view the `timeit` IPython function definition.

The easiest way you can implement the `%timeit` function yourself is to simply call `timeit.main`:

```
import timeit

timeit.main(args=['[x for x in range(1000000)]'])
```

The internals of the `timeit` module are nothing special. A basic version can be implemented with just an `eval` and a `time.perf_counter` (the highest resolution timer available in Python) combination:

```
import time
import functools


TIMEIT_TEMPLATE = '''
import time

def run(number):
    %(setup)s
    start = time.perf_counter()
    for i in range(number):
        %(statement)s
    return time.perf_counter() - start
'''
```

```
def timeit(statement='pass', setup='pass', repeat=1, number=1000000,
           globals_=None):
    # Get or create globals
    globals_ = globals() if globals_ is None else globals_

    # Create the test code so we can separate the namespace
    src = TIMEIT_TEMPLATE % dict(
        statement=statement,
        setup=setup,
        number=number,
    )
    # Compile the source
    code = compile(src, '<source>', 'exec')

    # Define locals for the benchmarked code
    locals_ = {}

    # Execute the code so we can get the benchmark fuction
    exec(code, globals_, locals_)

    # Get the run function
    run = functools.partial(locals_['run'], number=number)
    for i in range(repeat):
        yield run()
```

The actual `timeit` code is a bit more advanced in terms of checking the input but this example roughly shows how the `timeit.repeat` function can be implemented.

To register your own function in IPython, you need to use some IPython magic. Note that the magic is not a pun. The IPython module that takes care of commands such as these is actually called `magic`. To demonstrate:

```
from IPython.core import magic


@magic.register_line_magic(line):
    import timeit
    timeit.main(args[line])
```

To learn more about custom magic in IPython, take a look at the IPython documentation at `https://ipython.org/ipython-doc/3/config/custommagics.html`.

cProfile – finding the slowest components

The `profile` module makes it easily possible to analyze the relative CPU cycles used in a script/application. Be very careful not to compare these with the results from the `timeit` module. While the `timeit` module tries as best as possible to give an accurate benchmark of the absolute amount of time it takes to execute a code snippet, the `profile` module is only useful for relative results. The reason is that the profiling code itself incurs such a slowdown that the results are not comparable with non-profiled code. There is a way to make it a bit more accurate however, but more about that later.

Within this section we will be talking about the `profile` module but in the examples we will actually use the `cProfile` module. The `cProfile` module is a high-performance emulation of the pure Python `profile` module.

First profiling run

Let's profile our Fibonacci function from *Chapter 5*, *Decorators– Enabling Code Reuse by Decorating*, both with and without the cache function. First, the code:

```
import sys
import functools


@functools.lru_cache()
def fibonacci_cached(n):
    if n < 2:
        return n
    else:
        return fibonacci_cached(n - 1) + fibonacci_cached(n - 2)


def fibonacci(n):
    if n < 2:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)


if __name__ == '__main__':
    n = 30
```

```
    if sys.argv[-1] == 'cache':
        fibonacci_cached(n)
    else:
        fibonacci(n)
```

For readabilities sake, all `cProfile` statistics will be stripped of the `percall` and `cumtime` columns in all `cProfile` outputs. These columns are irrelevant for the purposes of these examples.

First we'll execute the function without cache:

```
# python3 -m cProfile -s calls test_fibonacci.py no_cache
         2692557 function calls (21 primitive calls) in 0.815
         seconds

   Ordered by: call count

   ncalls tottime percall filename:lineno(function)
2692537/1   0.815   0.815 test_fibonacci.py:13(fibonacci)
        7   0.000   0.000 {built-in method builtins.getattr}
        5   0.000   0.000 {built-in method builtins.setattr}
        1   0.000   0.000 {method 'update' of 'dict' objects}
        1   0.000   0.000 {built-in method builtins.isinstance}
        1   0.000   0.000 functools.py:422(decorating_function)
        1   0.000   0.815 test_fibonacci.py:1(<module>)
        1   0.000   0.000 {method 'disable' of '_lsprof.Profiler'}
        1   0.000   0.815 {built-in method builtins.exec}
        1   0.000   0.000 functools.py:43(update_wrapper)
           1   0.000   0.000 functools.py:391(lru_cache)
```

That's quite a lot of calls, isn't it? Apparently, we called the `test_fibonacci` function nearly 3 million times. That is where the profiling modules provide a lot of insight. Let's analyze the metrics a bit further:

- **Ncalls**: The number of calls that were made to the function
- **Tottime**: The total time spent in seconds within this function with all sub-functions excluded

  Percall, `tottime / ncalls`

- **Cumtime**: The total time spent within this function, including sub-functions

  Percall, `cumtime / ncalls`

Which is the most useful depends on your use case. It's quite simple to change the sort order using the `-s` parameter within the default output. But now let's see what the result is with the cached version. Once again, with stripped output:

```
# python3 -m cProfile -s calls test_fibonacci.py cache
         51 function calls (21 primitive calls) in 0.000 seconds

   Ordered by: call count

   ncalls tottime percall filename:lineno(function)
     31/1    0.000    0.000 test_fibonacci.py:5(fibonacci_cached)
        7    0.000    0.000 {built-in method builtins.getattr}
        5    0.000    0.000 {built-in method builtins.setattr}
        1    0.000    0.000 test_fibonacci.py:1(<module>)
        1    0.000    0.000 {built-in method builtins.isinstance}
        1    0.000    0.000 {built-in method builtins.exec}
        1    0.000    0.000 functools.py:422(decorating_function)
        1    0.000    0.000 {method 'disable' of '_lsprof.Profiler'}
        1    0.000    0.000 {method 'update' of 'dict' objects}
        1    0.000    0.000 functools.py:391(lru_cache)
        1    0.000    0.000 functools.py:43(update_wrapper)
```

This time we see a `tottime` of `0.000` because it's just too fast to measure. But also, while the `fibonacci_cached` function is still the most executed function, it's only being executed 31 times instead of 3 million.

Calibrating your profiler

To illustrate the difference between `profile` and `cProfile`, let's try the uncached run again with the `profile` module instead. Just a heads up, this is much slower so don't be surprised if it stalls a little:

```
   # python3 -m profile -s calls test_fibonacci.py no_cache
            2692558 function calls (22 primitive calls) in 7.696 seconds

      Ordered by: call count

      ncalls tottime percall filename:lineno(function)
   2692537/1   7.695   7.695 test_fibonacci.py:13(fibonacci)
           7   0.000   0.000 :0(getattr)
           5   0.000   0.000 :0(setattr)
```

```
     1   0.000   0.000 :0(isinstance)
     1   0.001   0.001 :0(setprofile)
     1   0.000   0.000 :0(update)
     1   0.000   0.000 functools.py:43(update_wrapper)
     1   0.000   7.696 profile:0(<code object <module> ...>)
     1   0.000   7.695 test_fibonacci.py:1(<module>)
     1   0.000   0.000 functools.py:391(lru_cache)
     1   0.000   7.695 :0(exec)
     1   0.000   0.000 functools.py:422(decorating_function)
     0   0.000         profile:0(profiler)
```

Huge difference, isn't it? Now the code is nearly 10 times slower and the only difference is using the pure Python `profile` module instead of the `cProfile` module. This does indicate a big problem with the `profile` module. The overhead from the module itself is great enough to skew the results, which means we should account for that offset. That's what the `Profile.calibrate()` function takes care of, as it calculates the bias incurred by the profile module. To calculate the bias, we can use the following script:

```
import profile

if __name__ == '__main__':
    profiler = profile.Profile()
    for i in range(10):
        print(profiler.calibrate(100000))
```

The numbers will vary slightly but you should be able to get a fair estimate of the bias using this code. If the numbers still vary a lot, you can increase the trials from `100000` to something even larger. This type of calibration only works for the profile module, but if you are looking for more accurate results and the `cProfile` module does not work for you due to inheritance or not being supported on your platform, you can use this code to set your bias globally and get more accurate results:

```
import profile


# The number here is bias calculated earlier
profile.Profile.bias = 2.0939406059394783e-06
```

For a specific `Profile` instance:

```
import profile


profiler = profile.Profile(bias=2.0939406059394783e-06)
```

Note that in general a smaller bias is better to use than a large one, because a large bias could cause very strange results. In some cases you will even get negative timings. Let's give it a try for our Fibonacci code:

```
import sys
import pstats
import profile
import functools


@functools.lru_cache()
def fibonacci_cached(n):
    if n < 2:
        return n
    else:
        return fibonacci_cached(n - 1) + fibonacci_cached(n - 2)


def fibonacci(n):
    if n < 2:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)


if __name__ == '__main__':
    profiler = profile.Profile(bias=2.0939406059394783e-06)
    n = 30

    if sys.argv[-1] == 'cache':
        profiler.runcall(fibonacci_cached, n)
    else:
        profiler.runcall(fibonacci, n)

    stats = pstats.Stats(profiler).sort_stats('calls')
    stats.print_stats()
```

While running it, it indeed appears that I've used a bias that's too large:

```
# python3 test_fibonacci.py no_cache
         2692539 function calls (3 primitive calls) in -0.778
         seconds


   Ordered by: call count
```

```
   ncalls tottime percall filename:lineno(function)
2692537/1  -0.778  -0.778 test_fibonacci.py:15(fibonacci)
        1   0.000   0.000 :0(setprofile)
        1   0.000  -0.778 profile:0(<function fibonacci at 0x...>)
        0   0.000         profile:0(profiler)
```

Still, it shows how the code can be used properly. You can even incorporate the bias calculation within the script using a snippet like this:

```
import profile


if __name__ == '__main__':
    profiler = profile.Profile()
    profiler.bias = profiler.calibrate(100000)
```

Selective profiling using decorators

Calculating simple timings is easy enough using decorators, but profiling is also important. Both are useful but serve different goals. Let's look at both the options:

```
import cProfile
import datetime
import functools


def timer(function):
    @functools.wraps(function)
    def _timer(*args, **kwargs):
        start = datetime.datetime.now()
        try:
            return function(*args, **kwargs)
        finally:
            end = datetime.datetime.now()
            print('%s: %s' % (function.__name__, end - start))
    return _timer


def profiler(function):
    @functools.wraps(function)
    def _profiler(*args, **kwargs):
        profiler = cProfile.Profile()
        try:
            profiler.enable()
            return function(*args, **kwargs)
```

```
            finally:
                profiler.disable()
                profiler.print_stats()
        return _profiler


    @profiler
    def profiled_fibonacci(n):
        return fibonacci(n)


    @timer
    def timed_fibonacci(n):
        return fibonacci(n)


    def fibonacci(n):
        if n < 2:
            return n
        else:
            return fibonacci(n - 1) + fibonacci(n - 2)


    if __name__ == '__main__':
        timed_fibonacci(32)
        profiled_fibonacci(32)
```

The code is simple enough, just a basic timer and profiler printing some default statistics. Which functions best for you depends on your use-case of course, but they definitely both have their uses. The added advantage of this selective profiling is that the output is more limited which helps with readability:

```
# python3 test_fibonacci.py
 timed_fibonacci: 0:00:01.050200
         7049157 function calls (3 primitive calls) in 2.024
         seconds


   Ordered by: standard name


   ncalls tottime percall filename:lineno(function)
        1   0.000   2.024 test_fibonacci.py:31(profiled_fibonacci)
7049155/1   2.024   2.024 test_fibonacci.py:41(fibonacci)
        1   0.000   0.000 {method 'disable' of '_lsprof.Profiler'}
```

As you can see, the profiler still makes the code about twice as slow, but it's definitely usable.

Using profile statistics

To get some more intricate profiling results, we will profile the `pystone` script. The `pystone` script is an internal Python performance test which benchmarks the Python interpreter fairly thoroughly. First, let's create the statistics using this script:

```
from test import pystone
import cProfile


if __name__ == '__main__':
    profiler = cProfile.Profile()
    profiler.runcall(pystone.main)
    profiler.dump_stats('pystone.profile')
```

When executing the script, you should get something like this:

```
# python3 test_pystone.py
Pystone(1.2) time for 50000 passes = 0.725432
This machine benchmarks at 68924.4 pystones/second
```

After running the script, you should have a `pystone.profile` file containing the profiling results. These results can be And the `pystone.profile` file which contains all of the profiling statistics. These statistics can be viewed through the `pstats` module which is bundled with Python:

```
import pstats


stats = pstats.Stats('pystone.profile')
stats.strip_dirs()
stats.sort_stats('calls', 'cumtime')
stats.print_stats(10)
```

In some cases, it can be interesting to combine the results from multiple measurements. That is possible by specifying multiple files or by using `stats.add(*filenames)`. But first, let's look at the regular output:

```
# python3 parse_statistics.py


         1050012 function calls in 0.776 seconds
```

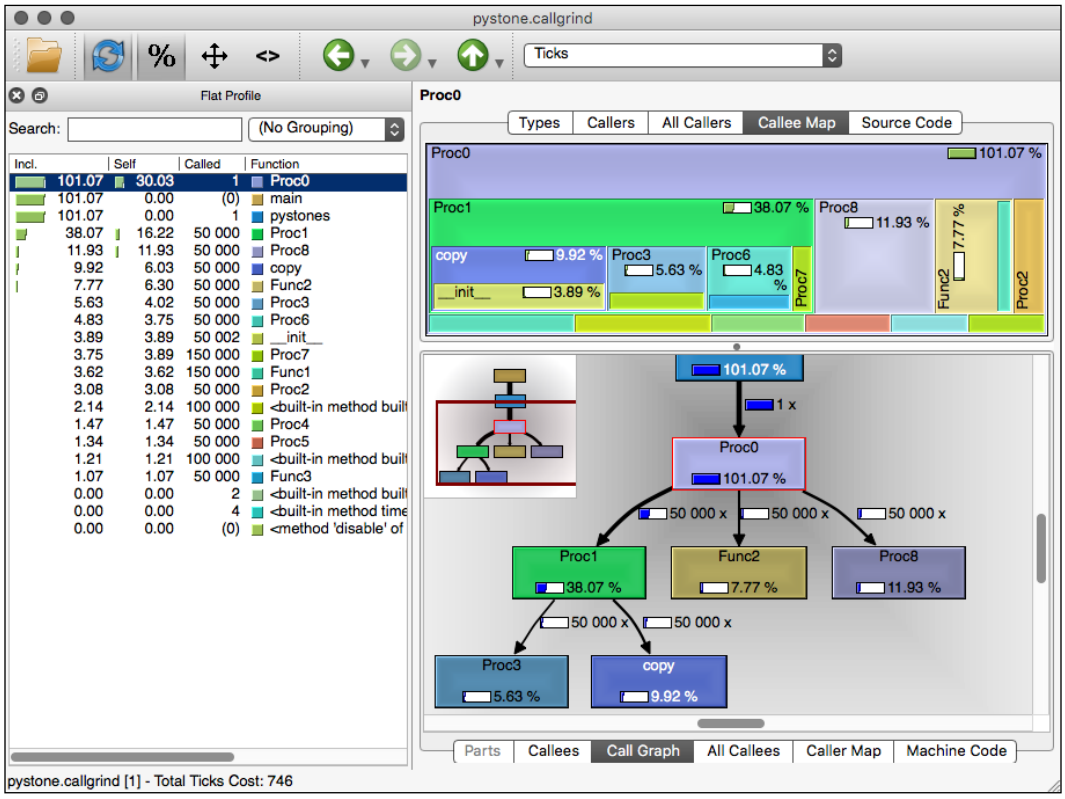
```
   Ordered by: call count, cumulative time
   List reduced from 21 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   150000    0.032    0.000    0.032    0.000 pystone.py:214(Proc7)
   150000    0.027    0.000    0.027    0.000 pystone.py:232(Func1)
   100000    0.016    0.000    0.016    0.000 {built-in method builtins.
chr}
   100000    0.010    0.000    0.010    0.000 {built-in method builtins.
ord}
    50002    0.029    0.000    0.029    0.000 pystone.py:52(__init__)
    50000    0.127    0.000    0.294    0.000 pystone.py:144(Proc1)
    50000    0.094    0.000    0.094    0.000 pystone.py:219(Proc8)
    50000    0.048    0.000    0.077    0.000 pystone.py:60(copy)
    50000    0.051    0.000    0.061    0.000 pystone.py:240(Func2)
    50000    0.031    0.000    0.043    0.000 pystone.py:171(Proc3)
```

Obviously, the parameters can easily be modified to change the sort order and the number of output lines. But that is not the only possibility of the statistics. There are quite a few packages around which can parse these results and visualize them. One option is RunSnakeRun, which although useful does not run on Python 3 currently. Also, we have QCacheGrind, a very nice visualizer for profile statistics but which requires some manual compiling to get running or some searching for binaries of course.

Let's look at the output from QCacheGrind. In the case of Windows, the QCacheGrindWin package provides a binary, whereas within Linux it is most likely available through your package manager, and with OS X you can try `brew install qcachegrind --with-graphviz`. But there is one more package you will require: the `pyprof2calltree` package. It transforms the `profile` output into a format that QCacheGrind understands. So, after a simple `pip install pyprof2calltree`, we can now convert the `profile` file into a `callgrind` file:

```
# pyprof2calltree -i pystone.profile -o pystone.callgrind
writing converted data to: pystone.callgrind
# qcachegrind pystone.callgrind
```

This results in running of the `QCacheGrind` application. After switching to the appropriate tabs, you should see something like the following image:



For a simple script such as this, pretty much all output works. However, with full applications, a tool such as QCacheGrind is invaluable. Looking at the output generated by QCacheGrind, it is immediately obvious which process took the most time. The structure at the top right shows bigger rectangles if the amount of time taken was greater, which is a very useful visualization of the chunks of CPU time that were used. The list at the left is very similar to `cProfile` and therefore nothing new. The tree at the bottom right can be very valuable or very useless as it is in this case. It shows you the percentage of CPU time taken in a function and more importantly, the relationship of that function with the other functions.

Because these tools scale depending on the input the results are useful for just about any application. Whether a function takes 100 milliseconds or 100 minutes makes no difference, the output will show a clear overview of the slow parts, which is what we will try to fix.

Line profiler

`line_profiler` is actually not a package that's bundled with Python, but it's far too useful to ignore. While the regular `profile` module profiles all (sub)functions within a certain block, `line_profiler` allows for profiling line per line within a function. The Fibonacci function is not best suited here, but we can use a prime number generator instead. But first, install `line_profiler`:

```
pip install line_profiler
```

Now that we have installed the `line_profiler` module (and with that the `kernprof` command), let's test `line_profiler`:

```
import itertools


@profile
def primes():
    n = 2
    primes = set()
    while True:
        for p in primes:
            if n % p == 0:
                break
        else:
            primes.add(n)
            yield n
        n += 1


if __name__ == '__main__':
    total = 0
    n = 2000
    for prime in itertools.islice(primes(), n):
        total += prime

    print('The sum of the first %d primes is %d' % (n, total))
```

You might be wondering where the `profile` decorator is coming from. It originates from the `line_profiler` module, which is why we have to run the script with the `kernprof` command:

```
# kernprof -l test_primes.py
The sum of the first 2000 primes is 16274627
Wrote profile results to test_primes.py.lprof
```

As the command says, the results have been written to the `test_primes.py.lprof` file. So let's look at the output of that with the `Time` column skipped for readability:

```
# python3 -m line_profiler test_primes.py.lprof
Timer unit: 1e-06 s

Total time: 2.33179 s
File: test_primes.py
Function: primes at line 4

Line #      Hits   Per Hit   % Time  Line Contents
================================================
     4                                @profile
     5                                def primes():
     6         1       3.0      0.0      n = 2
     7         1       1.0      0.0      primes = set()
     8         1       0.0      0.0      while True:
     9   2058163       0.5     43.1          for p in primes:
    10   2056163       0.6     56.0              if n % p == 0:
    11     15388       0.5      0.3                  break
    12                                        else:
    13      2000       1.2      0.1              primes.add(n)
    14      2000       0.5      0.0              yield n
    15     17387       0.6      0.4          n += 1
```

Wonderful output, isn't it? It makes it trivial to find the slow part within a bit of code. Within this code, the slowness is obviously originating from the loop, but within other code it might not be that clear.

This module can be added as an IPython extension as well, which enables the `%lprun` command within IPython. To load the extension, the `load_ext` command can be used from the IPython shell `%load_ext line_profiler`.

Improving performance

Much can be said about performance optimization, but truthfully, if you have read the entire book up to this point, you know most of the Python-specific techniques to write fast code. The most important factor in application performance will always be the choice of algorithms, and by extension, the data structures. Searching for an item within `list` is almost always a worse idea than searching for an item in `dict` or `set`.

Using the right algorithm

Within any application, the right choice of algorithm is by far the most important performance characteristic, which is why I am repeating it to illustrate the results of a bad choice:

```
In [1]: a = list(range(1000000))

In [2]: b = dict.fromkeys(range(1000000))

In [3]: %timeit 'x' in a
10 loops, best of 3: 20.5 ms per loop

In [4]: %timeit 'x' in b
10000000 loops, best of 3: 41.6 ns per loop
```

Checking whether an item is within a `list` is an `O(n)` operation and checking whether an item is within a `dict` is an `O(1)` operation. A huge difference when `n=1000000` obviously, in this simple test we can see that for 1 million items it's 500 times faster.

All other performance tips combined together might make your code twice as fast, but using the right algorithm for the job can cause a much greater improvement. Using an algorithm that takes `O(n)` time instead of `O(n^2)` time will make your code `1000` times faster for `n=1000`, and with a larger `n` the difference only grows further.

Global interpreter lock

One of the most obscure components of the CPython interpreter is the **global interpreter lock** (**GIL**), a **mutual exclusion lock** (**mutex**) required to prevent memory corruption. The Python memory manager is not thread-safe and that is why the GIL is needed. Without the GIL, multiple threads might alter memory at the same time, causing all sorts of unexpected and potentially dangerous results.

So what is the impact of the GIL in a real-life application? Within single-threaded applications it makes no difference whatsoever and is actually an extremely fast method for memory consistency. Within multithreaded applications however, it can slow your application down a bit, because only a single thread can access the GIL at a time. So if your code has to access the GIL a lot, it might benefit from some restructuring.

Luckily, Python offers a few other options for parallel processing: the `asyncio` module that we saw earlier and the `multiprocessing` library that we will see in *Chapter 13*, *Multiprocessing – When a Single CPU Core Is Not Enough*.

Try versus if

In many languages a `try/except` type of block incurs quite a performance hit, but within Python this is not the case. It's not that an `if` statement is heavy, but if you expect your `try/except` to succeed most of the time and only fail in rare cases, it's definitely a valid alternative. As always though, focus on readability and conveying the purpose of the code. If the intention of the code is clearer using an `if` statement, use the `if` statement. If `try/except` conveys the intention in a better way, use that.

Lists versus generators

Evaluating code lazily using generators is almost always a better idea than calculating the entire dataset. The most important rule of performance optimization is probably that you shouldn't calculate anything you're not going to use. If you're not sure that you are going to need it, don't calculate it.

Don't forget that you can easily chain multiple generators, so everything is calculated only when it's actually needed. Do be careful that this won't result in recalculation though; `itertools.tee` is generally a better idea than recalculating your results completely.

String concatenation

You might have seen benchmarks saying that using `+=` is much slower than joining strings. At one point this made quite a lot of difference indeed. With Python 3 however, most of the differences have vanished.

```
In [1]: %%timeit
   ...: s = ''
   ...: for i in range(1000000):
   ...:     s += str(i)
```

```
   ...:
1 loops, best of 3: 362 ms per loop

In [2]: %%timeit
   ...: ss = []
   ...: for i in range(1000000):
   ...:     ss.append(str(i))
   ...: s = ''.join(ss)
   ...:
1 loops, best of 3: 332 ms per loop

In [3]: %timeit ''.join(str(i) for i in range(1000000))
1 loops, best of 3: 324 ms per loop

In [4]: %timeit ''.join([str(i) for i in range(1000000)])
1 loops, best of 3: 294 ms per loop
```

There are still some differences of course, but they are so small that I recommend to simply ignore them and choose the most readable option instead.

Addition versus generators

As is the case with string concatenation, once a significant difference now too small to mention.

```
In [1]: %%timeit
   ...: x = 0
   ...: for i in range(1000000):
   ...:     x += i
   ...:
10 loops, best of 3: 73.2 ms per loop

In [2]: %timeit x = sum(i for i in range(1000000))
10 loops, best of 3: 75.3 ms per loop

In [3]: %timeit x = sum([i for i in range(1000000)])
10 loops, best of 3: 71.2 ms per loop

In [4]: %timeit x = sum(range(1000000))
10 loops, best of 3: 25.6 ms per loop
```

What does help though is letting Python handle everything internally using native functions, as can be seen in the last example.

Map versus generators and list comprehensions

Once again, readability counts more than performance. There are a few cases where `map` is faster than list comprehensions and generators, but only if the `map` function can use a predefined function. As soon as you need to whip out `lambda`, it's actually slower. Not that it matters much, since readability should be key anyhow, use generators or list comprehensions instead of `map`:

```
In [1]: %timeit list(map(lambda x: x/2, range(1000000)))
10 loops, best of 3: 182 ms per loop

In [2]: %timeit list(x/2 for x in range(1000000))
10 loops, best of 3: 122 ms per loop

In [3]: %timeit [x/2 for x in range(1000000)]
10 loops, best of 3: 84.7 ms per loop
```

As you can see, the list comprehension is obviously quite a bit faster than the generator. In many cases I would still recommend the generator over the list comprehension though, if only because of the memory usage and the potential laziness. If for some reason you are only going to use the first 10 items, you're still wasting a lot of resources by calculating the full list of items.

Caching

We have already covered the `functools.lru_cache` decorator in *Chapter 5*, *Decorators – Enabling Code Reuse by Decorating* but the importance should not be underestimated. Regardless of how fast and smart your code is, not having to calculate results is always better and that's what caching does. Depending on your use case, there are many options available. Within a simple script, `functools.lru_cache` is a very good contender, but between multiple executions of an application, the `cPickle` module can be a life saver as well.

If multiple servers are involved, I recommend taking a look at **Redis**. The Redis server is a single threaded in-memory server which is extremely fast and has many useful data structures available. If you see articles or tutorials about improving performance using Memcached, simply replace Memcached with Redis everywhere. Redis is superior to Memcached in every way and in its most basic form the API is compatible.

Lazy imports

A common problem in application load times is that everything is loaded immediately at the start of the program, while with many applications this is actually not needed and certain parts of the application only require loading when they are actually used. To facilitate this, one can occasionally move the imports inside of functions so they can be loaded on demand.

While it's a valid strategy in some cases, I don't generally recommend it for two reasons:

1. It makes your code less clear; having all imports in the same style at the top of the file improves readability.
2. It doesn't make the code faster as it just moves the load time to a different part.

Using optimized libraries

This is actually a very broad tip, but useful nonetheless. If there's a highly optimized library which suits your purpose, you most likely won't be able to beat its performance without a significant amount of effort. Libraries such as `numpy`, `pandas`, `scipy`, and `sklearn` are highly optimized for performance and their native operations can be incredibly fast. If they suit your purpose, be sure to give them a try. Just to illustrate how fast `numpy` can be compared to plain Python, refer to the following:

```
In [1]: import numpy

In [2]: a = list(range(1000000))

In [3]: b = numpy.arange(1000000)

In [4]: %timeit c = [x for x in a if x > 500000]
10 loops, best of 3: 44 ms per loop

In [5]: %timeit d = b[b > 500000]
1000 loops, best of 3: 1.61 ms per loop
```

The `numpy` code does exactly the same as the Python code, except that it uses `numpy` arrays instead of Python lists. This little difference has made the code more than 25 times faster.

Just-in-time compiling

Just-in-time (**JIT**) compiling is a method of dynamically compiling (parts) an application during runtime. Because there is much more information available at runtime, this can have a huge effect and make your application much faster.

The `numba` package provides selective JIT compiling for you, allowing you to mark the functions that are JIT compiler compatible. Essentially, if your functions follow the functional programming paradigm of basing the calculations only on the input, then it will most likely work with the JIT compiler.

Basic example of how the `numba` JIT compiler can be used:

```
import numba


@numba.jit
def sum(array):
    total = 0.0
    for value in array:
        total += value
    return value
```

The use cases for these are limited, but if you are using `numpy` or pandas you will most likely benefit from `numba`.

Another very interesting fact to note is that `numba` supports not only CPU optimized execution but GPU as well. This means that for certain operations you can use the fast processor in your video card to process the results.

Converting parts of your code to C

We will see more about this in *Chapter 14*, *Extensions in C/C++, System Calls, and C/C++ Libraries*, but if high performance is really required, then a native C function can help quite a lot. This doesn't even have to be that difficult. The Cython module makes it trivial to write parts of your code with performance very close to native C code.

Following is an example from the Cython manual to approximate the value of pi:

```
cdef inline double recip_square(int i):
    return 1./(i*i)

def approx_pi(int n=10000000):
    cdef double val = 0.
    cdef int k
```

```
    for k in xrange(1,n+1):
        val += recip_square(k)
    return (6 * val)**.5
```

While there are some small differences such as `cdef` instead of `def` and type definitions for the values and parameters, the code is largely the same as regular Python would be, but certainly much faster.

Memory usage

So far we have simply looked at the execution times and ignored the memory usage of the scripts. In many cases, the execution times are the most important, but memory usage should not be ignored. In almost all cases, CPU and memory are traded; a code either uses a lot of CPU or a lot of memory, which means that both do matter a lot.

Tracemalloc

Monitoring memory usage used to be something that was only possible through external Python modules such as **Dowser** or **Heapy**. While those modules still work, they are largely obsolete now because of the `tracemalloc` module. Let's give the `tracemalloc` module a try to see how easy memory usage monitoring is nowadays:

```
import tracemalloc


if __name__ == '__main__':
    tracemalloc.start()

    # Reserve some memory
    x = list(range(1000000))

    # Import some modules
    import os
    import sys
    import asyncio

    # Take a snapshot to calculate the memory usage
    snapshot = tracemalloc.take_snapshot()
    for statistic in snapshot.statistics('lineno')[:10]:
        print(statistic)
```

This results in:

```
# python3 test_tracemalloc.py
test_tracemalloc.py:8: size=35.3 MiB, count=999745, average=37 B
<frozen importlib._bootstrap_external>:473: size=1909 KiB, count=20212,
average=97 B
<frozen importlib._bootstrap>:222: size=895 KiB, count=3798, average=241
B
collections/__init__.py:412: size=103 KiB, count=1451, average=72 B
<string>:5: size=36.6 KiB, count=133, average=282 B
collections/__init__.py:406: size=29.9 KiB, count=15, average=2039 B
abc.py:133: size=26.1 KiB, count=102, average=262 B
ipaddress.py:608: size=21.3 KiB, count=182, average=120 B
<frozen importlib._bootstrap_external>:53: size=21.2 KiB, count=140,
average=155 B
types.py:234: size=15.3 KiB, count=124, average=127 B
```

You can easily see how every part of the code allocated memory and where it might be wasted. While it might still be unclear which part was actually causing the memory usage, there are options for that as well, as we will see in the following sections.

Memory profiler

The `memory_profiler` module is very similar to `line_profiler` discussed earlier, but for memory usage instead. Installing it is as easy as `pip install memory_profiler`, but the optional `pip install psutil` is also highly recommended (and required in the case of Windows) as it increases your performance by a large amount. To test `line_profiler`, we will use the following script:

```
import memory_profiler


@memory_profiler.profile
def main():
    n = 100000
    a = [i for i in range(n)]
    b = [i for i in range(n)]
    c = list(range(n))
    d = list(range(n))
    e = dict.fromkeys(a, b)
    f = dict.fromkeys(c, d)


if __name__ == '__main__':
    main()
```

Note that we actually import the `memory_profiler` here although that is not strictly required. It can also be executed through `python3 -m memory_profiler your_scripts.py`:

```
# python3 test_memory_profiler.py
Filename: test_memory_profiler.py

Line #    Mem usage    Increment   Line Contents
================================================
     4     11.0 MiB      0.0 MiB   @memory_profiler.profile
     5                             def main():
     6     11.0 MiB      0.0 MiB       n = 100000
     7     14.6 MiB      3.5 MiB       a = [i for i in range(n)]
     8     17.8 MiB      3.2 MiB       b = [i for i in range(n)]
     9     21.7 MiB      3.9 MiB       c = list(range(n))
    10     25.5 MiB      3.9 MiB       d = list(range(n))
    11     38.0 MiB     12.5 MiB       e = dict.fromkeys(a, b)
    12     44.1 MiB      6.1 MiB       f = dict.fromkeys(c, d)
```

Even though everything runs as expected, you might be wondering about the varying amounts of memory used by the lines of code here. Why does `a` take `3.5 MiB` and `b` only `3.2 MiB`? This is caused by the Python memory allocation code; it reserves memory in larger blocks, which is subdivided and reused internally. Another problem is that `memory_profiler` takes snapshots internally, which results in memory being attributed to the wrong variables in some cases. The variations should be small enough to not make a large difference in the end, but some changes are to be expected.

This module can be added as an IPython extension as well, which enables the `%mprun` command within IPython. To load the extension, the `load_ext` command can be used from the IPython shell `%load_ext memory_profiler`. Another very useful command is `%memit` which is the memory equivalent of the `%timeit` command.

Memory leaks

The usage of these modules will generally be limited to the search for memory leaks. Especially the `tracemalloc` module has a few features to make that fairly easy. The Python memory management system is fairly simple; it just has a simple reference counter to see if an object is used. While this works great in most cases, it can easily introduce memory leaks when circular references are involved. The basic premise of a memory leak with leak detection code looks like this:

```
 1 import tracemalloc
 2
 3
 4 class Spam(object):
 5     index = 0
 6     cache = {}
 7
 8     def __init__(self):
 9         Spam.index += 1
10         self.cache[Spam.index] = self
11
12
13 class Eggs(object):
14     eggs = []
15
16     def __init__(self):
17         self.eggs.append(self)
18
19
20 if __name__ == '__main__':
21     # Initialize some variables to ignore them from the leak
22     # detection
23     n = 200000
24     spam = Spam()
25
26     tracemalloc.start()
27     # Your application should initialize here
28
29     snapshot_a = tracemalloc.take_snapshot()
30     # This code should be the memory leaking part
31     for i in range(n):
32         Spam()
33
34     Spam.cache = {}
35     snapshot_b = tracemalloc.take_snapshot()
```

```
    # And optionally more leaking code here
    for i in range(n):
        a = Eggs()
        b = Eggs()
        a.b = b
        b.a = a

    Eggs.eggs = []
    snapshot_c = tracemalloc.take_snapshot()

    print('The first leak:')
    statistics = snapshot_b.compare_to(snapshot_a, 'lineno')
    for statistic in statistics[:10]:
        print(statistic)

    print('\nThe second leak:')
    statistics = snapshot_c.compare_to(snapshot_b, 'lineno')
    for statistic in statistics[:10]:
        print(statistic)
```

Let's see how bad this code is actually leaking:

```
# python3 test_leak.py
The first leak:
tracemalloc.py:349: size=528 B (+528 B), count=3 (+3), average=176 B
test_leak.py:34: size=288 B (+288 B), count=2 (+2), average=144 B
test_leak.py:32: size=120 B (+120 B), count=2 (+2), average=60 B
tracemalloc.py:485: size=64 B (+64 B), count=1 (+1), average=64 B
tracemalloc.py:487: size=56 B (+56 B), count=1 (+1), average=56 B
tracemalloc.py:277: size=32 B (+32 B), count=1 (+1), average=32 B
test_leak.py:31: size=28 B (+28 B), count=1 (+1), average=28 B
test_leak.py:9: size=28 B (+28 B), count=1 (+1), average=28 B

The second leak:
test_leak.py:41: size=18.3 MiB (+18.3 MiB), count=400000 (+400000),
average=48 B
test_leak.py:40: size=18.3 MiB (+18.3 MiB), count=400000 (+400000),
average=48 B
test_leak.py:38: size=10.7 MiB (+10.7 MiB), count=200001 (+200001),
average=56 B
test_leak.py:39: size=10.7 MiB (+10.7 MiB), count=200002 (+200002),
average=56 B
```

```
tracemalloc.py:349: size=680 B (+152 B), count=6 (+3), average=113 B
test_leak.py:17: size=72 B (+72 B), count=1 (+1), average=72 B
test_leak.py:43: size=64 B (+64 B), count=1 (+1), average=64 B
test_leak.py:32: size=56 B (-64 B), count=1 (-1), average=56 B
tracemalloc.py:487: size=112 B (+56 B), count=2 (+1), average=56 B
tracemalloc.py:277: size=64 B (+32 B), count=2 (+1), average=32 B
```

In absolute memory usage the increase is not even that great, but it is definitely leaking a little. The first leak is negligible; at the last iteration we see an increase of 28 bytes which is next to nothing. The second leak however leaks a lot and peaks at a 18.3 megabyte increase. Those are memory leaks, the Python garbage collector (`gc`) is smart enough to clean circular references eventually but it won't clean them until a certain limit is reached. More about that soon.

Whenever you want to have a circular reference that does not cause memory leaks, the `weakref` module is available. It creates reference which don't count towards the object reference count. Before we look at the `weakref` module, let's take a look at the object references themselves through the eyes of the Python garbage collector (`gc`):

```
import gc


class Eggs(object):

    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return '<%s: %s>' % (self.__class__.__name__, self.name)


# Create the objects
a = Eggs('a')
b = Eggs('b')

# Add some circular references
a.b = a
b.a = b

# Remove the objects
del a
del b

# See if the objects are still there
```

```
print('Before manual collection:')
for object_ in gc.get_objects():
    if isinstance(object_, Eggs):
        print('\t', object_, gc.get_referents(object_))

print('After manual collection:')
gc.collect()
for object_ in gc.get_objects():
    if isinstance(object_, Eggs):
        print('\t', object_, gc.get_referents(object_))

print('Thresholds:', gc.get_threshold())
```

So let's have a look at the output:

```
# python3 test_refcount.py
Before manual collection:
        <Eggs: a> [{'b': <Eggs: a>, 'name': 'a'}, <class '__main__.
Eggs'>]
        <Eggs: b> [{'name': 'b', 'a': <Eggs: b>}, <class '__main__.
Eggs'>]
After manual collection:
Thresholds: (700, 10, 10)
```

As we can see here, until we manually call the garbage collector, the `Eggs` objects will stay in the memory. Even after explicitly deleting the objects. So does this mean you are always required to manually call `gc.collect()` to remove these references? Luckily that's not needed, as the Python garbage collector will automatically collect once the thresholds have been reached. By default, the thresholds for the Python garbage collector are set to `700, 10, 10` for the three generations of collected objects. The collector keeps track of all the memory allocations and deallocations in Python, and as soon as the number of allocations minus the number of deallocations reaches 700, the object is either removed if it's not referenced anymore or it is moved to the next generation if it still has a reference. The same is repeated for generation 2 and 3, albeit with the lower thresholds of 10.

This begs the question: where and when is it useful to manually call the garbage collector? Since the Python memory allocator reuses blocks of memory and only rarely releases it, for long running scripts the garbage collector can be very useful. That's exactly where I recommend its usage: long running scripts in memory-strapped environments and specifically, right before you allocate a large amount of memory.

More to the point however, the `gc` module can help you a lot when looking for memory leaks as well. The `tracemalloc` module can show you the parts that take the most memory in bytes but the `gc` module can help you find the most defined objects. Just be careful with setting the garbage collector debug settings such as `gc.set_debug(gc.DEBUG_LEAK)`; it returns a large amount of output even if you don't reserve any memory yourself. Revisiting our `Spam` and `Eggs` script from earlier, let's see where and how the memory is being used using the garbage collection module:

```
import gc
import collections


class Spam(object):
    index = 0
    cache = {}

    def __init__(self):
        Spam.index += 1
        self.cache[Spam.index] = self


class Eggs(object):
    eggs = []

    def __init__(self):
        self.eggs.append(self)


if __name__ == '__main__':
    n = 200000
    for i in range(n):
        Spam()

    for i in range(n):
        a = Eggs()
        b = Eggs()
        a.b = b
        b.a = a

    Spam.cache = {}
    Eggs.eggs = []
    objects = collections.Counter()
    for object_ in gc.get_objects():
        objects[type(object_)] += 1

    for object_, count in objects.most_common(5):
        print('%d: %s' % (count, object_))
```

The output is probably close to what you were already expecting:

```
# python3 test_leak.py
400617: <class 'dict'>
400000: <class '__main__.Eggs'>
962: <class 'wrapper_descriptor'>
920: <class 'function'>
625: <class 'method_descriptor'>
```

The large amount of `dict` objects is because of the internal state of the classes, but beyond that we simply see the `Eggs` objects just as we would expect. The `Spam` objects were properly removed by the garbage collector because they and all of the references were just removed. The `Eggs` objects couldn't be removed because of the circular references. Now we will repeat the same example using the `weakref` module to see if it makes a difference:

```
import gc
import weakref
import collections


class Eggs(object):
    eggs = []

    def __init__(self):
        self.eggs.append(self)


if __name__ == '__main__':
    n = 200000
    for i in range(n):
        a = Eggs()
        b = Eggs()
        a.b = weakref.ref(b)
        b.a = weakref.ref(a)

    Eggs.eggs = []
    objects = collections.Counter()
    for object_ in gc.get_objects():
        objects[type(object_)] += 1

    for object_, count in objects.most_common(5):
        print('%d: %s' % (count, object_))
```

Now let's see what remained this time:

```
# python3 test_leak.py
962: <class 'wrapper_descriptor'>
919: <class 'function'>
625: <class 'method_descriptor'>
618: <class 'dict'>
535: <class 'builtin_function_or_method'>
```

Nothing besides some standard built-in Python objects, which is exactly what we had hoped for. Be careful with weak references though, as they can easily blow up in your face if the referenced object has disappeared:

```
import weakref


class Eggs(object):
    pass


if __name__ == '__main__':
    a = Eggs()
    b = Eggs()
    a.b = weakref.ref(b)

    print(a.b())
    del b
    print(a.b())
```

This results in one working reference and a dead one:

```
# python3 test_weakref.py
<__main__.Eggs object at 0x104891a20>
None
```

Reducing memory usage

In general, memory usage probably won't be your biggest problem in Python, but it can still be useful to know what you can do to reduce memory usage. When trying to reduce memory usage, it's important to understand how Python allocates memory.

There are four concepts which you need to know about within the Python memory manager:

- First we have the heap. The heap is the collection of all Python managed memory. Note that this is separate from the regular heap and mixing the two could result in corrupt memory and crashes.
- Second are the arenas. These are the chunks that Python requests from the system. These chunks have a fixed size of 256 KiB each and they are the objects that make up the heap.
- Third we have the pools. These are the chunks of memory that make up the arenas. These chunks are 4 KiB each. Since the pools and arenas have fixed sizes, they are simple arrays.
- Fourth and last, we have the blocks. The Python objects get stored within these and every block has a specific format depending on the data type. Since an integer takes up more space than a character, for efficiency a different block size is used.

Now that we know how the memory is allocated, we can also understand how it can be returned to the operating system. Whenever an arena is completely empty, it can and will be freed. To increase the likelihood of this happening, some heuristics are used to maximize the usage of fuller arenas.

It is important to note that the regular heap and Python heap are maintained separately as mixing them can result in corruption and/or crashing of applications. Unless you write your own extensions, you will probably never have to worry about manual memory allocation though.

Generators versus lists

The most important tip is to use generators whenever possible. Python 3 has come a long way in replacing lists with generators already, but it really pays off to keep that in mind as it saves not only memory, but CPU as well when not all of that memory needs to be kept at the same time.

To illustrate the difference:

```
Line #    Mem usage    Increment   Line Contents
================================================
     4     11.0 MiB      0.0 MiB   @memory_profiler.profile
     5                             def main():
     6     11.0 MiB      0.0 MiB    a = range(1000000)
     7     49.7 MiB     38.6 MiB    b = list(range(1000000))
```

The `range()` generator takes such little memory that it doesn't even register, whereas the list of numbers takes `38.6 MiB`.

Recreating collections versus removing items

One very important detail about collections in Python is that many of them can only grow; they won't just shrink by themselves. To illustrate:

```
Line #    Mem usage    Increment   Line Contents
================================================
     4     11.5 MiB      0.0 MiB   @memory_profiler.profile
     5                             def main():
     6                             # Generate a huge dict
     7     26.3 MiB     14.8 MiB   a = dict.fromkeys(range(100000))
     8
     9                             # Remove all items
    10     26.3 MiB      0.0 MiB   for k in list(a.keys()):
    11     26.3 MiB      0.0 MiB   del a[k]
    12
    13                             # Recreate the dict
    14     23.6 MiB     -2.8 MiB   a = dict((k, v) for k, v in a.items())
```

This is one of the most common memory usage mistakes made with lists and dictionaries. Besides recreating the objects, there is, of course, also the option of using generators instead so the memory is never allocated at all.

Using slots

If you've used Python for a long time you may have seen the `__slots__` feature of classes. It allows you to specify which fields you want to store in a class and it skips all the others by not implementing `instance.__dict__`. While this method does save a little bit of memory in your class definitions, I recommend against its usage as there are several downsides to using it. The most important one is that they make inheritance non-obvious (adding `__slots__` to a subclassed class that doesn't have `__slots__` has no effect). It also makes it impossible to modify class attributes on the fly and breaks `weakref` by default. And lastly, classes with slots cannot be pickled without defining a `__getstate__` function.

For completeness however, here's a demonstration of the slots feature and the difference in memory usage:

```
import memory_profiler


class Slots(object):
    __slots__ = 'index', 'name', 'description'

    def __init__(self, index):
        self.index = index
        self.name = 'slot %d' % index
        self.description = 'some slot with index %d' % index


class NoSlots(object):

    def __init__(self, index):
        self.index = index
        self.name = 'slot %d' % index
        self.description = 'some slot with index %d' % index


@memory_profiler.profile
def main():
    slots = [Slots(i) for i in range(25000)]
    no_slots = [NoSlots(i) for i in range(25000)]
    return slots, no_slots


if __name__ == '__main__':
    main()
```

And the memory usage:

```
# python3 test_slots.py
Filename: test_slots.py

Line #    Mem usage    Increment   Line Contents
================================================
    21     11.1 MiB      0.0 MiB   @memory_profiler.profile
    22                             def main():
```

```
    23     17.0 MiB      5.9 MiB   slots = [Slots(i) for i in
range(25000)]
    24     25.0 MiB      8.0 MiB   no_slots = [NoSlots(i) for i in
range(25000)]
    25     25.0 MiB      0.0 MiB   return slots, no_slots
```

You might argue that this is not a fair comparison, since they both store a lot of data which skews the results. And you would indeed be right, because the "bare" comparison storing only `index` and nothing else gives `2 MiB` versus `4.5 MiB`. But let's be honest, if you're not going to store data, then what's the point in creating class instances? That's why I recommend against the usage of `__slots__` and instead recommend the usage of tuples or `collections.namedtuple` if memory is that important. There is one more structure that's even more memory efficient, the `array` module. It stores the data in pretty much a bare memory array. Note that this is generally slower than lists and much less convenient to use.

Performance monitoring

So far we have seen how to measure and improve both CPU and memory performance, but there is one part we have completely skipped over. Performance changes due to external factors such as growing amounts of data are very hard to predict. In real life applications, bottlenecks aren't constant. They change all the time and code that was once extremely fast might bog down as soon as more load is applied.

Because of that I recommend implementing a monitoring solution that tracks the performance of anything and everything over time. The big problem with performance monitoring is that you can't know what will slow down in the future and what the cause is going to be. I've even had websites slow down because of Memcached and Redis calls. These are memory only caching servers that respond well within a millisecond which makes slowdowns highly unlikely, until you do over a 100 cache calls and the latency towards the cache server increases from 0.1 milliseconds to 2 milliseconds, and all of a sudden those 100 calls take 200 milliseconds instead of 10 milliseconds. Even though 200 milliseconds still sounds like very little, if your total page load time is generally below 100 milliseconds, that is all of a sudden an enormous increase and definitely noticeable.

To monitor performance and to be able to track changes over time and find the responsible components, I am personally a big fan of the Statsd statistic collection server together with the Graphite interface. Even though usability is a bit lacking, the result is a graphing interface which you can dynamically query to analyze when, where, and how your performance changed. To be able to use these you will have to send the metrics from your application towards the Statsd server. To do just that, I have written the Python-Statsd (`https://pypi.python.org/pypi/python-statsd`) and Django-Statsd (`https://pypi.python.org/pypi/django-statsd`) packages. These packages allow you to monitor your application from beginning to end, and in the case of Django you will be able to monitor your performance per application or view and within those see all of the components, such as the database, template, and caching layers. This way, you know exactly what is causing the slowdowns in your website (or application).

Summary

When it comes to performance, there is no holy grail, no single thing you can do to ensure peak performance in all cases. This shouldn't worry you however, as in most cases you will never need to tune the performance and if you do, a single tweak could probably fix your problem. You should be able to find performance problems and memory leaks in your code now which is what matters most, so just try to contain yourself and only tweak when it's actually needed.

The most important outtakes from this chapter are:

- Test before you invest any effort. Making some functions faster seems like a great achievement but it is only rarely needed.
- Choosing the correct data structure/algorithm is much more effective than any other performance optimization.
- Circular references drain the memory until the garbage collector starts cleaning.
- Slots are not worth the effort.

The next chapter will discuss multiprocessing, a library which makes it trivial to employ multiple processors for your scripts. If you can't squeeze any more performance out of your script, multiprocessing might be your answer, as every (remote?) CPU core can make your script faster.

13
Multiprocessing – When a Single CPU Core Is Not Enough

In the previous chapter, we discussed factors that influence performance and some methods to increase performance. This chapter can actually be seen as an extension to the list of performance tips. In this chapter, we will discuss the multiprocessing module, a module that makes it very easy to make your code run on multiple CPU cores and even on multiple machines. This is an easy way to work around the **Global Interpreter Lock** (**GIL**) that was discussed in the previous chapter.

To summarize, this chapter will cover:

- Local multiprocessing
- Remote multiprocessing
- Data sharing and synchronization between processes

Multithreading versus multiprocessing

Within this book we haven't really covered multithreading yet, but you have probably seen multithreaded code in the past. The big difference between multithreading and multiprocessing is that with multithreading everything is still executed within a single process. That effectively limits your performance to a single CPU core. It actually limits you even further because the code has to deal with the GIL limitations of CPython.

The GIL is the global lock that Python uses for safe memory access. It is discussed in more detail in *Chapter 12, Performance – Tracking and Reducing Your Memory and CPU Usage*, about performance.

To illustrate that multithreading code doesn't help performance in all cases and can actually be slightly slower than single threaded code, look at this example:

```
import datetime
import threading


def busy_wait(n):
    while n > 0:
        n -= 1


if __name__ == '__main__':
    n = 10000000
    start = datetime.datetime.now()
    for _ in range(4):
        busy_wait(n)
    end = datetime.datetime.now()
    print('The single threaded loops took: %s' % (end - start))

    start = datetime.datetime.now()
    threads = []
    for _ in range(4):
        thread = threading.Thread(target=busy_wait, args=(n,))
        thread.start()
        threads.append(thread)

    for thread in threads:
        thread.join()

    end = datetime.datetime.now()
    print('The multithreaded loops took: %s' % (end - start))
```

With Python 3.5, which has the new and improved GIL implementation (introduced in Python 3.2), the performance is quite comparable but there is no improvement:

```
# python3 test_multithreading.py
The single threaded loops took: 0:00:02.623443
The multithreaded loops took: 0:00:02.597900
```

With Python 2.7, which still has the old GIL, the performance is a lot better in the single threaded variant:

```
# python2 test_multithreading.py
The single threaded loops took: 0:00:02.010967
The multithreaded loops took: 0:00:03.924950
```

From this test we can conclude that Python 2 is faster in some cases while Python 3 is faster in other cases. What you should take from this is that there is no performance reason to choose between Python 2 or Python 3 specifically. Just note that Python 3 is at least as fast as Python 2 in most cases and if that is not the case, it will be fixed soon.

Regardless, for CPU-bound operations, threading does not offer any performance benefit since it executes on a single processor core. For I/O bound operations however, the `threading` library does offer a clear benefit, but in that case I would recommend trying `asyncio` instead. The biggest problem with `threading` is that if one of the threads blocks, the main process blocks.

The `multiprocessing` library offers an API that is very similar to the `threading` library but utilizes multiple processes instead of multiple threads. The advantages are that the GIL is no longer an issue and that multiple processor cores and even multiple machines can be used for processing.

To illustrate the performance difference, let's repeat the test while using the `multiprocessing` module instead of `threading`:

```
import datetime
import multiprocessing


def busy_wait(n):
    while n > 0:
        n -= 1


if __name__ == '__main__':
    n = 10000000
    start = datetime.datetime.now()

    processes = []
    for _ in range(4):
        process = multiprocessing.Process(
            target=busy_wait, args=(n,))
        process.start()
```

```
            processes.append(process)

        for process in processes:
            process.join()

        end = datetime.datetime.now()
        print('The multiprocessed loops took: %s' % (end - start))
```

When running it, we see a huge improvement:

```
# python3 test_multiprocessing.py
The multiprocessed loops took: 0:00:00.671249
```

Note that this was run on a quad core processor, which is why I chose four processes. The `multiprocessing` library defaults to `multiprocessing.cpu_count()` which counts the available CPU cores, but that method fails to take CPU hyper-threading into account. Which means it would return 8 in my case and that is why I hardcoded it to 4 instead.

It's important to note that because the `multiprocessing` library uses multiple processes, the code needs to be imported from the sub processes. The result is that the `multiprocessing` library does not work within the Python or IPython shells. As we will see later in this chapter, IPython has its own provisions for multiprocessing.

Hyper-threading versus physical CPU cores

In most cases, hyper-threading is very useful and improves performance, but when you truly maximize CPU usage it is generally better to only use the physical processor count. To demonstrate how this affects the performance, we will run the tests from the previous section again. This time with 1, 2, 4, 8, and 16 processes to demonstrate how it affects the performance. Luckily, the `multiprocessing` library has a nice `Pool` class to manage the processes for us:

```
import sys
import datetime
import multiprocessing


def busy_wait(n):
    while n > 0:
```

```
            n -= 1


    if __name__ == '__main__':
        n = 10000000
        start = datetime.datetime.now()
        if sys.argv[-1].isdigit():
            processes = int(sys.argv[-1])
        else:
            print('Please specify the number of processes')
            print('Example: %s 4' % ' '.join(sys.argv))
            sys.exit(1)

        with multiprocessing.Pool(processes=processes) as pool:
            # Execute the busy_wait function 8 times with parameter n
            pool.map(busy_wait, [n for _ in range(8)])

        end = datetime.datetime.now()
        print('The multithreaded loops took: %s' % (end - start))
```

The pool code makes starting a pool of workers and processing a queue a bit simpler as well. In this case we used `map` but there are several other options such as `imap`, `map_async`, `imap_unordered`, `apply`, `apply_async`, `starmap`, and `starmap_async`. Since these are very similar to how the similarly named `itertools` methods work, there won't be specific examples for all of them.

But now, the tests with varying amounts of processes:

```
# python3 test_multiprocessing.py 1
The multithreaded loops took: 0:00:05.297707
# python3 test_multiprocessing.py 2
The multithreaded loops took: 0:00:02.701344
# python3 test_multiprocessing.py 4
The multithreaded loops took: 0:00:01.477845
# python3 test_multiprocessing.py 8
The multithreaded loops took: 0:00:01.579218
# python3 test_multiprocessing.py 16
The multithreaded loops took: 0:00:01.595239
```

You probably weren't expecting these results, but this is exactly the problem with hyper-threading. As soon as the single processes actually use 100 percent of a CPU core, the task switching between the processes actually reduces performance. Since there are only `4` physical cores, the other `4` have to fight to get something done on the processor cores. This fight takes time which is why the `4` process version is slightly faster than the `8` process version. Additionally, the scheduling effect can be seen in the runs using `1` and `2` cores as well. If we look at the single core version, we see that it took `5.3` seconds, which means that `4` cores should do it in `5.3 / 4 = 1.325` seconds instead of the `1.48` seconds it actually took. The `2` core version has a similar effect, `2.7 / 2 = 1.35` seconds which is still faster than `4` core version.

If you are truly pressed for performance with a CPU-bound problem then matching the physical CPU cores is the best solution. If you do not expect to maximize all cores all the time, then I recommend leaving it to the default as hyper-threading definitely has some performance benefits in other scenarios.

It all depends on your use-case however and the only way to know for certain is to test for your specific scenario:

- Disk I/O bound? A single process is most likely your best bet.
- CPU bound? The amount of physical CPU cores is your best bet.
- Network I/O bound? Start with the defaults and tune if needed.
- No obvious bound but many parallel processes are needed? Perhaps you should try `asyncio` instead of `multiprocessing`.

Note that the creation of multiple processes is not free in terms of memory and open files, whereas you could have a nearly unlimited amount of coroutines this is not the case for processes. Depending on your operating system configuration, it could max out long before you even reach a hundred, and even if you reach those numbers, CPU scheduling will be your bottleneck instead.

Creating a pool of workers

Creating a processing pool of worker processes is generally a difficult task. You need to take care of scheduling jobs, processing the queue, handling the processes, and the most difficult part, handling synchronization between the processes without too much overhead.

With `multiprocessing` however, these problems have been solved already. You can simply create a process pool with a given number of processes and just add tasks to it whenever you need to. The following is an example of a multiprocessing version of the map operator and demonstrates that processing will not stall the application:

```
import time
import multiprocessing


def busy_wait(n):
    while n > 0:
        n -= 1


if __name__ == '__main__':
    n = 10000000
    items = [n for _ in range(8)]
    with multiprocessing.Pool() as pool:
        results = []
        start = time.time()
        print('Start processing...')
        for _ in range(5):
            results.append(pool.map_async(busy_wait, items))
        print('Still processing %.3f' % (time.time() - start))
        for result in results:
            result.wait()
            print('Result done %.3f' % (time.time() - start))
        print('Done processing: %.3f' % (time.time() - start))
```

The processing itself is pretty straightforward. The point is that the pool stays available and you are not required to wait for it. Just add jobs whenever you need to and use the asynchronous results as soon as they are available:

```
# python3 test_pool.py
Start processing...
Still processing 0.000
Result done 1.513
Result done 2.984
Result done 4.463
Result done 5.978
Result done 7.388
Done processing: 7.388
```

Sharing data between processes

This is really the most difficult part about multiprocessing, multithreading, and distributed programming - which data to pass along and which data to skip. The theory is really simple, however: whenever possible don't transfer any data, don't share anything, and keep everything local. Essentially the functional programming paradigm, which is why functional programming mixes really well with multiprocessing. In practice, regrettably, this is simply not always possible. The `multiprocessing` library has several options to share data: `Pipe`, `Namespace`, `Queue`, and a few others. All these options might tempt you to share your data between the processes all the time. This is indeed possible, but the performance impact is, in many cases, more than what the distributed calculation will offer as extra power. All data sharing options come at the price of synchronization between all processing kernels, which takes a lot of time. Especially with distributed options, these synchronizations can take several milliseconds or, if executed globally, cause hundreds of milliseconds of latency.

The multiprocessing namespace behaves just as a regular object would work, with one small difference that all the actions are safe for multiprocessing. With all this power, namespaces are still very easy to use:

```
import multiprocessing
manager = multiprocessing.Manager()
namespace = manager.Namespace()
namespace.spam = 123
namespace.eggs = 456
```

A pipe is not that much more interesting either. It's just a bidirectional communication endpoint which allows both reading and writing. In this regard, it simply offers you a reader and a writer, and because of that, you can combine multiple processes/endpoints. The only thing you must always keep in mind when synchronizing data is that locking takes time. For a proper lock to be set, all the parties need to agree that the data is locked, which is a process that takes time. And that simple fact slows down execution much more than most people would expect.

On a regular hard disk setup, the database servers aren't able to handle more than about 10 transactions per second on the same row due to locking and disk latency. Using lazy file syncing, SSDs, and battery backed RAID cache, that performance can be increased to handle, perhaps, a 100 transactions per second on the same row. Those are simple hardware limitations, because you have multiple processes trying to write to a single target you need to synchronize the actions between the processes and that takes a lot of time.

The "database servers" statistic is a common statistic for all database servers that offer safe and consistent data storage.

Even with the fastest hardware available, synchronization can lock all the processes and produce enormous slowdowns, so if at all possible, try to avoid sharing data between multiple processes. Put simply, if all the processes are reading and writing from/to the same object, it is generally faster to use a single process instead.

Remote processes

So far, we have only executed our scripts on multiple local processors, but we can actually expand this further. Using the `multiprocessing` library, it's actually very easy to execute jobs on remote servers, but the documentation is currently still a bit cryptic. There are actually a few ways of executing processes in a distributed way, but the most obvious one isn't the easiest one. The `multiprocessing.connection` module has both the `Client` and `Listener` classes, which facilitate secure communication between the clients and servers in a simple way. Communication is not the same as process management and queue management however, those features requires some extra effort. The multiprocessing library is still a bit bare in this regard, but it's most certainly possible given a few different processes.

Distributed processing using multiprocessing

First of all, we will start with a module with containing a few constants which should be shared between all clients and the server, so the secret password and the hostname of the server are available to all. In addition to that, we will add our prime calculation functions, which we will be using later. The imports in the following modules will expect this file to be stored as `constants.py`, but feel free to call it anything you like as long as you modify the imports and references:

```
host = 'localhost'
port = 12345
password = b'some secret password'


def primes(n):
    for i, prime in enumerate(prime_generator()):
        if i == n:
            return prime


def prime_generator():
```

```
    n = 2
    primes = set()
    while True:
        for p in primes:
            if n % p == 0:
                break
        else:
            primes.add(n)
            yield n
        n += 1
```

Now it's time to create the actual server which links the functions and the job queue:

```
import constants
import multiprocessing
from multiprocessing import managers


queue = multiprocessing.Queue()
manager = managers.BaseManager(address=('', constants.port),
                               authkey=constants.password)

manager.register('queue', callable=lambda: queue)
manager.register('primes', callable=constants.primes)

server = manager.get_server()
server.serve_forever()
```

After creating the server, we need to have a script that sends the jobs, which will actually be a regular client. It's simple enough really and a regular client can also function as a processor, but to keep things sensible we will use them as separate scripts. The following script will add 0 to 999 to the queue for processing:

```
from multiprocessing import managers
import functions


manager = managers.BaseManager(
    address=(functions.host, functions.port),
    authkey=functions.password)
manager.register('queue')
manager.connect()

queue = manager.queue()
for i in range(1000):
    queue.put(i)
```

Lastly, we need to create a client to actually process the queue:

```
from multiprocessing import managers
import functions


manager = managers.BaseManager(
    address=(functions.host, functions.port),
    authkey=functions.password)
manager.register('queue')
manager.register('primes')
manager.connect()

queue = manager.queue()
while not queue.empty():
    print(manager.primes(queue.get()))
```

From the preceding code you can see how we pass along functions; the manager allows registering of functions and classes which can be called from the clients as well. With that we pass along a queue from the multiprocessing class which is safe for both multithreading and multiprocessing. Now we need to start the processes themselves. First the server which keeps on running:

```
# python3 multiprocessing_server.py
```

After that, run the producer to generate the prime generation requests:

```
# python3 multiprocessing_producer.py
```

And now we can run multiple clients on multiple machines to get the first 1000 primes. Since these clients now print the first 1000 primes, the output is a bit too lengthy to show here, but you can simply run this in parallel on multiple machines to generate your output:

```
# python3 multiprocessing_client.py
```

Instead of printing, you can obviously use queues or pipes to send the output to a different process if you'd like. As you can see verify though, it's still a bit of work to process things in parallel and it requires some code synchronization to work. There are a few alternatives available, such as **ØMQ**, **Celery**, and **IPyparallel**. Which of these is the best and most suitable depends on your use case. If you are simply looking for processing tasks on multiple CPUs, then multiprocessing and IPyparallel are probably your best choices. If you are looking for background processing and/or easy offloading to multiple machines, then ØMQ and Celery are better choices.

Distributed processing using IPyparallel

The IPyparallel module (previously, IPython Parallel) is a module that makes it really easy to process code on multiple computers at the same time. The library supports more features than you are likely to need, but the basic usage is important to know just in case you need to do heavy calculations which can benefit from multiple computers. First let's start with installing the latest IPyparallel package and all the IPython components:

```
pip install -U ipython[all] ipyparallel
```

Especially on Windows, it might be easier to install IPython using Anaconda instead, as it includes binaries for many science, math, engineering, and data analysis packages. To get a consistent installation, the Anaconda installer is also available for OS X and Linux systems.

Secondly, we need a cluster configuration. Technically this is optional, but since we are going to create a distributed IPython cluster, it is much more convenient to configure everything using a specific profile:

```
# ipython profile create --parallel --profile=mastering_python
[ProfileCreate] Generating default config file: '~/.ipython/profile_
mastering_python/ipython_config.py'
[ProfileCreate] Generating default config file: '~/.ipython/profile_
mastering_python/ipython_kernel_config.py'
[ProfileCreate] Generating default config file: '~/.ipython/profile_
mastering_python/ipcontroller_config.py'
[ProfileCreate] Generating default config file: '~/.ipython/profile_
mastering_python/ipengine_config.py'
[ProfileCreate] Generating default config file: '~/.ipython/profile_
mastering_python/ipcluster_config.py'
```

These configuration files contain a huge amount of options so I recommend searching for a specific section instead of walking through them. A quick listing gave me about 2500 lines of configuration in total for these five files. The filenames already provide hint about the purpose of the configuration files, but we'll explain them in a little more detail since they are still a tad confusing.

ipython_config.py

This is the generic IPython configuration file; you can customize pretty much everything about your IPython shell here. It defines how your shell should look, which modules should be loaded by default, whether or not to load a GUI, and quite a bit more. For the purpose of this chapter not all that important but it's definitely worth a look if you're going to use IPython more often. One of the things you can configure here is the automatic loading of extensions, such as `line_profiler` and `memory_profiler` discussed in the previous chapter. For example:

```
c.InteractiveShellApp.extensions = [
    'line_profiler',
    'memory_profiler',
]
```

ipython_kernel_config.py

This file configures your IPython kernel and allows you to overwrite/extend `ipython_config.py`. To understand its purpose, it's important to know what an IPython kernel is. The kernel, in this context, is the program that runs and introspects the code. By default this is `IPyKernel`, which is a regular Python interpreter, but there are also other options such as `IRuby` or `IJavascript` to run Ruby or JavaScript respectively.

One of the more useful options is the possibility to configure the listening port(s) and IP addresses for the kernel. By default the ports are all set to use a random number, but it is important to note that if someone else has access to the same machine while you are running your kernel, they will be able to connect to your IPython kernel which can be dangerous on shared machines.

ipcontroller_config.py

`ipcontroller` is the master process of your IPython cluster. It controls the engines and the distribution of tasks, and takes care of tasks such as logging.

The most important parameter in terms of performance is the `TaskScheduler` setting. By default, the `c.TaskScheduler.scheme_name` setting is set to use the Python LRU scheduler, but depending on your workload, others such as `leastload` and `weighted` might be better. And if you have to process so many tasks on such a large cluster that the scheduler becomes the bottleneck, there is also the `plainrandom` scheduler that works surprisingly well if all your machines have similar specs and the tasks have similar durations.

For the purpose of our test we will set the IP of the controller to *, which means that **all** IP addresses will be accepted and that every network connection will be accepted. If you are in an unsafe environment/network and/or don't have any firewalls which allow you to selectively enable certain IP addresses, then this method is **not** recommended! In such cases, I recommend launching through more secure options, such as `SSHEngineSetLauncher` or `WindowsHPCEngineSetLauncher` instead.

But, assuming your network is indeed safe, set the factory IP to all the local addresses:

```
c.HubFactory.client_ip = '*'
c.RegistrationFactory.ip = '*'
```

Now start the controller:

```
# ipcontroller --profile=mastering_python
[IPControllerApp] Hub listening on tcp://*:58412 for registration.
[IPControllerApp] Hub listening on tcp://127.0.0.1:58412 for
registration.
[IPControllerApp] Hub using DB backend: 'NoDB'
[IPControllerApp] hub::created hub
[IPControllerApp] writing connection info to ~/.ipython/profile_
mastering_python/security/ipcontroller-client.json
[IPControllerApp] writing connection info to ~/.ipython/profile_
mastering_python/security/ipcontroller-engine.json
[IPControllerApp] task::using Python leastload Task scheduler
[IPControllerApp] Heartmonitor started
[IPControllerApp] Creating pid file: .ipython/profile_mastering_python/
pid/ipcontroller.pid
[scheduler] Scheduler started [leastload]
[IPControllerApp] client::client b'\x00\x80\x00A\xa7' requested
'connection_request'
[IPControllerApp] client::client [b'\x00\x80\x00A\xa7'] connected
```

Pay attention to the files that were written to the security directory of the profile directory. They have the authentication information which is used by `ipengine` to find `ipcontroller`. It contains the ports, encryption keys, and IP address.

ipengine_config.py

`ipengine` is the actual worker process. These processes run the actual calculations, so to speed up the processing you will need these on as many machines as you have available. You probably won't need to change this file, but it can be useful if you want to configure centralized logging or need to change the working directory. Generally, you don't want to start the `ipengine` process manually since you will most likely want to launch multiple processes per computer. That's where our next command comes in, the `ipcluster` command.

ipcluster_config.py

The `ipcluster` command is actually just an easy shorthand to start a combination of `ipcontroller` and `ipengine` at the same time. For a simple local processing cluster, I recommend using this, but when starting a distributed cluster, it can be useful to have the control that the separate use of `ipcontroller` and `ipengine` offers. In most cases the command offers enough options, so you might have no need for the separate commands.

The most important configuration option is `c.IPClusterEngines.engine_launcher_class`, as this controls the communication method between the engines and the controller. Along with that, it is also the most important component for secure communication between the processes. By default it's set to `ipyparallel.apps.launcher.LocalControllerLauncher` which is designed for local processes but `ipyparallel.apps.launcher.SSHEngineSetLauncher` is also an option if you want to use SSH to communicate with the clients. Or `ipyparallel.apps.launcher.WindowsHPCEngineSetLauncher` for Windows HPC.

Before we can create the cluster on all machines, we need to transfer the configuration files. Your options are to transfer all the files or to simply transfer the files in your IPython profile's `security` directory.

Now it's time to start the cluster, since we already started the `ipcontroller` separately, we only need to start the engines. On the local machine we simply need to start it, but the other machines don't have the configuration yet. One option is copying the entire IPython profile directory, but the only file that really needs copying is `security/ipcontroller-engine.json`. After creating the profile using the profile creation command that is. So unless you are going to copy the entire IPython profile directory, you need to execute the profile creation command again:

```
# ipython profile create --parallel --profile=mastering_python
```

After that, simply copy the `ipcontroller-engine.json` file and you're done. Now we can start the actual engines:

```
# ipcluster engines --profile=mastering_python -n 4
[IPClusterEngines] IPython cluster: started
[IPClusterEngines] Starting engines with [daemon=False]
[IPClusterEngines] Starting 4 Engines with LocalEngineSetLauncher
```

Note that the `4` here was chosen for a quad-core processor, but any number would do. The default will use the amount of logical processor cores, but depending on the workload it might be better to match the amount of physical processor cores instead.

Now we can run some parallel code from our IPython shell. To demonstrate the performance difference, we will use a simple sum of all the numbers from 0 to 10,000,000. Not an extremely heavy task, but when performed 10 times in succession, a regular Python interpreter takes a while:

```
In [1]: %timeit for _ in range(10): sum(range(10000000))
1 loops, best of 3: 2.27 s per loop
```

This time however, to illustrate the difference, we will run it a 100 times to demonstrate how fast a distributed cluster is. Note that this is with only three machines cluster, but it's still quite a bit faster:

```
In [1]: import ipyparallel

In [2]: client = ipyparallel.Client(profile='mastering_python')

In [3]: view = client.load_balanced_view()

In [4]: %timeit view.map(lambda _: sum(range(10000000)), range(100)).
wait()
1 loop, best of 3: 909 ms per loop
```

More fun however is the definition of parallel functions in IPyParallel. With just a simple decorator, a function is marked as parallel:

```
In [1]: import ipyparallel

In [2]: client = ipyparallel.Client(profile='mastering_python')

In [3]: view = client.load_balanced_view()

In [4]: @view.parallel()
   ...: def loop():
```

```
...:     return sum(range(10000000))
...:

In [5]: loop.map(range(10))
Out[5]: <AsyncMapResult: loop>
```

The IPyParallel library offers many more useful features, but that is outside the scope of this book. Even though IPyParallel is a separate entity from the rest of Jupyter/IPython, it does integrate well, which makes combining them easy enough.

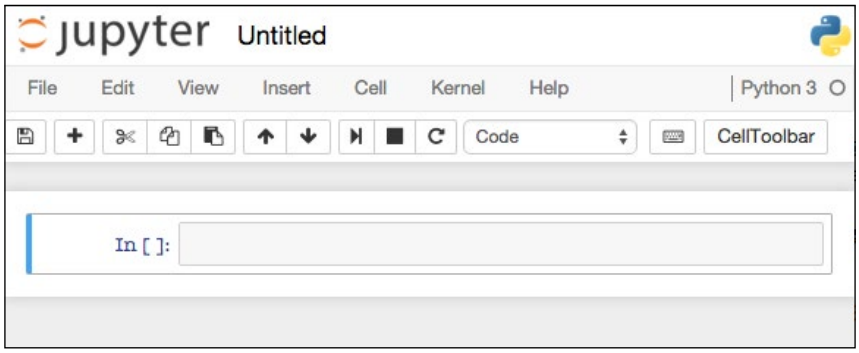
One of the most convenient ways of using IPyParallel is through the Jupyter/IPython Notebooks. To demonstrate, we first have to make sure to enable the parallel processing in the Jupyter Notebook since IPython notebooks execute single threaded by default:

```
ipcluster nbextension enable
```

After that we can start the `notebook` and see what it's all about:

```
# jupyter notebook
Unrecognized JSON config file version, assuming version 1
Loading IPython parallel extension
Serving notebooks from local directory: ./
0 active kernels
The Jupyter Notebook is running at: http://localhost:8888/
Use Control-C to stop this server and shut down all kernels (twice to
skip confirmation).
```

With the Jupyter Notebook you can create scripts in your web browser which can easily be shared with others later. It is really very useful for sharing scripts and debugging your code, especially since web pages (as opposed to command line environments) can display images easily. This helps a lot with graphing data. Here's a screenshot of our Notebook:



Summary

This chapter has shown us how multiprocessing works, how we can pool a lot of jobs, and how we should share data between multiple processes. But more interestingly, it has also shown how we can distribute processing across multiple machines which helps a lot in speeding up heavy calculations.

The most important lesson you can learn from this chapter is that you should always try to avoid data sharing and synchronisation between multiple processes or servers, as it is slow and will thus slow down your applications a lot. Whenever possible, keep your calculations and data local.

In the next chapter we will learn about creating extensions in C/C++ to increase performance and allow low-level access to memory and other hardware resources. While Python will generally protect you from silly mistakes, C and C++ most certainly won't.

> *"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows away your whole leg."*
>
> *- Bjarne Stroustrup (the creator of C++)*

14
Extensions in C/C++, System Calls, and C/C++ Libraries

Now that we know a bit more about performance and multiprocessing, we will explain another subject that is at least somewhat performance-related—the usage of C and/or C++ extensions.

There are multiple reasons to consider C/C++ extensions. Having existing libraries available is an important one, but truthfully, the most important reason is performance. In *Chapter 12, Performance – Tracking and Reducing Your Memory and CPU Usage*, we saw that the `cProfile` module is about 10 times faster than the `profile` module, which indicates that at least some C extensions are faster than their pure Python equivalents. This chapter will not focus on performance that much, however. The goal here is interaction with non-Python libraries. Any performance improvement will just be a completely unintentional side effect.

We will discuss the following options in this chapter:

- Ctypes for handling foreign (C/C++) functions and data from Python
- **CFFI** (short for **C Foreign Function Interface**), similar to `ctypes` but with a slightly different approach
- Writing native C/C++ to extend Python

Introduction

Before you start with this chapter, it is important to note that this chapter will require a working compiler that plays nicely with your Python interpreter. Unfortunately, these vary from platform to platform. While generally easy enough for most Linux distributions, this can be a big challenge on Windows. With OS X, it's generally easy enough provided you install the correct tools.

The generic building instructions are always available in the Python manual:

```
https://docs.python.org/3.5/extending/building.html
```

Do you need C/C++ modules?

In almost all cases, I'm inclined to say that you don't need C/C++ modules. If you are really strapped for best performance, then there are almost always highly optimized libraries available that fit your purpose. There are some cases where native C/C++ (or just "not Python") is a requirement. If you need to communicate directly with hardware that has specific timings, then Python might just not do the trick for you. Generally, however, that kind of communication should be left to a driver that takes care of the specific timings. Regardless, even if you will never write one of these modules yourself, you might still need to know how they work when you are debugging a project.

Windows

For Windows, the general recommendation is Visual Studio. The specific version depends on your Python version:

- Python 3.2 and lower: Microsoft Visual Studio 2008
- Python 3.3 and 3.4: Microsoft Visual Studio 2010
- Python 3.5 and 3.6: Microsoft Visual Studio 2015

The specifics of installing Visual Studio and compiling Python modules fall a bit outside of the scope of this book. Luckily, the Python documentation has some documentation available to get you started:

```
https://docs.python.org/3.5/extending/windows.html
```

OS X

For a Mac, the process is mostly straightforward, but there are a few tips specific to OS X.

First, install Xcode through the Mac App Store. Once you have done that, you should be able to run the following command:

```
xcode-select --install
```

Next up is the fun part. Because OS X comes with a bundled Python version (which is generally out of date), I would recommend installing a new Python version through Homebrew instead. The most up-to-date instructions for installing Homebrew can be found on the Homebrew homepage (`http://brew.sh/`), but the gist of installing Homebrew is this command:

```
# /usr/bin/ruby -e "$(curl -fsSL \

https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

After that, make sure you check whether everything is set up correctly using the `doctor` command:

```
# brew doctor
```

When all of this is done, simply install Python through Homebrew and make sure you use that Python release when executing your scripts:

```
# brew install python3
# python3 –version
Python 3.5.1
which python3
/usr/local/bin/python3
```

Also ensure that the Python process is in `/usr/local/bin`, that is, the homebrewed version. The regular OS X version would be in `/usr/bin/` instead.

Linux/Unix

The installation for Linux/Unix systems greatly depends on the distribution, but it is generally simple to do.

For Fedora, Red Hat, Centos, and other systems that use `yum` as the package manager, use these lines:

```
# sudo yum install yum-utils
# sudo yum-builddep python3
```

For Debian, Ubuntu, and other systems that use `apt` as the package manager, use the following line:

```
# sudo apt-get build-dep python3.5
```

Note that Python 3.5 is not available everywhere yet, so you might need Python 3.4 instead.

For most systems, to get help with the installation, a web search along the lines of `<operating system> python.h` should do the trick.

Calling C/C++ with ctypes

The `ctypes` library makes it easily possible to call functions from C libraries, but you do need to be careful with memory access and data types. Python is generally very lenient in memory allocation and type casting; C is, most definitely, not that forgiving.

Platform-specific libraries

Even though all platforms will have a standard C library available somewhere, the location and the method of calling it differs per platform. For the purpose of having a simple environment that is easily accessible to most people, I will assume the use of an Ubuntu (virtual) machine. If you don't have a native Ubuntu available, you can easily run it through VirtualBox on Windows, Linux, and OS X.

Since you will often want to run examples on your native system instead, we will first show the basics of loading `printf` from the standard C library.

Windows

One problem of calling C functions from Python is that the default libraries are platform-specific. While the following example will work just fine on Windows systems, it won't run on other platforms:

```
>>> import ctypes
>>> ctypes.cdll
<ctypes.LibraryLoader object at 0x...>
>>> libc = ctypes.cdll.msvcrt
>>> libc
<CDLL 'msvcrt', handle ... at ...>
>>> libc.printf
<_FuncPtr object at 0x...>
```

Because of these limitations, not all examples can work for every Python version and distribution without requiring manual compilation. The basic premise of calling functions from external libraries functions is to simply access their names as properties of the `ctypes` import. There is a difference, however; on Windows, the modules will generally be auto-loaded, while on Linux/Unix systems, you will need to load them manually.

Linux/Unix

Calling standard system libraries from Linux/Unix does require manual loading, but it's nothing too involved luckily. Fetching the `printf` function from the standard C library is quite simple:

```
>>> import ctypes
>>> ctypes.cdll
<ctypes.LibraryLoader object at 0x...>
>>> libc = ctypes.cdll.LoadLibrary('libc.so.6')
>>> libc
<CDLL 'libc.so.6', handle ... at ...>
>>> libc.printf
<_FuncPtr object at 0x...>
```

OS X

For OS X, explicit loading is also required, but beyond that, it is quite similar to how everything works on regular Linux/Unix systems:

```
>>> import ctypes
>>> libc = ctypes.cdll.LoadLibrary('libc.dylib')
>>> libc
<CDLL 'libc.dylib', handle ... at 0x...>
>>> libc.printf
<_FuncPtr object at 0x...>
```

Making it easy

Besides the way libraries are loaded, there are more differences—unfortunately—but these examples at least give you the standard C library. It allows you to call functions such as `printf` straight from your C implementation. If, for some reason, you have trouble loading the right library, there is always the `ctypes.util.find_library` function. As always, I recommend explicit over implicit declarations, but things can be made easier using this function. Let's illustrate a run on an OS X system:

```
>>> from ctypes import util
>>> from ctypes import cdll
>>> libc = cdll.LoadLibrary(util.find_library('libc'))
>>> libc
<CDLL '/usr/lib/libc.dylib', handle ... at 0x...>
```

Calling functions and native types

Calling a function through `ctypes` is nearly as simple as calling native Python functions. The notable difference is the arguments and return statements. These should be converted to native C variables:

These examples will assume that you have `libc` in your scope from one of the examples in the previous paragraphs.

```
>>> spam = ctypes.create_string_buffer(b'spam')
>>> ctypes.sizeof(spam)
5
>>> spam.raw
b'spam\x00'
>>> spam.value
b'spam'
>>> libc.printf(spam)
4
spam>>>
```

As you can see, to call the `printf` function you *must*—and I cannot stress this enough—convert your values from Python to C explicitly. While it might appear to work without this initially, it really doesn't:

```
>>> libc.printf(123)
segmentation fault (core dumped)  python3
```

Remember to use the `faulthandler` module from *Chapter 11, Debugging – Solving the Bugs* to debug segfaults.

Another thing to note from the example is that `ctypes.sizeof(spam)` returns `5` instead of `4`. This is caused by the trailing null character, which C strings require. This is visible in the raw property of the C string. Without it, the `printf` function won't know where the string will end.

To pass along other types (such as integers) towards `libc` functions, we have to use some conversion as well. In some cases, it is optional:

```
>>> format_string = ctypes.create_string_buffer(b'Number: %d\n')
>>> libc.printf(format_string, 123)
Number: 123
12
>>> x = ctypes.c_int(123)
>>> libc.printf(format_string, x)
Number: 123
12
```

But not in all cases, so it's definitely recommended that you convert your values explicitly in all cases:

```
>>> format_string = ctypes.create_string_buffer(b'Number: %.3f\n')
>>> libc.printf(format_string, 123.45)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 2: <class 'TypeError'>: Don't know how to
convert parameter 2
>>> x = ctypes.c_double(123.45)
>>> libc.printf(format_string, x)
Number: 123.450
16
```

It's important to note that even though these values are usable as native C types, they are still mutable through the `value` attribute:

```
>>> x = ctypes.c_double(123.45)
>>> x.value
123.45
>>> x.value = 456
>>> x
c_double(456.0)
```

However, this is not the case if the original object was immutable, which is a very important distinction to make. The `create_string_buffer` object creates a mutable string object, whereas `c_wchar_p`, `c_char_p`, and `c_void_p` create references to the actual Python string. Since strings are immutable in Python, these values are also immutable. You can still change the `value` property, but it will only assign a new string. Actually, passing one of these to a C function that mutates the internal value will cause problems.

The only values that should convert to C without any issues are integers, strings, and bytes, but I personally recommend that you always convert all of your values so that you are certain of which type you will get and how to treat it.

Complex data structures

We have already seen that we can't just pass along Python values to C, but what if we need more complex objects? That is, not just bare values that are directly translatable to C but complex objects containing multiple values. Luckily, we can easily create (and access) C structures using `ctypes`:

```
>>> class Spam(ctypes.Structure):
...     _fields_ = [
...         ('spam', ctypes.c_int),
...         ('eggs', ctypes.c_double),
...     ]
...>>> spam = Spam(123, 456.789)
>>> spam.spam
123
>>> spam.eggs
456.789
```

Arrays

Within Python, we generally use a list to represent a collection of objects. These are very convenient in that you can easily add and remove values. Within C, the default collection object is the array, which is just a block of memory with a fixed size.

The size of the block in bytes is decided by multiplying the number of items with the size of the type. In the case of a `char`, this is `8` bits, so if you wish to store `100` chars, you would have `100 * 8 bits = 800 bits = 100 bytes`.

This is literally all it is—a block of memory—and the only reference you receive from C is a pointer to the memory address where the block of memory begins. Since the pointer does have a type, `char*` in this case, C will know how many bytes to jump ahead when trying to access a different item. Effectively, when trying to access item 25 in a `char` array, you simply need to do `array_pointer + 25 * sizeof(char)`. This has a convenient shortcut: `array_pointer[25]`.

Note that C does not store the number of items in the array, so even though our array has only 100 items, it won't block us from doing `array_pointer[1000]` and reading other (random) memory.

If you take all of that into account, it is definitely usable, but mistakes are quickly made and C is unforgiving. No warnings, just crashes and strangely behaving code. Beyond that, let's see how easily we can declare an array with `ctypes`:

```
>>> TenNumbers = 10 * ctypes.c_double
>>> numbers = TenNumbers()
>>> numbers[0]
0.0
```

As you can see, because of the fixed sizes and the requirement of declaring the type before using it, its usage is slightly awkward. However, it does function as you would expect, and the values are initialized to zero by default. Obviously, this can be combined with the previously discussed structures as well:

```
>>> Spams = 5 * Spam
>>> spams = Spams()
>>> spams[0].eggs = 123.456
>>> spams
<__main__.Spam_Array_5 object at 0x...>
>>> spams[0]
<__main__.Spam object at 0x...>
>>> spams[0].eggs
123.456
>>> spams[0].spam
0
```

Even though you cannot simply append to these arrays to resize them, they are actually resizable with a few constraints. Firstly, the new array needs to be larger than the original array. Secondly, the size needs to be specified in bytes, not items. To illustrate, we have this example:

```
>>> TenNumbers = 10 * ctypes.c_double
>>> numbers = TenNumbers()
>>> ctypes.resize(numbers, 11 * ctypes.sizeof(ctypes.c_double))
>>> ctypes.resize(numbers, 10 * ctypes.sizeof(ctypes.c_double))
>>> ctypes.resize(numbers, 9 * ctypes.sizeof(ctypes.c_double))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: minimum size is 80
>>> numbers[:5] = range(5)
>>> numbers[:]
[0.0, 1.0, 2.0, 3.0, 4.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

Gotchas with memory management

Besides the obvious memory allocation issues and mixing mutable and immutable objects, there is one more strange memory mutability issue:

```
>>> class Point(ctypes.Structure):
...     _fields_ = ('x', ctypes.c_int), ('y', ctypes.c_int)
...
>>> class Vertex(ctypes.Structure):
...     _fields_ = ('a', Point), ('b', Point), ('c', Point)
...
>>> v = Vertex()
>>> v.a = Point(0, 1)
>>> v.b = Point(2, 3)
>>> v.c = Point(4, 5)
>>> v.a.x, v.a.y, v.b.x, v.b.y, v.c.x, v.c.y
(0, 1, 2, 3, 4, 5)
>>> v.a, v.b, v.c = v.b, v.c, v.a
>>> v.a.x, v.a.y, v.b.x, v.b.y, v.c.x, v.c.y
(2, 3, 4, 5, 2, 3)
>>> v.a.x = 123
>>> v.a.x, v.a.y, v.b.x, v.b.y, v.c.x, v.c.y
(123, 3, 4, 5, 2, 3)
```

Why didn't we get `2, 3, 4, 5, 0, 1`? The problem is that these objects are copied to a temporary buffer variable. In the meantime, the values of that object are being changed because it contains separate objects internally. After that, the object is transferred back, but the values have already changed, giving the incorrect results.

CFFI

The `CFFI` library offers options very similar to `ctypes`, but it's a bit more direct. Unlike the `ctypes` library, a C compiler is really a necessity for `CFFI`. With it comes the opportunity to directly call your C compiler in a very easy way:

```
>>> import cffi
>>> ffi = cffi.FFI()
>>> ffi.cdef('int printf(const char* format, ...);')
>>> libc = ffi.dlopen(None)
>>> arg = ffi.new('char[]', b'spam')
>>> libc.printf(arg)
4
spam>>>
```

Okay… so that looks a bit weird right? We had to define how the `printf` function looks and specify the arguments to `printf` with a valid C type declaration. Getting back to the declarations, however, instead of `None` to `ffi.dlopen`, you can also specify the library you wish to load. If you remember the `ctypes.util.find_library` function, you can use that again in this case:

```
>>> from ctypes import util
>>> import cffi
>>> libc = ffi.dlopen(util.find_library('libc'))
>>> ffi.printf
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'FFI' object has no attribute 'printf'
```

But it still won't make its definitions readily available for you. The function definitions are still required to make sure that everything works as you would like it to.

Complex data structures

The `CFFI` definitions are somewhat similar to the `ctypes` definitions, but instead of having Python emulating C, it's just plain C that is accessible from Python. In reality, it's just a small syntactical difference. Whereas `ctypes` is a library for accessing C from Python while remaining as close to the Python syntax as possible, `CFFI` uses plain C syntax to access C systems, which actually removes some confusion for people experienced with C. I personally find `CFFI` easier to use because I know what is actually happening, whereas I am not always a 100% certain with `ctypes`. Let's repeat the `Vertex` and `Point` example with CFFI:

```
>>> import cffi
>>> ffi = cffi.FFI()
>>> ffi.cdef('''
... typedef struct {
...     int x;
...     int y;
... } point;
...
... typedef struct {
...     point a;
...     point b;
...     point c;
... } vertex;
... ''')
>>> vertices = ffi.new('vertex[]', 5)
>>> v = vertices[0]
>>> v.a.x = 1
>>> v.a.y = 2
>>> v.b.x = 3
>>> v.b.y = 4
>>> v.c.x = 5
>>> v.c.y = 6
>>> v.a.x, v.a.y, v.b.x, v.b.y, v.c.x, v.c.y
(1, 2, 3, 4, 5, 6)
v.a, v.b, v.c = v.b, v.c, v.a
v.a.x, v.a.y, v.b.x, v.b.y, v.c.x, v.c.y
>>> v.a, v.b, v.c = v.b, v.c, v.a
>>> v.a.x, v.a.y, v.b.x, v.b.y, v.c.x, v.c.y
(3, 4, 5, 6, 3, 4)
```

As you can see, the mutable variable issues remain but the code is just as usable.

Arrays

Allocation memory for new variables is almost trivial with `CFFI`. The previous paragraph showed you an example of array allocation; let's see the possibilities of array definitions now:

```
>>> import cffi
>>> ffi = cffi.FFI()
>>> x = ffi.new('int[10]')
>>> y = ffi.new('int[]', 10)
>>> x[0:10] = range(10)
>>> y[0:10] = range(10, 0, -1)
>>> list(x)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(y)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

In this case, you might wonder why the slice includes both the start and the stop. This is actually a requirement for `CFFI`. Not always problematic but a tad annoying nonetheless. Currently, however, it's unavoidable.

ABI or API?

As always, there are some caveats—unfortunately. The examples so far have partially used the ABI, which loads the binary structures from the libraries. With the standard C library, this is generally safe; with other libraries, it generally isn't. The difference between the API and the ABI is that the latter calls the functions at a binary level, directly addressing memory, directly calling memory locations, and expecting them to be functions. Effectively, it's the difference between `ffi.dlopen` and `ffi.cdef`. Here, the `dlopen` is not always safe but `cdef` is, because it passes a compiler instead of just guessing how to call a method.

CFFI or ctypes?

This really depends on what you are looking for. If you have a C library that you simply need to call and you don't need anything special, then `ctypes` is most likely the better choice. If you're actually writing your own C library and trying to link it, well, `CFFI` is probably a more convenient option. If you're not familiar with the C programming language, then I would definitely recommend `ctypes`. Alternatively, you'll find `CFFI` to be a more convenient option.

Native C/C++ extensions

The libraries that we have used so far only showed us how to access a C/C++ library within our Python code. Now we are going to look at the other side of the story—how C/C++ functions/modules within Python are actually written and how modules such as `cPickle` and `cProfile` are created.

A basic example

Before we can actually start with writing and using native C/C++ extensions, we have a few prerequisites. First of all, we need the compiler and Python headers; the instructions in the beginning of this chapter should have taken care of this for us. After that, we need to tell Python what to compile. The `setuptools` package mostly takes care of this, but we do need to create a `setup.py` file:

```
import setuptools

spam = setuptools.Extension('spam', sources=['spam.c'])

setuptools.setup(
    name='Spam',
    version='1.0',
    ext_modules=[spam],
)
```

This tells Python that we have an `Extension` object named `Spam` that will be based on `spam.c`.

Now, let's write a function in C that sums all perfect squares (`2*2`, `3*3`, and so on) up to a given number. The Python code will look like this:

```
def sum_of_squares(n):
    sum = 0

    for i in range(n):
        if i * i < n:
            sum += i * i
        else:
            break

    return sum
```

The raw C version of this code would look something like this:

```
long sum_of_squares(long n){
    long sum = 0;
```

```
    /* The actual summing code */
    for(int i=0; i<n; i++){
        if((i * i) < n){
            sum += i * i;
        }else{
            break;
        }
    }

    return sum;
}
```

And the Python C version looks like this:

```
#include <Python.h>

static PyObject* spam_sum_of_squares(PyObject *self, PyObject
        *args){
    /* Declare the variables */
    int n;
    int sum = 0;

    /* Parse the arguments */
    if(!PyArg_ParseTuple(args, "i", &n)){
        return NULL;
    }

    /* The actual summing code */
    for(int i=0; i<n; i++){
        if((i * i) < n){
            sum += i * i;
        }else{
            break;
        }
    }

    /* Return the number but convert it to a Python object first
     */
    return PyLong_FromLong(sum);
}

static PyMethodDef spam_methods[] = {
    /* Register the function */
    {"sum_of_squares", spam_sum_of_squares, METH_VARARGS,
     "Sum the perfect squares below n"},
```

```
        /* Indicate the end of the list */
        {NULL, NULL, 0, NULL},
    };

    static struct PyModuleDef spam_module = {
        PyModuleDef_HEAD_INIT,
        "spam", /* Module name */
        NULL, /* Module documentation */
        -1, /* Module state, -1 means global. This parameter is
               for sub-interpreters */
        spam_methods,
    };

    /* Initialize the module */
    PyMODINIT_FUNC PyInit_spam(void){
        return PyModule_Create(&spam_module);
    }
```

It looks quite complicated, but it's really not that hard. There is just a lot of overhead in this case because we only have a single function. Generally, you would have several functions, in which case you only need to expand the `spam_methods` array and create the functions. The next paragraph will explain the code in more detail, but first let's look at how to run our first example. We need to build and install the module:

```
# python setup.py build install
running build
running build_ext
running install
running install_lib
running install_egg_info
Removing lib/python3.5/site-packages/Spam-1.0-py3.5.egg-info
Writing lib/python3.5/site-packages/Spam-1.0-py3.5.egg-info
```

Now, let's create a little test script to time the difference between the Python version and the C version:

```
    import sys
    import spam
    import timeit


    def sum_of_squares(n):
        sum = 0
```

```
        for i in range(n):
            if i * i < n:
                sum += i * i
            else:
                break

        return sum


    if __name__ == '__main__':
        c = int(sys.argv[1])
        n = int(sys.argv[2])
        print('%d executions with n: %d' % (c, n))
        print('C sum of squares: %d took %.3f seconds' % (
            spam.sum_of_squares(n),
            timeit.timeit('spam.sum_of_squares(n)', number=c,
                          globals=globals()),
        ))
        print('Python sum of squares: %d took %.3f seconds' % (
            sum_of_squares(n),
            timeit.timeit('sum_of_squares(n)', number=c,
                          globals=globals()),
        ))
```

And now let's execute it:

```
# python3 test_spam.py 10000 1000000
10000 executions with n: 1000000
C sum of squares: 332833500 took 0.008 seconds
Python sum of squares: 332833500 took 1.778 seconds
```

Perfect! Exactly the same results but more than 200 times faster!

C is not Python – size matters

The Python language makes programming so easy that you might forget about the underlying data structures at times; with C, you can't afford to do that. Just take our example from the previous chapter but with different parameters:

```
# python3 test_spam.py 1000 10000000
1000 executions with n: 10000000
C sum of squares: 1953214233 took 0.002 seconds
Python sum of squares: 10543148825 took 0.558 seconds
```

It's still very fast, but what happened to the numbers? The Python and C versions give different results, `1953214233` versus `10543148825`. This is caused by integer overflows in C. Whereas Python numbers can essentially have any size, with C, a regular number has a fixed size. How much you get depends on the type you use (`int`, `long`, and so on) and your architecture (32-bit, 64-bit, and so on), but it's definitely something to be careful with. It might be hundreds of times faster in some cases, but that is meaningless if the results are incorrect.

We can increase the size a bit, of course. This makes it better:

```
static PyObject* spam_sum_of_squares(PyObject *self, PyObject *args){
    /* Declare the variables */
    unsigned long long int n;
    unsigned long long int sum = 0;

    /* Parse the arguments */
    if(!PyArg_ParseTuple(args, "K", &n)){
        return NULL;
    }

    /* The actual summing code */
    for(unsigned long long int i=0; i<n; i++){
        if((i * i) < n){
            sum += i * i;
        }else{
            break;
        }
    }

    /* Return the number but convert it to a Python object first */
    return PyLong_FromUnsignedLongLong(sum);
}
```

If we test it now, we realize that it works great:

```
# python3 test_spam.py 1000 100000001000 executions with n: 10000000
C sum of squares: 10543148825 took 0.002 seconds
Python sum of squares: 10543148825 took 0.635 seconds
```

Unless we make the number even larger:

```
# python3 test_spam.py 1 100000000000000
~/Dropbox/Mastering Python/code/h14
1 executions with n: 100000000000000
C sum of squares: 1291890006563070912 took 0.006 seconds
Python sum of squares: 333333283333335000000 took 2.081 seconds
```

So how can you fix this? The simple answer is that you can't. The complex answer is that you can if you use a different data type to store your data. The C language by itself doesn't have the "big number support" that Python has. Python supports infinitely large numbers by combining several regular numbers in the actual memory. Within C, there are no commonly available provisions for this, so there is simply no easy way to get this working. But we can check for errors instead:

```
static unsigned long long int get_number_from_object(int* overflow,
PyObject* some_very_large_number){
    return PyLong_AsLongLongAndOverflow(sum, overflow);
}
```

Note that this only works for `PyObject*`, which means it doesn't work for internal C overflows. But you can, of course, just keep the original Python long around and perform operations on that instead. So, you do have big number support in C without too much effort.

The example explained

We have seen the results from our example, but if you're not familiar with the Python C API, you might be confused as to why the function parameters look the way they do. The basic calculations within `spam_sum_of_squares` are identical to the regular C `sum_of_squares` function, but there are a few small differences. Firstly, the type definition for a function using the Python C API should look something like this:

```
static PyObject* spam_sum_of_squares(PyObject *self, PyObject
        *args)
```

static

This means that the function is `static`. A function that's static can be called only from the same translation unit within the compiler. This effectively results in a function that cannot be linked from other modules, which allows the compiler to optimize a bit further. Since functions in C are global by default, this can be very useful to prevent collisions. Just to be sure, however, we have prefixed the function name with `spam_` to indicate that this function comes from the `spam` module.

Be careful not to confuse the word `static` here with the `static` before a variable. They are completely different beasts. A `static` variable means that the variable that will exist for the entire runtime of the program instead of the runtime of just the function.

PyObject*

The `PyObject` type is the basic type for Python data types, which means that all Python objects can be cast to `PyObject*` (the `PyObject` pointer). Effectively, it only tells the compiler what kind of properties to expect, which can be used later for type identification and memory management. Instead of direct access to `PyObject*`, it is generally a better idea to use the available macros, such as `Py_TYPE(some_object)`. Internally, this expands to `(((PyObject*)(o))->ob_type)`, which is why the macro is generally a better idea. Besides being unreadable, a typo can easily happen.

The list of properties is long and depends greatly on the type of object. For those, I would like to refer to the Python documentation:

`https://docs.python.org/3/c-api/typeobj.html`

The entire Python C API could fill a book of its own, but it is luckily well documented within the Python manual. The usage, on the other hand, might be less obvious.

Parsing arguments

With regular C and Python, you specify the arguments explicitly, since variable-sized arguments are a bit tricky with C. This is because they need to be parsed separately. `PyObject* args` is the reference to objects containing the actual values. To parse these, you need to know how many and which type of variables to expect. In the example, we used the `PyArg_ParseTuple` function, which parses the arguments as positional arguments only, but it is quite easily possible to parse named arguments as well using `PyArg_ParseTupleAndKeywords` or `PyArg_VaParseTupleAndKeywords`. The difference between the last two is that the first one uses a variable number of arguments to specify the destination and the latter uses a `va_list` to set the values to. But first, let's analyze the code from the actual example:

```
if(!PyArg_ParseTuple(args, "i", &n)){
    return NULL;
}
```

We know that `args` is the object containing the reference to the actual arguments. The `"i"` is a format string, which in this case will try to parse a single integer. And `&n` tells the function to store the value at the memory address of the `n` variable.

The format string is the important part here. Depending on the character, you get a different data type, but there are many; `i` specifies a regular integer, and `s` converts your variable to a c-string (actually a `char*`, which is a null-terminated character array). It should be noted that this function is, luckily, smart enough to take overflows into consideration as well.

Parsing multiple arguments is quite similar; you simply need to add multiple characters to the format string and multiple destination variables:

```
PyObject* callback;
int n;

/* Parse the arguments */
if(!PyArg_ParseTuple(args, "Oi", &callback, &n)){
    return NULL;
}
```

The version with keyword arguments is similar but requires a few more code changes as the list of methods needs to be informed that the function takes keyword arguments. Otherwise, the `kwargs` parameter would never arrive:

```
static PyObject* function(
        PyObject *self,
        PyObject *args,
        PyObject *kwargs){
    /* Declare the variables */
    int sum = 0;

    PyObject* callback;
    int n;

    static char* keywords[] = {"callback", "n", NULL};

    /* Parse the arguments */
    if(!PyArg_ParseTupleAndKeywords(args, kwargs, "Oi", keywords,
                &callback, &n)){
        return NULL;
    }

    Py_RETURN_NONE;
}

static PyMethodDef methods[] = {
    /* Register the function with kwargs */
    {"function", function, METH_VARARGS | METH_KEYWORDS,
     "Some kwargs function"},
    /* Indicate the end of the list */
    {NULL, NULL, 0, NULL},
};
```

Note that this still supports normal arguments, but keyword arguments are also supported now.

C is not Python – errors are silent or lethal

As we saw in the previous example, integer overflows are not something you will generally notice, and unfortunately there's no good cross-platform way to catch them. However, those are actually the easier errors to handle; the worst one is generally memory management. With Python, if you get an error, you will get an exception that you can catch. But with C, you can't really handle it gracefully. Take a division by zero for example:

```
# python3 -c '1/0'
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ZeroDivisionError: division by zero
```

This is simple enough to catch with `try: ... except ZeroDivisionError: ...`. With C on the other hand, if you get a bad error, it will kill your entire process. But debugging C code is what C compilers have debuggers for, and to find the cause of the error, you can use the `faulthandler` module discussed in *Chapter 11*, *Debugging – Solving the Bugs*. Right now, let's see how we can properly throw errors from C. Let's use the `spam` module from earlier, but for brevity, we will omit the rest of the C code:

```
    static PyObject* spam_eggs(PyObject *self, PyObject *args){
        PyErr_SetString(PyExc_RuntimeError, "Too many eggs!");
        return NULL;
    }

    static PyMethodDef spam_methods[] = {
        /* Register the function */
        {"eggs", spam_eggs, METH_VARARGS,
         "Count the eggs"},
        /* Indicate the end of the list */
        {NULL, NULL, 0, NULL},
    };
```

Here is the execution:

```
# python3 setup.py clean build install
...
# python3 -c 'import spam; spam.eggs()'
Traceback (most recent call last):
  File "<string>", line 1, in <module>
RuntimeError: Too many eggs!
```

The syntax is slightly different—`PyErr_SetString` instead of `raise`—but it's the same basic principle, luckily.

Calling Python from C – handling complex types

We have seen how to call C functions from Python, but now let's try Python from C and back. Instead of using the readily available `sum` function, we will build one of our own with a callback and handling of any type of iterable. While this sounds simple enough, it does actually require a bit of type meddling as you can only expect `PyObject*` as arguments. This is contrary to the simple types, such as integers, chars, and strings, which are immediately converted to the native Python version:

```
static PyObject* spam_sum(PyObject* self, PyObject* args){
    /* Declare all variables, note that the values for sum and
     * callback are defaults in the case these arguments are not
     * specified */
    long long int sum = 0;
    int overflow = 0;
    PyObject* iterator;
    PyObject* iterable;
    PyObject* callback = NULL;
    PyObject* value;
    PyObject* item;

    /* Now we parse a PyObject* followed by, optionally
     * (the | character), a PyObject* and a long long int */
    if(!PyArg_ParseTuple(args, "O|OL", &iterable, &callback,
                &sum)){
        return NULL;
    }

    /* See if we can create an iterator from the iterable. This is
     * effectively the same as doing iter(iterable) in Python */
    iterator = PyObject_GetIter(iterable);
    if(iterator == NULL){
        PyErr_SetString(PyExc_TypeError,
                "Argument is not iterable");
        return NULL;
    }

    /* Check if the callback exists or wasn't specified. If it was
     * specified check whether it's callable or not */
```

```
        if(callback != NULL && !PyCallable_Check(callback)){
            PyErr_SetString(PyExc_TypeError,
                    "Callback is not callable");
            return NULL;
        }

        /* Loop through all items of the iterable */
        while((item = PyIter_Next(iterator))){
            /* If we have a callback available, call it. Otherwise
             * just return the item as the value */
            if(callback == NULL){
                value = item;
            }else{
                value = PyObject_CallFunction(callback, "O", item);
            }

            /* Add the value to sum and check for overflows */
            sum += PyLong_AsLongLongAndOverflow(value, &overflow);
            if(overflow > 0){
                PyErr_SetString(PyExc_RuntimeError,
                        "Integer overflow");
                return NULL;
            }else if(overflow < 0){
                PyErr_SetString(PyExc_RuntimeError,
                        "Integer underflow");
                return NULL;
            }

            /* If we were indeed using the callback, decrease the
             * reference count to the value because it is a separate
             * object now */
            if(callback != NULL){
                Py_DECREF(value);
            }
            Py_DECREF(item);
        }
        Py_DECREF(iterator);

        return PyLong_FromLongLong(sum);
    }
```

Make sure you note the `PyDECREF` calls, which ensure that you don't leak these objects. Without them, the objects will stay in use and the Python interpreter won't be able to clear them.

This function is callable in three different ways:

```
>>> import spam
>>> x = range(10)
>>> spam.sum(x)
45
>>> spam.sum(x, lambda y: y + 5)
95
>>> spam.sum(x, lambda y: y + 5, 5)
100
```

Another important issue is that even though we catch overflow errors when converting to `long long int`, this code is still not safe. If we sum even two very large numbers (close to the `long long int` limit), we will still have an overflow:

```
>>> import spam
>>> n = (2 ** 63) - 1
>>> x = n,
>>> spam.sum(x)
9223372036854775807
>>> x = n, n
>>> spam.sum(x)
-2
```

Summary

In this chapter, you learned the most important aspects of writing code that uses `ctypes`, `CFFI`, and how to extend the Python functionality using native C. These topics can be extensive enough to fill books on their own, but you should have a grasp of the most important topics now. Even though you are able to create C/C++ extensions now, I still recommend that you avoid these as much as possible. This is because bugs are so easily made by not being careful enough. It is actually likely that at least some of the examples given in this chapter contain bugs when it comes to memory management and can crash your Python interpreter when given the wrong input. Unfortunately, this is a side effect of C. A tiny mistake can have a huge impact.

While building the examples in this chapter, you may have noticed that we used a `setup.py` file and imported from the `setuptools` library. This is what the next chapter will cover—packaging your code into an installable Python library and distributing it on the Python package index.

15
Packaging – Creating Your Own Libraries or Applications

The chapters thus far have covered how to write, test and, debug the Python code. With all of that, there is only one thing that remains, that is packaging and distributing your Python libraries /and applications. To create installable packages we will use the `setuptools` package which is bundled with Python these days. If you have created packages before, you might remember `distribute` and `distutils2`, but it is very important to remember that these have all been replaced by `setuptools` and `distutils` and you shouldn't use them anymore!

What types of program can we package with `setuptools`? We will show you several cases:

- Regular packages
- Packages with data
- Installing executables and custom `setuptools` commands
- Running tests on the package
- Packages containing C/C++ extensions

Installing packages

Before we actually get started, it is important to know how to install a package properly. There are at least four different options for installing a package. The first and most obvious is by using the plain `pip` command:

```
pip install package
```

This can also be achieved by using `setup.py` directly:

```
cd package
python setup.py install
```

This installs the package within your Python environment which would be the likely `virtualenv`/`venv` if you are using it or the global environment otherwise.

For development however, this is not recommended. To test your code, you would need to either reinstall the package for every test or modify the files within the Python's `site-packages` directory, which would mean it would be outside of your revision control system as well. That's where the development installs come in; instead of copying the package files to the Python package directory, they simply install a link within the `site-packages` directory to the path where the package is actually located. This allows you to modify the code and immediately see the results in the scripts and applications you run without the need to reinstall your code after each change.

As is the case with a regular install, both `pip` and `setup.py` versions are available:

```
pip install -e package_directory
```

And the `setup.py` version:

```
cd package_directory
python setup.py develop
```

Setup parameters

The previous chapters have actually already shown us a couple of examples, but let's reiterate and review what the most important parts actually do. The core function you will be using in this entire chapter is `setuptools.setup`.

For the most simple packages, the `distutils` package bundled with Python will be sufficient as well, but I recommend `setuptools` regardless. The `setuptools` package has many great features that `distutils` lacks and nearly all Python environments will have `setuptools` available anyhow.

Before we continue, make sure you have the latest version of both `pip` and `setuptools`:

```
pip install -U pip setuptools
```

The `setuptools` and `distutils` packages have changed significantly over the last few years and the documentation/examples written before 2014 are most likely out of date. Be careful not to implement deprecated examples and skip any documentation/examples using `distutils`.

Now that we have all the prerequisites, let's create an example containing the most important fields with inline documentation:

```
import setuptools


if __name__ == '__main__':
    setuptools.setup(
        name='Name',
        version='0.1',

        # This automatically detects the packages in the specified
        # (or current directory if no directory is given).
        packages=setuptools.find_packages(),

        # The entry points are the big difference between
        # setuptools and distutils, the entry points make it
        # possible to extend setuptools and make it smarter and/or
        # add custom commands.
        entry_points={

            # The following would add: python setup.py
            # command_name
            'distutils.commands': [
                'command_name = your_package:YourClass',
            ],

            # The following would make these functions callable as
            # standalone scripts. In this case it would add the
            # spam command to run in your shell.
            'console_scripts': [
                'spam = your_package:SpamClass',
            ],
        },

        # Packages required to use this one, it is possible to
        # specify simply the application name, a specific version
        # or a version range. The syntax is the same as pip
```

```
# accepts.
install_requires=['docutils>=0.3'],

# Extra requirements are another amazing feature of
# setuptools, it allows people to install extra
# dependencies if you are interested. In this example
# doing a "pip install name[all]" would install the
# python-utils package as well.
extras_requires={
    'all': ['python-utils'],
},

# Packages required to install this package, not just for
# running it but for the actual install. These will not be
# installed but only downloaded so they can be used during
# the install. The pytest-runner is a useful example:
setup_requires=['pytest-runner'],

# The requirements for the test command. Regular testing
# is possible through: python setup.py test The Pytest
# module installs a different command though: python
# setup.py pytest
tests_require=['pytest'],

# The package_data, include_package_data and
# exclude_package_data arguments are used to specify which
# non-python files should be included in the package. An
# example would be documentation files.  More about this
# in the next paragraph
package_data={
    # Include (restructured text) documentation files from
    # any directory
    '': ['*.rst'],
    # Include text files from the eggs package:
    'eggs': ['*.txt'],
},

# If a package is zip_safe the package will be installed
# as a zip file. This can be faster but it generally
# doesn't make too much of a difference and breaks
# packages if they need access to either the source or the
# data files. When this flag is omitted setuptools will
# try to autodetect based on the existance of datafiles
# and C extensions. If either exists it will not install
```

```
        # the package as a zip. Generally omitting this parameter
        # is the best option but if you have strange problems with
        # missing files, try disabling zip_safe.
        zip_safe=False,

        # All of the following fileds are PyPI metadata fields.
        # When registering a package at PyPI this is used as
        # information on the package page.
        author='Rick van Hattem',
        author_email='wolph@wol.ph',

        # This should be a short description (one line) for the
        # package
        description='Description for the name package',

        # For this parameter I would recommend including the
        # README.rst

        long_description='A very long description',
        # The license should be one of the standard open source
        # licenses: https://opensource.org/licenses/alphabetical
        license='BSD',

        # Homepage url for the package
        url='https://wol.ph/',
    )
```

That was quite a lot of code and comments, but it covers most of the options you will ever encounter in real-life packages. The most interesting and versatile parameters discussed here will be covered in the following sections separately.

Additional documentation can be found in the `pip` and `setuptools` documentation, as well as in the Python Packaging User Guide:

- `http://pythonhosted.org/setuptools/`
- `https://pip.pypa.io/en/stable/`
- `http://python-packaging-user-guide.readthedocs.org/en/latest/`

Packages

In our example, we simply use `packages=setuptools.find_packages()`. In most cases this will work just fine, but it's important to understand what it does. The `find_packages` function looks through all the directories within the given directory and adds it to the list if it has an `__init__.py` file inside. So instead of `find_packages()` you can generally use `['your_package']` instead. If you have several packages however, that tends to get tedious. That's where `find_packages()` is useful; simply specify some inclusion parameters (second parameter) or some exclusion parameters (third parameter) and you'll have all the relevant packages within your project. For example:

```
packages = find_packages(exclude=['tests', 'docs'])
```

Entry points

The `entry_points` parameter is arguably the most useful feature of `setuptools`. It allows you to add hooks to many things within `setuptools` but the most useful two are the possibility to add both the command line and GUI commands and to extend the `setuptools` commands. The command line and GUI commands will even be converted to executables on Windows. The example in the first section already demonstrated both the features:

```
entry_points={
    'distutils.commands': [
        'command_name = your_package:YourClass',
    ],
    'console_scripts': [
        'spam = your_package:SpamClass',
    ],
},
```

This demonstration only shows how to call the functions but it doesn't show the actual functions.

Creating global commands

The first, a simple example, is nothing special at all; just a function that gets called as a regular `main` function where you need to specify `sys.argv` yourself (or better, use `argparse`). This is the `setup.py` file:

```
import setuptools


if __name__ == '__main__':
```

```
    setuptools.setup(
        name='Our little project',
        entry_points={
            'console_scripts': [
                'spam = spam.main:main',
            ],
        },
    )
```

And, of course, here's the `spam/main.py` file:

```
import sys


def main():
    print('Args:', sys.argv)
```

Be sure not to forget to create a `spam/__init__.py` file. It can be empty but it needs to exist for Python to know that it's a package.

Now, let's give it a try by installing the package:

```
# pip install -e .
Installing collected packages: Our-little-project
  Running setup.py develop for Our-little-project
Successfully installed Our-little-project
# spam 123 abc
Args: ['~/envs/mastering_python/bin/spam', '123', 'abc']
```

See how easy it was to create a `spam` command that installs in your regular command line shell! On Windows it will actually give you an executable which will be added to your path but regardless of the platform it will be as a separate executable that's callable.

Custom setup.py commands

Writing custom `setup.py` commands can be very useful. One example is `sphinx-pypi-upload-2` which I use in all my packages and is my fork of the unmaintained `sphinx-pypi-upload` package. It's a package that makes it trivial to build and upload Sphinx documentation to the Python package index, which is very useful when distributing your packages. With the `sphinx-pypi-upload-2` package you can do the following (which I do when distributing any of the packages I maintain):

```
python setup.py sdist bdist_wheel upload build_sphinx upload_sphinx
```

This command builds your package and uploads it to PyPI, and builds the Sphinx documentation and uploads it to PyPI as well.

But you want to see how this works, of course. First, here's `setup.py` for our `spam` command:

```
import setuptools


if __name__ == '__main__':
    setuptools.setup(
        name='Our little project',
        entry_points={
            'distutils.commands': [
                'spam = spam.command:SpamCommand',
            ],
        },
    )
```

Second, the `SpamCommand` class. The basic essentials are inheriting `setuptools.Command` and making sure to implement all the needed methods. Note that all of these need to be implemented but can be left empty if desired. Here is the `spam/command.py` file:

```
import setuptools


class SpamCommand(setuptools.Command):
    description = 'Make some spam!'
# Specify the commandline arguments for this command here. This
# parameter uses the getopt module for parsing'
    user_options = [
        ('spam=', 's', 'Set the amount of spams'),
    ]

    def initialize_options(self):
# This method can be used to set default values for the
# options. These defaults can be overridden by
# command-line, configuration files and the setup script
# itself.
        self.spam = 3

    def finalize_options(self):
# This method allows you to override the values for the
# options, useful for automatically disabling
# incompatible options and for validation.
```

```
            self.spam = max(0, int(self.spam))

        def run(self):
            # The actual running of the command.
            print('spam' * self.spam)
```

Executing it is simple enough:

```
# pip install -e .
Installing collected packages: Our-little-project
  Running setup.py develop for Our-little-project
Successfully installed Our-little-project-0.0.0
# python setup.py --help-commands
[...]
Extra commands:
  [...]
  spam              Make some spam!
  test              run unit tests after in-place build
  [...]

usage: setup.py [global_opts] cmd1 [cmd1_opts] [cmd2 [cmd2_opts] ...]
   or: setup.py --help [cmd1 cmd2 ...]
   or: setup.py --help-commands
   or: setup.py cmd -help

# python setup.py --help spam
Common commands: (see '--help-commands' for more)

[...]

Options for 'SpamCommand' command:
  --spam (-s)  Set the amount of spams

usage: setup.py [global_opts] cmd1 [cmd1_opts] [cmd2 [cmd2_opts] ...]
   or: setup.py --help [cmd1 cmd2 ...]
   or: setup.py --help-commands
```

```
   or: setup.py cmd --help


# python setup.py spam
running spam
spamspamspam
# python setup.py spam -s 5
running spam
spamspamspamspamspam
```

There are very few cases where you will actually need the custom `setup.py` commands, but the example is still useful since it is currently an undocumented part of `setuptools`.

Package data

In most cases you probably won't have to include the package data, but in the cases where you do need data to go with your package, there are a few different options. First, it is important to know which files are included in your package by default:

- Python source files in the package directories recursively
- The `setup.py` and `setup.cfg` files
- Tests: `test/test*.py`
- All `*.txt` and `*.py` files in the `examples` directory
- All `*.txt` files in the root directory

So after the defaults, we have the first solution: the `package_data` argument to the setup function. The syntax for that is simple enough, a dictionary where the keys are the packages and the values are the patterns to include:

```
package_data = {
    'docs': ['*.rst'],
}
```

The second solution is using a `MANIFEST.in` file. This file contains patterns to include, exclude, and more. The `include` and `exclude` commands use patterns to match. These patterns are glob-style patterns (see the `glob` module for documentation: `https://docs.python.org/3/library/glob.html`) and have three variants for both the include and exclude commands:

- `include`/`exclude`: These commands only work for the given path and nothing else

- `recursive-include/recursive-exclude`: These commands are similar to the `include/exclude` commands but process the given paths recursively
- `global-include/global-exclude`: Be very careful with these, they will include or exclude these files anywhere within the source tree

Besides the `include/exclude` commands, there are also two others; the `graft` and `prune` commands which include or exclude directories including all the files under a given directory. This can be useful for tests and documentation since they can include non-standard files. Beyond those examples, it's almost always better to explicitly include the files you need and ignore all the others. Here's an example `MANIFEST.in`:

```
# Comments can be added with a hash tag
include LICENSE CHANGES AUTHORS

# Include the docs, tests and examples completely
graft docs
graft tests
graft examples

# Always exclude compiled python files
global-exclude *.py[co]

# Remove documentation builds
prune docs/_build
```

Testing packages

In *Chapter 10, Testing and Logging – Preparing for Bugs*, the testing chapter, we saw a few of the many testing systems for Python. As you might suspect, at least some of these have `setup.py` integration.

Unittest

Before we start, we should create a test script for our package. For actual tests, look at *Chapter 10, Testing and Logging – Preparing for Bugs*, the testing chapter. In this case, we will just use a no-op test, `test.py`:

```
import unittest


class Test(unittest.TestCase):

    def test(self):
        pass
```

The standard `python setup.py test` command will run the regular `unittest` command:

```
# python setup.py -v test
running test
running "unittest --verbose"
running egg_info
writing Our_little_project.egg-info/PKG-INFO
writing dependency_links to Our_little_project.egg-info/dependency_links.
txt
writing top-level names to Our_little_project.egg-info/top_level.txt
writing entry points to Our_little_project.egg-info/entry_points.txt
reading manifest file 'Our_little_project.egg-info/SOURCES.txt'
writing manifest file 'Our_little_project.egg-info/SOURCES.txt'
running build_ext
test (test.Test) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

It is possible to tell `setup.py` to use different tests using the `--test-module`, `--test-suite`, or `--test-runner` arguments. While these are easy enough to use, I recommend skipping the regular `test` command and trying `nose` or `py.test` instead.

py.test

The `py.test` package has several methods of integration: `pytest-runner`, your own test command, and the deprecated method of generating a `runtests.py` script to test. If one of your packages is still using `runtests.py`, I strongly recommend switching to one of the other options.

But before we discuss the other options, let's make sure we have some tests. So let's create a test in our package. We will store it in `test_pytest.py`:

```
def test_a():
    pass


def test_b():
    pass
```

Now, the other test options. Since the custom command doesn't really add much and actually makes things more complicated, we will skip that. If you want to customize how the tests are being run, use the `pytest.ini` and `setup.cfg` files instead. The best option is `pytest-runner` which makes running tests a trivial task:

```
# pip install pytest-runner
Collecting pytest-runner
  Using cached pytest_runner-2.7-py2.py3-none-any.whl
Installing collected packages: pytest-runner
Successfully installed pytest-runner-2.7
# python setup.py pytest
running pytest
running egg_info
writing top-level names to Our_little_project.egg-info/top_level.txt
writing dependency_links to Our_little_project.egg-info/dependency_links.
txt
writing entry points to Our_little_project.egg-info/entry_points.txt
writing Our_little_project.egg-info/PKG-INFO
reading manifest file 'Our_little_project.egg-info/SOURCES.txt'
writing manifest file 'Our_little_project.egg-info/SOURCES.txt'
running build_ext
======================== test session starts ========================
platform darwin -- Python 3.5.1, pytest-2.8.7, py-1.4.31, pluggy-0.3.1
rootdir: h15, inifile: pytest.ini
collected 2 items

test_pytest.py ..

===================== 2 passed in 0.01 seconds ======================
```

To properly integrate this method, we should make a few changes to the `setup.py` script. They are not strictly needed but it makes things more convenient for others using your package, others that may not be aware that you are using `py.test`, for example. First, we make sure that the standard `python setup.py test` command actually runs the `pytest` command instead by modifying `setup.cfg`:

```
[aliases]
test=pytest
```

Second, we want to make sure that the `setup.py` command installs the packages we need to run the `py.test` tests. To do that, we need to modify `setup.py` as well:

```
import setuptools


if __name__ == '__main__':
    setuptools.setup(
        name='Our little project',
        entry_points={
            'distutils.commands': [
                'spam = spam.command:SpamCommand',
            ],
        },
        setup_requires=['pytest-runner'],
        tests_require=['pytest'],
    )
```

The beauty of this approach is that the regular `python setup.py test` command works and all needed requirements are automatically installed before running the tests. Because the `pytest` requirement is only in the `tests_require` section however, they will not be installed if the test command isn't run. The only package that will always be installed is the `pytest-runner` package and that's a really light package so it will be very light to install and run.

Nosetests

The `nose` package handles the installation only and is slightly different from `py.test`. The only difference is that `py.test` has a separate `pytest-runner` package for the test runner and nose package has a built-in `nosetests` command. So without further ado, here is the nose version:

```
# pip install nose
Collecting nose
  Using cached nose-1.3.7-py3-none-any.whl
Installing collected packages: nose
Successfully installed nose-1.3.7
# python setup.py nosetests
running nosetests
running egg_info
writing top-level names to Our_little_project.egg-info/top_level.txt
writing entry points to Our_little_project.egg-info/entry_points.txt
```

```
writing Our_little_project.egg-info/PKG-INFO
writing dependency_links to Our_little_project.egg-info/dependency_lin
ks.txt
reading manifest file 'Our_little_project.egg-info/SOURCES.txt'
writing manifest file 'Our_little_project.egg-info/SOURCES.txt'
..
----------------------------------------------------------------------
Ran 2 tests in 0.006s

OK
```

C/C++ extensions

The previous chapter already covered this somewhat, as it's a requirement to compile the C/C++ files. But that chapter didn't explain what and how the `setup.py` was doing in this case.

For convenience, we will repeat the `setup.py` file:

```
import setuptools

spam = setuptools.Extension('spam', sources=['spam.c'])

setuptools.setup(
    name='Spam',
    version='1.0',
    ext_modules=[spam],
)
```

Before you start with these extensions, you should learn the following commands:

- `build`: This is actually not a C/C++ specific build function (try `build_clib` for that) but a combined build function to build everything within `setup.py`.
- `clean`: This cleans the results from the `build` command. This is generally not needed but sometimes the detection of files that need to be recompiled to work is incorrect. So if you encounter strange or unexpected issues, try cleaning the project first.

Regular extensions

The `setuptools.Extension` class tells `setuptools` that a module named `spam` uses the source file `spam.c`. This is just the simplest version of an extension, a name, and a list of sources, but in many cases you are going to need more than the simple case.

One example is the `pillow` library which detects the libraries available on the system and adds extensions based on that. But because these extensions include libraries, some extra compilation flags are required. The basic PIL module itself doesn't appear too involved but the libs are actually filled with all auto-detected libraries with the matching macro definitions:

```
exts = [(Extension("PIL._imaging", files, libraries=libs,
                   define_macros=defs))]
```

The `freetype` extension has something similar:

```
if feature.freetype:
    exts.append(Extension(
        "PIL._imagingft", ["_imagingft.c"],
libraries=["freetype"]))
```

Cython extensions

The `setuptools` library is actually a bit smarter than the regular `distutils` library when it comes to extensions. It actually adds a little trick to the `Extension` class. Remember the brief introduction to `Cython` in *Chapter 12*, *Performance – Tracking and Reducing Your Memory and CPU Usage* about performance? The `setuptools` library makes it a bit more convenient to compile those. The `Cython` manual recommends you to use something similar to the following code:

```
from distutils.core import setup
from Cython.Build import cythonize

setup(
    ext_modules = cythonize("eggs.pyx")
)
```

Here `eggs.pyx` contains:

```
def make_eggs(int n):
    print('Making %d eggs: %s' % (n, n * 'eggs '))
```

The problem with this approach is that `setup.py` will break unless you have `Cython` installed:

```
# python setup.py build
Traceback (most recent call last):
```

```
  File "setup.py", line 2, in <module>
    import Cython
ImportError: No module named 'Cython'
```

To prevent that issue, we are just going to let `setuptools` handle this:

```
import setuptools

eggs = setuptools.Extension('eggs', sources=['eggs.pyx'])

setuptools.setup(
    name='Eggs',
    version='1.0',
    ext_modules=[eggs],
    setup_requires=['Cython'],
)
```

Now `Cython` will be automatically installed if needed and the code will work just fine:

```
# python setup.py build
running build
running build_ext
cythoning eggs.pyx to eggs.c
building 'eggs' extension
...
# python setup.py develop
running develop
running egg_info
creating Eggs.egg-info
writing dependency_links to Eggs.egg-info/dependency_links.txt
writing top-level names to Eggs.egg-info/top_level.txt
writing Eggs.egg-info/PKG-INFO
writing manifest file 'Eggs.egg-info/SOURCES.txt'
reading manifest file 'Eggs.egg-info/SOURCES.txt'
writing manifest file 'Eggs.egg-info/SOURCES.txt'
running build_ext
skipping 'eggs.c' Cython extension (up-to-date)
copying build/... ->
Creating Eggs.egg-link (link to .)
```

```
Adding Eggs 1.0 to easy-install.pth file


Installed Eggs
Processing dependencies for Eggs==1.0
Finished processing dependencies for Eggs==1.0
# python -c 'import eggs; eggs.make_eggs(3)'
Making 3 eggs: eggs eggs eggs
```

For development purposes however, `Cython` also offers a simpler method which doesn't require manual building. First, to make sure we are actually using this method, let's install `Cython` and uninstall and clean `eggs` completely:

```
# pip uninstall eggs -y
Uninstalling Eggs-1.0:
  Successfully uninstalled Eggs-1.0
# pip uninstall eggs -y
Cannot uninstall requirement eggs, not installed
# python setup.py clean
# pip install cython
```

Now let's try and run our `eggs.pyx` module:

```
>>> import pyximport
>>> pyximport.install()
(None, <pyximport.pyximport.PyxImporter object at 0x...>)
>>> import eggs
>>> eggs.make_eggs(3)
Making 3 eggs: eggs eggs eggs
```

That's how easy it is to run the `pyx` files without explicit compiling.

Wheels – the new eggs

For pure Python packages, the `sdist` (source distribution) command has always been enough. For C/C++ packages however, it is usually not that convenient. The problem with C/C++ packages is that compilation is needed unless you use a binary package. Traditionally those were generally the `.egg` files but they never really solved the issue quite right. That is why the `wheel` format has been introduced (PEP 0427), a binary package format that contains both source and binaries and can install on both Windows and OS X without requiring a compiler. As an added bonus, it installs faster for pure Python packages as well.

Implementation is luckily simple. First, install the `wheel` package:

```
# pip install wheel
```

Now you'll be able to use the `bdist_wheel` command to build your packages. The only small gotcha is that by default the packages created by Python 3 will only work on Python 3, so Python 2 installations will fall back to the `sdist` file. To fix that, you can add the following to your `setup.cfg` file:

```
[bdist_wheel]
universal = 1
```

The only important thing to note here is that in the case of C extensions, this can go wrong. The binary C extensions for Python 3 are not compatible with those from Python 2. So if you have a pure Python package and are targeting both Python 2 and 3, enable the flag. Otherwise just leave it as the default.

Distributing to the Python Package Index

Once you have everything up and running, tested, and documented, it is time to actually push the project to the **Python Package Index** (**PyPI**). Before pushing the package to PyPI, we need to make sure everything is in order.

First, let's check the `setup.py` file for issues:

```
# python setup.py check
running check
warning: check: missing required meta-data: url


warning: check: missing meta-data: either (author and author_email) or
(maintainer and maintainer_email) must be supplied
```

It seems that we forgot to specify a `url` and the `author` or `maintainer` information. Let's fill those:

```
import setuptools

eggs = setuptools.Extension('eggs', sources=['eggs.pyx'])

setuptools.setup(
    name='Eggs',
    version='1.0',
    ext_modules=[eggs],
    setup_requires=['Cython'],
    url='https://wol.ph/',
```

```
        author='Rick van Hattem (Wolph)',
        author_email='wolph@wol.ph',
    )
```

Now let's check again:

```
# python setup.py check
running check
```

Perfect! No errors and everything looks good.

Now that our `setup.py` is in order, let's try testing. Since our little test project has virtually no tests, this will come up close to empty. But if you're starting a new project, then I recommend trying to maintain 100 percent test coverage from the beginning. Implementing all the tests later is usually more difficult, and testing while you work generally makes you think more about the design decisions of the code. Running the test is easy enough:

```
# python setup.py test
running test
running egg_info
writing dependency_links to Eggs.egg-info/dependency_links.txt
writing Eggs.egg-info/PKG-INFO
writing top-level names to Eggs.egg-info/top_level.txt
reading manifest file 'Eggs.egg-info/SOURCES.txt'
writing manifest file 'Eggs.egg-info/SOURCES.txt'
running build_ext
skipping 'eggs.c' Cython extension (up-to-date)
copying build/... ->

----------------------------------------------------------------------
Ran 0 tests in 0.000s

OK
```

Now that we have all in check, the next step is building the documentation. As mentioned earlier, the `sphinx` and `sphinx-pypi-upload-2` packages can help here:

```
# python setup.py build_sphinx
running build_sphinx
Running Sphinx v1.3.5
...
```

Once we are certain that everything is correct, we can build the package and upload it to PyPI. For pure Python releases, you can use the `sdist` (source distribution) command. For a package that uses a native installer, there are a few options, such as `bdist_wininst` and `bdist_rpm`, available. I personally use the following for nearly all my packages:

```
# python setup.py build_sphinx upload_sphinx sdist bdist_wheel upload
```

This automatically builds the Sphinx documentation, uploads the documentation to PyPI, builds the package with the source, and uploads the package with the source.

This will obviously only succeed if you are the owner of that specific package and are authorized with PyPI.

Before you can upload the packages, you need to register the package on PyPI. This can be done using the `register` command, but since that immediately registers the package at the PyPI servers, it should not be used while testing.

Summary

After reading this chapter, you should be able to create Python packages containing not only pure-Python files but also extra data, compiled C/C++ extensions, documentation, and tests. With all these tools at your disposal, you are now able to make high quality Python packages that can easily be reused in other projects and packages.

The Python infrastructure makes it really quite easy to create new packages and split your project into multiple subprojects. This allows you to create simple and reusable packages with fewer bugs because everything is easily testable. While you shouldn't go overboard with splitting up the packages, if a script or module has a purpose of its own then it's a candidate for packaging separately.

With this chapter we have come to the end of the book. I sincerely hope you enjoyed reading it and have learned about new and interesting topics. Any and all feedback is greatly appreciated, so feel free to contact me through my website at `https://wol.ph/`.

Bibliography

This Learning Path is a blend of content, all packaged up keeping your journey in mind. It includes content from the following Packt products:

- Learning Python- Fabrizio Romano
- Python 3 Object-Oriented Programming , Second Edition , Dusty Phillips
- Mastering Python- Rick van Hattem

Thank you for buying

Python: Journey from Novice to Expert

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

Please check `www.PacktPub.com` for information on our titles