python™

Learn Python Programming

An in-depth introduction to
the fundamentals of Python

Third Edition

Fabrizio Romano
Heinrich Kruger

Packt>

Learn Python Programming

Third Edition

An in-depth introduction to the fundamentals of Python

Fabrizio Romano

Heinrich Kruger

Packt>

BIRMINGHAM—MUMBAI

Learn Python Programming

Third Edition

Copyright © 2021 Packt Publishing

"To Elisa, the love of my lives, and to all those who gift the world their beautiful smiles."

Fabrizio Romano

"To my wife, Debi, without whose love, support, and endless patience I would not have been able to do this."

Heinrich Kruger

Contributors

About the authors

Fabrizio Romano was born in Italy in 1975. He holds a master's degree in Computer Science Engineering from the University of Padova. He's been working as a professional software developer since 1999. Fabrizio has been part of Sohonet's Product Team since 2016. In 2020, the Television Academy honored them with an Emmy Award in Engineering Development for advancing remote collaboration.

I would like to thank everyone at Packt, and the reviewers, who helped us in making this book a success. I also want to thank Heinrich Kruger and Tom Viner for joining me in this adventure. My deepest gratitude goes to my wife-to-be, Elisa, who never made me feel like I was neglecting her, even though I was. Thank you for your love and support.

Heinrich Kruger was born in South Africa in 1981. He holds a master's degree in Computer Science from Utrecht University in the Netherlands. He has been working as a professional software developer since 2014 and has worked alongside Fabrizio in the Product Team at Sohonet since 2017.

I want to thank Fabrizio for asking me to help him with this book. It's been a great experience working with you, my friend. I would also like to thank Tom Viner and Dong-hee Na for their helpful comments, as well as everyone at Packt who helped us along the way. Most of all, I thank my wife, Debi, for all her love, encouragement, and support.

About the reviewers

Tom Viner is a principal software developer living in London. He has over 13 years' experience in building web applications and has been using Python and Django for 10 years. He has special interests in open-source software, web security, and test-driven development.

> *I would like to thank Fabrizio Romano and Heinrich Kruger for inviting me to review this book.*

Dong-hee Na is a software engineer and an open-source enthusiast. He works at Line Corporation as a backend engineer. He has professional experience in machine learning projects based on Python and C++. As for his open-source work, he focuses on the compiler and interpreter area, especially for Python-related projects. He has been a CPython core developer since 2020.

Table of Contents

Preface

The first edition of this book came out on the day of my 40th birthday. It feels like yesterday, but actually it was 6 years ago. In a few weeks, the book became a top seller, and to this day that translates into lovely messages and emails I get from readers all over the world.

A couple of years later, I wrote a second edition. That turned out to be a better book, which kept growing in sales and popularity.

And now here we are, at the third edition, and this time it won't just be me narrating the story, because for this edition I have been joined by my dear friend and colleague, Heinrich Kruger.

Together, we have reworked the book's structure. We removed what we felt didn't fit anymore, and added what we thought would benefit you the most. We have shuffled things around, amended old chapters, and written new ones. We have made sure that both our contributions and our best ideas are on each page you will read. We are both very happy about this.

I always wanted to work on a project like this with Heinrich, for whom I have felt enormous respect since I got to know him. He has brought to this book his unique perspective, his incredible talent as a software developer, and he's helped me with my English too!

Everything has been updated to Python 3.9, but of course most of the code will still work with any recent version of Python 3. The scary chapter about concurrency is gone, and the one on Web programming has been replaced with another which introduces the concept of APIs. We have also added a whole new chapter about packaging Python applications, which we feel is the perfect way to close the book.

We are confident this edition is much better than the previous ones; it's more mature, it tells a better story, and it will take you places.

One thing I am particularly happy about is that the soul of the book is still the same. This is not just a book about Python. This is, first and foremost, a book about programming. A book that aims to convey to you as much information as possible, and sometimes, for practical reasons, it does so by pointing you to the Web to dig deeper, to investigate further.

It is designed to last. It expresses concepts and information in a way that should stand the test of time, for as long as possible. We have put in a great amount of thinking to achieve that.

And it will require you to work hard. The code is available for you to download, and we do encourage you to play with it, expand it, change it, break it, and see things for yourself. We want you to develop critical thinking. We want you to be independent, empowered.

That is the soul of the book, and our hope is that wherever you are in your journey, it will help you go further, become a better programmer, in any way that is possible.

When we received the drafts from the second edition to start working on the third one, I was surprised to notice I could not find myself in those pages. Those pages have shown me how my thinking, and therefore my writing, has changed, in the past few years.

Change is interwoven in the very fabric of this universe. Everything changes, all the time. So, our wish for you is that you never fixate on opinions, that you never grow stale. Instead, we hope our work, and the way we present it to you, will help you stay flexible, smart, pragmatic, and adaptable.

We wish you good luck! And don't forget to enjoy the ride!

Who this book is for

This book is for people who have some programming experience, but not necessarily with Python. Some knowledge of basic programming concepts will come in handy, although it is not a requirement.

Even if you already have some experience with Python, this book can still be useful to you, both as a reference to Python's fundamentals, and for providing a wide range of considerations and suggestions collected over four combined decades of experience.

What this book covers

Chapter 1, *A Gentle Introduction to Python*, introduces you to fundamental programming concepts and constructs of the Python language. It also guides you through getting Python up and running on your computer.

Chapter 2, *Built-In Data Types*, introduces you to Python built-in data types. Python has a very rich set of native data types, and this chapter will give you a description and examples for each of them.

Chapter 3, *Conditionals and Iteration*, teaches you how to control the flow of code by inspecting conditions, applying logic, and performing loops.

Chapter 4, *Functions, the Building Blocks of Code*, teaches you how to write functions. Functions are essential to code reuse, to reducing debugging time, and, in general, to writing higher quality code.

Chapter 5, *Comprehensions and Generators*, introduces you to the functional aspects of Python programming. This chapter teaches you how to write comprehensions and generators, which are powerful tools that you can use to write faster, more concise code, and save memory.

Chapter 6, *OOP, Decorators, and Iterators*, teaches you the basics of object-oriented programming with Python. It shows you the key concepts and all the potentials of this paradigm. It also shows you one of the most useful features of the language: decorators.

Chapter 7, *Exceptions and Context Managers*, introduces the concept of exceptions, which represent errors that occur in applications, and how to handle them. It also covers context managers, which are very useful when dealing with resources.

Chapter 8, *Files and Data Persistence*, teaches you how to deal with files, streams, data interchange formats, and databases.

Chapter 9, *Cryptography and Tokens*, touches upon the concepts of security, hashes, encryption, and tokens, which are essential for writing secure software.

Chapter 10, *Testing*, teaches you the fundamentals of testing, and guides you through a few examples on how to test your code, in order to make it more robust, fast, and reliable.

Chapter 11, *Debugging and Profiling*, shows you the main methods for debugging and profiling code and some examples of how to apply them.

Chapter 12, *GUIs and Scripting*, guides you through an example from two different points of view: one implementation is a script, and the other one is a Graphical User Interface (GUI) application.

Chapter 13, *Data Science in Brief*, illustrates a few key concepts by means of a comprehensive example, using the powerful Jupyter Notebook.

Chapter 14, *Introduction to API Development*, introduces API development and type hinting in Python. It also provides different examples on how to consume an API.

Chapter 15, *Packaging Python Applications*, guides you through the process of preparing a project to be published, and shows you how to upload the result onto the **Python Package Index** (**PyPI**).

To get the most out of this book

You are encouraged to follow the examples in this book. You will need a computer, an internet connection, and a browser. The book is written in Python 3.9, but it should also work, for the most part, with any recent version of Python 3. We have given guidelines on how to install Python on your operating system. The procedures to do that normally get out of date quickly, so we recommend you refer to the most up-to-date guide on the Web to find precise setup instructions. We have also explained how to install all the extra libraries used in the various chapters. No particular editor is required to type the code; however, we suggest that those who are interested in following the examples should consider adopting a proper coding environment. We have offered suggestions on this matter in the first chapter.

Download the example code files

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Learn-Python-Programming-Third-Edition`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://static.packt-cdn.com/downloads/9781801815093_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: "Within the learn.pp folder, we will create a virtual environment."

A block of code is set as follows:

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

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
# 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
```

Any command-line input or output is written as follows:

```
>>> import sys
>>> print(sys.version)
```

Bold: Indicates a new term, an important word, or words that you see on the screen, for example, in menus or dialog boxes. For example: "When an error is detected during execution, it is called an **exception**."

> Warnings or important notes appear like this.

> Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com`, and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packtpub.com`.

Share Your Thoughts

Once you've read *Learn Python Programming, Third edition*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

1

A Gentle Introduction to Python

"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:

"...the process of designing and building an executable computer program to accomplish a specific computing result or to perform a specific task. Programming involves tasks such as: analysis, generating algorithms, profiling algorithms' accuracy and resource consumption, and the implementation of algorithms in a chosen programming language (commonly referred to as coding)."

(https://en.wikipedia.org/wiki/Computer_programming)

In a nutshell, **computer programming**, or coding, as it is sometimes known, 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. They need to be told everything: 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 a network or a disk; and how to react when something unforeseen happens, in the case of, say, something being broken or missing.

You can code in many different styles and languages. Is it hard? We would say *yes* and *no*. It's a bit like writing—it is something that everybody can learn. But what if you want to become a poet? Writing alone is not enough. You have to acquire a whole other set of skills, and this will involve 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 we have no doubt you can do it. Anyone can; in fact, we all program all the time, only we aren't aware of it. Take the following 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 and that 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 mug, and stir.

So, how is this programming?

Well, we gathered resources (the kettle, coffee, water, teaspoon, and mug) and we verified some conditions concerning them (the kettle is plugged in, the mug is clean, and 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 into the mug and stirring.

Can you see the parallel? We 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.

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. You need to learn a language as well so that the computer can be instructed.

So, this is what this book is for. We'll show you one way in which you can code successfully, and we'll try to do that by means of many simple but focused examples (our favorite kind).

In this chapter, we are going to cover the following:

- Python's characteristics and ecosystem
- Guidelines on how to get up and running with Python and virtual environments
- How to run Python programs
- How to organize Python code and its execution model

A proper introduction

We love to make references to the real world when we teach coding; we believe they help people to better retain the concepts. 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 are instructing a computer about the things it has to do. Where does the action happen? In many places: the computer memory, hard drives, network cables, the 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 being written. A person can be an object. A car is an object. A pair of trousers is an object. Luckily, Python understands objects very well.

The two main features any object has are **properties** and **methods**. Let's take the example of a person as an object. Typically, in a computer program, you'll represent people as customers or employees. The properties that you store against them are things like a name, a Social Security number, an age, whether they have a driving license, an email, gender, and so on. In a computer program, you store all the data needed in order to use an object for the purpose that needs to be served. If you are coding a website to sell clothes, you probably want to store the heights and weights as well as other measures of your customers so that the appropriate clothes can be suggested to 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 color (*black*) property of a pen to identify it (most likely it was being kept alongside different colored pens for the distinction to be necessary).

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 represent me.

So, now that you know what objects are, that they expose methods that can be 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 (`https://docs.python.org/3/reference/datamodel.html`):

> *"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, 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 ID 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 also never changes. The type states 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 be changed or not: if it can, the object is said to be **mutable**; otherwise, it is said to be **immutable**.

How, then, 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), and collections, 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.

To do all this, we need a language. That's what Python is for. Python is the language we will 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 creation 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.

We started programming when we were both very young. Fabrizio started at the age of 7, on a Commodore VIC-20, which was later replaced by its bigger brother, the Commodore 64. The language it used was **BASIC**. Heinrich started when he learned Pascal in high school. Between us, we've programmed in Pascal, Assembly, C, C++, Java, JavaScript, Visual Basic, PHP, ASP, ASP .NET, C#, and plenty of others we can't even remember; only when we landed on Python did we finally have that feeling that you have when you find the right couch in the shop. When all of your body is yelling: *Buy this one! This one is perfect!*

It took us about a day to get used to it. Its syntax is a bit different from what we were used to, but after getting past that initial feeling of discomfort (like having new shoes), we both 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 (we recommend you read the Python page on Wikipedia to get a more detailed introduction).

In our opinion, Python epitomizes 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 specific **operating system** (**OS**) 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 aren't that experienced as a programmer, but this is a major feature. It means less cluttering in your head, as well as less skimming through the documentation, and less need for mappings 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 being able to respond more quickly to 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), maintain, debug, and refactor.

Another important aspect is that Python runs without the need for lengthy and time-consuming compilation and linkage steps, so there is no need to wait to see the results of your work.

An extensive library

Python has an incredibly extensive standard library (it is said to come with *batteries included*). If that wasn't enough, the Python international community 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 realize that a certain feature is required, in most cases, there is at least one library where that feature has already been implemented.

Software quality

Python is heavily focused on readability, coherence, and quality. The language's uniformity allows for high readability, and this is crucial nowadays, as coding is more of a collective effort than a solo endeavor. Another important aspect of Python is its intrinsic multiparadigm nature. You can use it as a scripting language, but you can also exploit object-oriented, imperative, and functional programming styles—it is extremely 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 gluing agent between complex applications that need to talk to each other in some way. This is more of an advanced topic, but in the real world, this feature is important.

Satisfaction and enjoyment

Last, but by no means least, there is the fun of it! Working with Python is fun; we can code for eight hours and leave the office happy and satisfied, unaffected by 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 of why we would recommend Python to everyone. Of course, there are many other technical and advanced features that we could have mentioned, but they don't really pertain to an introductory section like this one. They will come up naturally, chapter after chapter, as we learn about Python in greater detail.

What are the drawbacks?

Probably, the only drawback that one could find in Python, which is not due to personal preferences, is its execution speed. Typically, Python is slower than its compiled siblings. 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 increased runtimes due to the fact that Python is not compiled down to the machine level, as other languages are.

Despite this, Python speed is rarely a problem today, hence its wide use regardless of this aspect. 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. Moreover, many programs spend a great proportion of the time waiting for I/O operations to complete; therefore, the raw execution speed is often a secondary factor to the overall performance.

In situations where speed really is crucial, one can switch to faster Python implementations, such as **PyPy**, which provides, on average, just over a four-fold speedup by implementing advanced compilation techniques (check `https://pypy.org/` for reference). It is also possible to write performance-critical parts of your code in faster languages, such as C or C++, and integrate that with your Python code. Libraries such as **pandas** and **NumPy** (which are commonly used for doing data science in Python) use such techniques.

> There are a number of different implementations of the Python language. In this book, we will use the reference implementation, known as CPython. You can find a list of other implementations at: `https://www.python.org/download/alternatives/`

If that isn't convincing enough, you can always consider that Python has been used to drive the backend of services such as Spotify and Instagram, where performance is a concern. From this, it can be seen that Python has does its job perfectly well.

Who is using Python today?

Still not convinced? Let's take a very brief look at the companies using Python today: Google, YouTube, Dropbox, Zope Corporation, Industrial Light & Magic, Walt Disney Feature Animation, Blender 3D, Pixar, NASA, the NSA, Red Hat, Nokia, IBM, Netflix, Yelp, Intel, Cisco, HP, Qualcomm, JPMorgan Chase, and Spotify—to name just a few. Even games such as Battlefield 2, Civilization IV, and The Sims 4 are implemented using Python.

Python is used in many different contexts, such as system programming, web and API programming, GUI applications, gaming and robotics, rapid prototyping, system integration, data science, database applications, real-time communication, and much more. Several prestigious universities have also adopted Python as their main language in computer science courses.

Setting up the environment

Before talking about the installation of Python on your system, let us tell you about the Python version you will be using in this book.

Python 2 versus Python 3

Python comes in two main versions: Python 2, which is the older version, and Python 3, which is the most recent rendition. The two versions, though similar, are incompatible in some respects.

In the real world, Python 2 is now only running legacy software. Python 3 has been out since 2008, and the lengthy transition phase from Version 2 has mostly come to an end. Python 2 was widely used in the industry, and it took a long time and sometimes a huge effort to make the transition. Some Python 2 software will never be updated to Python 3, simply because the cost and effort involved is not considered worth it. Some companies, therefore, prefer to keep their old legacy systems running just as they are, rather than updating them just for the sake of it.

At the time of writing, Python 2 has been deprecated and all of the most widely used libraries have been ported to Python 3. It is strongly recommended to start new projects in Python 3.

During the transition phase, many libraries were rewritten to be compatible with both versions, mostly harnessing the power of the *six* library (the name comes from the multiplication 2 x 3, due to the porting from Version 2 to 3), which helps you to introspect and adapt the behavior according to the version used. Now that Python 2 has reached its **end of life** (**EOL**), some libraries have started to reverse that trend and are dropping support for Python 2.

> According to PEP 373 (`https://legacy.python.org/dev/peps/pep-0373/`), the EOL of Python 2.7 was set to 2020. The last version is 2.7.18; there will not be a Python 2.8.

On Fabrizio's machine (MacBook Pro), this is the latest Python version:

```
>>> import sys
>>> print(sys.version)
3.9.2 (default, Mar  1 2021, 23:29:21)
[Clang 12.0.0 (clang-1200.0.32.29)]
```

So, you can see that the version is 3.9.2, which was out on the 1st of March 2021. The preceding text is a little bit of Python code that was typed into a console. We'll talk about this in a moment.

All the examples in this book will be run using Python 3.9. If you wish to follow the examples and download the source code for this book, please make sure you are using the same version.

Installing Python

We never really understood 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 instructions and the time you actually try them out, months have passed—if you're lucky. One version change, and things may not work in the way they are described in the book. Luckily, we have the web now, so in order to help you get up and running, we will just give you pointers and objectives.

We are conscious that the majority of readers would probably have preferred to have guidelines in the book. We doubt it would have made their life easier, as we believe that if you want to get started with Python you have to put in that initial effort in order to get familiar with the ecosystem. It is very important, and it will boost your confidence to face the material in the chapters ahead. If you get stuck, remember that Google is your friend—when it comes to setting up, everything related to this can be found online.

Setting up the Python interpreter

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

Getting Python and the libraries you need up and running requires a bit of handiwork. Linux and macOS seem to be the most user-friendly for Python programmers; Windows, on the other hand, may require a bit more effort.

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 resource on Python and its ecosystem is
> `https://docs.python-guide.org`. You can find instructions
> there to set up Python on different operating systems, using
> different methods.

Find the appropriate "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, we would suggest making a complete installation, just to be safe, of all the components the installer holds). If you need more guidance on how to install Python on Windows, please check out this page on the official documentation: `https://docs.python.org/3/using/windows.html`.

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

> Please note that we usually refer to the Python interactive shell
> simply as the **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 in this book, please make sure you are running the console with administrator rights.

On macOS, start a Terminal by going to **Applications > Utilities > Terminal**.

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

We will use the term **console** interchangeably to indicate the Linux console, the Windows Command Prompt, and the Macintosh Terminal. We will also indicate the command-line prompt with the Linux default format, like this:

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

If you're not familiar with that, please take some time to learn the basics of how a console works. In a nutshell, after the $ sign, you will normally find an instruction that you have to type. Pay attention to capitalization and spaces, as they are very important.

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 roughly what you should see when you run Python (it will change in some details according to the version and OS):

```
fab $ python3
Python 3.9.2 (default, Mar  1 2021, 23:29:21)
[Clang 12.0.0 (clang-1200.0.32.29)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

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

About virtual environments

When working with Python, it is very common to use virtual environments. Let's see what they are and why we need them 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 explore in *Chapter 14, Introduction to API Development*. Let's say the Django version you install for Project X is 2.2.

Now, your website is so good that you get another client, Y. She 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 3.0 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. But there's a solution: virtual environments!

Virtual environments are 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 2.2, while Project Y will hold Django 3.0.

> 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. So, take this as a rule, such as brushing your teeth before going to bed: always create a virtual environment when you start a new project.

When it comes to creating a virtual environment on your system, there are a few different methods to carry this out. As of Python 3.5, the suggested way to create a virtual environment is to use the `venv` module. You can look it up on the official documentation page (`https://docs.python.org/3/library/venv.html`) for further information.

> If you're using a Debian-based distribution of Linux, for example, you will need to install the `venv` module before you can use it:
>
> ```
> $ sudo apt-get install python3.9-venv
> ```

Another common way of creating virtual environments is to use the `virtualenv` third-party Python package. You can find it on its official website: `https://virtualenv.pypa.io`.

In this book, we will use the recommended technique, which leverages the `venv` module from the Python standard library.

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, when you want to work with it, is to activate it. Activating virtual environments basically produces some path juggling behind the scenes so that when you call the Python interpreter from that shell, you're actually calling the active virtual environment one, instead of the system one. We will show you a full example on both Windows and Ubuntu (on a Mac, it's very similar to Ubuntu). We will:

1. Open a terminal and change into the folder (directory) we use as root for our projects (our folder is `srv`). We are going to create a new folder called `my-project` and change into it.

2. Create a virtual environment called `lpp3ed`.

3. After creating the virtual environment, we will activate it. The methods are slightly different between Linux, macOS, and Windows.

4. Then, we will make sure that we are running the desired Python version (3.9.X) by running the Python interactive shell.

5. Deactivate the virtual environment.

> Some developers prefer to call all virtual environments the same name (for example, `.venv`). This way they can configure tools and run scripts against any virtual environment by just knowing the name of the project they are located in. The dot in `.venv` is there because in Linux/macOS, prepending a name with a dot makes that file or folder invisible.

These steps are all you need to start a project.

We are going to start with an example on Windows (note that you might get a slightly different result, according to your OS, Python version, and so on). In this listing, lines that start with a hash, #, are comments, spaces have been introduced for readability, and an arrow, →, indicates where the line has wrapped around due to lack of space:

```
C:\Users\Fab\srv>mkdir my-project  # step 1
C:\Users\Fab\srv>cd my-project
```

```
C:\Users\Fab\srv\my-project>where python  # check system python
C:\Users\Fab\AppData\Local\Programs\Python\Python39\python.exe
C:\Users\Fab\AppData\Local\Microsoft\WindowsApps\python.exe

C:\Users\Fab\srv\my-project>python -m venv lpp3ed  # step 2

C:\Users\Fab\srv\my-project>lpp3ed\Scripts\activate  # step 3

# check python again, now virtual env python is listed first
(lpp3ed) C:\Users\Fab\srv\my-project>where python
C:\Users\Fab\srv\my-project\lpp3ed\Scripts\python.exe
C:\Users\Fab\AppData\Local\Programs\Python\Python39\python.exe
C:\Users\Fab\AppData\Local\Microsoft\WindowsApps\python.exe

(lpp3ed) C:\Users\Fab\srv\my-project>python  # step 4
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55)
→ [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()

(lpp3ed) C:\Users\Fab\srv\my-project>deactivate  # step 5
C:\Users\Fab\srv\my-project>
```

Each step has been marked with a comment, so you should be able to follow along quite easily.

On a Linux machine, the steps are the same, but the commands are structured slightly differently. Moreover, you might have to execute some additional setup steps to be able to use the venv module to create a virtual environment. It is impossible to give instructions for all the Linux distributions out there, so please have a look online to find what is appropriate for your distribution.

Once you are set up, these are the instructions necessary to create a virtual environment:

```
fab@fvm:~/srv$ mkdir my-project  # step 1
fab@fvm:~/srv$ cd my-project

fab@fvm:~/srv/my-project$ which python3.9  # check system python
/usr/bin/python3.9  # <-- system python3.9

fab@fvm:~/srv/my-project$ python3.9 -m venv lpp3ed  # step 2
```

```
fab@fvm:~/srv/my-project$ source ./lpp3ed/bin/activate  # step 3

# check python again: now using the virtual environment's one
(lpp3ed) fab@fvm:~/srv/my-project$ which python
/home/fab/srv/my-project/lpp3ed/bin/python

(lpp3ed) fab@fvm:~/srv/my-project$ python  # step 4
Python 3.9.2 (default, Feb 20 2021, 20:56:08)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()

(lpp3ed) fab@fvm:~/srv/my-project$ deactivate  # step 5
fab@fvm:~/srv/my-project$
```

Something to notice here is that in order to activate the virtual environment, we need to run the `lpp3ed/bin/activate` script, which needs to be sourced. When a script is *sourced*, it means that it is executed in the current shell, and therefore its effects last after the execution. 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 it).

At this point, you should be able to create and activate a virtual environment. Please try and create another one without us guiding you. Get acquainted with this procedure—it is something that you will always be doing: *we never work system-wide with Python*, remember? Virtual environments are extremely important.

The source code for the book contains a dedicated folder for each chapter. When the code shown in the chapter requires third-party libraries to be installed, we will include a `requirements.txt` file (or an equivalent `requirements` folder with more than one text file inside) that you can use to install the libraries required to run that code. We suggest that when experimenting with the code for a chapter, you create a dedicated virtual environment for that chapter. This way, you will be able to get some practice in the creation of virtual environments, and the installation of third-party libraries.

Installing third-party libraries

> In order to install third-party libraries, we need to use the Python Package Installer, known as **pip**. Chances are that it is already available to you within your virtual environment, but if not, you can learn all about it on its documentation page: `https://pip.pypa.io`.

The following example shows how to create a virtual environment and install a couple of third-party libraries taken from a requirements file.

```
mpro:srv fab$ mkdir my-project
mpro:srv fab$ cd my-project/

mpro:my-project fab$ python3.9 -m venv lpp3ed
mpro:my-project fab$ source ./lpp3ed/bin/activate

(lpp3ed) mpro:my-project fab$ cat requirements.txt
Django==3.1.7
requests==2.25.1

# the following instruction shows how to use pip to install
# requirements from a file
(lpp3ed) mpro:my-project fab$ pip install -r requirements.txt
Collecting Django==3.1.7
  Using cached Django-3.1.7-py3-none-any.whl (7.8 MB)

... much more collection here ...

Collecting requests==2.25.1
  Using cached requests-2.25.1-py2.py3-none-any.whl (61 kB)

Installing collected packages: ..., Django, requests, ...
Successfully installed Django-3.1.7 ... requests-2.25.1 ...

(lpp3ed) mpro:my-project fab$
```

As can be seen at the bottom of the listing, pip has installed both libraries that are in the requirements file, plus a few more. This happened because both `django` and `requests` have their own list of third-party libraries that they depend upon, and therefore pip will install them automatically for us.

Now, with the scaffolding out of the way, we are ready to talk a bit more about Python and how it can be used. Before we do that though, allow us to say a few words about the console.

Your friend, the console

In this, the 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 on, you're losing time. Getting things done with the console, counter-intuitive though it may at first seem, results in higher productivity and speed. Believe us, we know—you will have to trust us on this.

Speed and productivity are important, and even though we have nothing against the mouse, being fluent with the console is very good for another reason: when you develop code that ends up on some server, the console might be the only available tool to access the code on that server. If you make friends with it, you will never get lost when it is of utmost importance that you don't (typically, when the website is down and you have to investigate very quickly what has happened).

If you're still not convinced, please us the benefit of the doubt and give it a try. It's easier than you think, and you won't 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 to 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 one; or you can use a script to work with files and folders; you can create or modify configuration files—technically, there is not much that cannot be done in a script.

It is rather 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 (or a similar technology) to schedule jobs (commands or shell scripts) to run periodically at fixed times, dates, or intervals.

We have Python scripts to do all the menial tasks that would take us minutes or more to do manually, and at some point, we decided to automate. We'll devote half of *Chapter 12*, *GUIs and Scripting*, to 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 up 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 few lines that should look something like this:

```
(lpp3ed) mpro:my-project fab$ python
Python 3.9.2 (default, Mar  1 2021, 23:29:21)
[Clang 12.0.0 (clang-1200.0.32.29)] on darwin
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 will 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 math:

```
>>> 3 + 7
10
>>> 10 / 4
2.5
>>> 2 ** 1024
179769313486231590772930519078902473361797697894230657273430081157
732675805500963132708477322407536021120113879871393357658789768814
416622492847430639474124377767893424865485276302219601246094119453
082952085005768838150682342462881473913110540827237163350510684586
298239947245938479716304835356329624224137216
```

The last operation is showing you something incredible. We raise 2 to the power of 1024, and Python handles 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.

We 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 much more. You will find that the interactive shell soon becomes one of your dearest friends on this journey you are embarking on.

Another solution, which comes in a much nicer graphic layout, is to use the **Integrated Development and Learning Environment** (**IDLE**). It's quite a simple **Integrated Development Environment** (**IDE**), which is intended mostly for beginners. It has a slightly larger set of capabilities than the bare 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 on any other system. You can find more 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* was chosen in honor of Eric 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 an application. We'll see many examples throughout this book of this mode. We will understand more about it in a moment, when we talk about how Python code is organized and run.

Running Python as a GUI application

Python can also be run as a **Graphical User Interface** (**GUI**). There are several frameworks available, some of which are cross-platform, and some others that are platform-specific. In *Chapter 12*, *GUIs and Scripting*, 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 GUI toolkit that takes desktop application development to a higher level than the conventional approach. It is the standard GUI for **Tool Command Language** (**Tcl**), but also for many other dynamic languages, and it can produce rich native applications that run seamlessly under Windows, Linux, macOS, and more.

Tkinter comes bundled with Python; therefore, it gives the programmer easy access to the GUI world, and for these reasons, we have chosen it to be the framework for the GUI examples that are presented in this book.

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

- PyQt5/PySide 2
- wxPython
- Kivy

Describing them in detail is outside the scope of this book, but you can find all the information you need on the Python website:

```
https://docs.python.org/3/faq/gui.html
```

Information can be found in the *What platform-independent GUI toolkits exist for Python?* section. If GUIs are what you're looking for, remember to choose the one you want according to some basic 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 section, we will start to enter the proverbial rabbit hole and introduce 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 are on Windows or macOS, which typically hide file extensions from the user, we recommend that you change the configuration so that you can see the complete names of the files. This is not strictly a requirement, only a suggestion that may come in handy when discerning files from each other.

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 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, which is 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 a **package**, which allows you to group modules together. A package is nothing more than a folder that must contain a special file, `__init__.py`. This does not need to hold any code, but its presence is required to tell Python that this is not just a typical folder—it is actually a package.

As always, an example will make all of this much clearer. We have created an example structure in our book project, and when we type in the console:

```
$ tree -v example
```

We get a tree representation of the contents of the `ch1/example` folder, which holds the code for the examples of this chapter. Here's what the structure of a really 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, we expect to find various utility tools, and in fact, we can guess that the modules there are named based on the types 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 simple folder.

Had this software been organized within modules only, it would have been harder to infer its structure. We placed 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 could not organize the code into packages and modules.

How do we use modules and packages?

When a developer is writing an application, it is 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 likely that it will be needed for more than one field.

For example, in a poll application, where the user is asked many questions, it's likely that several of them will require a numeric answer. These might be:

- 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, said more formerly, duplicate) the validation logic in every place where we expect a numeric answer. This would violate the **don't repeat yourself (DRY)** principle, which states that you should never repeat the same piece of code more than once in your application. In spite of the DRY principle, we feel the need here to stress the importance of this principle: *you should never repeat the same piece of code more than once in your application*!

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 copy.

- You may want to amend the way you carry out the validation, and again, you would have to change it in every copy.

- You may forget to fix or amend a piece of logic because you missed it when searching for all its occurrences. This would leave wrong or 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 the coding best practices. For this particular example, we need to be able to reuse a piece of code. To do this effectively, 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 a **function**.

We are not going too deep into the specifics here, so please just remember that a function is a block of organized, reusable code that 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. Details will be seen once 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 are writing a super-simple script, functions will be used all the time. Functions will be explored in *Chapter 4*, *Functions, the Building Blocks of Code*.

Python comes with a very extensive library, as mentioned a few pages ago. Now is a good time to define what a **library** is: a collection of functions and objects that provide functionalities to enrich the abilities of a language. For example, within Python's `math` library, a plethora of functions can be found, one of which is the `factorial` function, which 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:
>
> $$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$
>
> 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 are not clear right now; please just concentrate on the import part. We use a library by importing what we need from it, which will then be used specifically. In Python, to calculate 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 in the console for us (in this case, the result of the function call: 120).

Let's go back to our example, the one with `core.py`, `run.py`, `util`, and so on. Here, the package `util` is our utility library. This is 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 have 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 section, we would like to introduce you to some important concepts, such as scope, names, and namespaces. You can read all about Python's execution model in the official language reference (`https://docs.python.org/3/reference/executionmodel.html`), of course, but we would argue that it is quite technical and abstract, so let us 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 to obtain this. 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 are looking for. When software is structured correctly, it also promotes code reuse. Furthermore, disorganized software is more likely to contain scattered pieces of duplicated logic.

As a first example, let us take a book. We refer to a book by its title; 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 see a quick example (again, notice that anything that follows a # is a comment):

```
>>> n = 3  # integer number
>>> address = "221b Baker Street, NW1 6XE, London"  # Sherlock Holmes'
address
>>> employee = {
...     'age': 45,
...     'role': 'CTO',
...     'SSN': 'AB1234567',
... }
>>> # let's print them
>>> n
3
>>> address
'221b Baker Street, NW1 6XE, London'
>>> employee
{'age': 45, 'role': 'CTO', 'SSN': 'AB1234567'}
>>> other_name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'other_name' is not defined
>>>
```

Remember that each Python object has an identity, a type, and a value. We defined three objects in the preceding code; let's now examine their types and values:

- 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 object with three key/value pairs)

Fear not, we know we haven't covered what a dictionary is. We'll see, in *Chapter 2, Built-In Data Types*, that it is the king of Python data structures.

> Have you noticed that the prompt changed from `>>>` to `...`
> when we 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**, and these can be used
to retrieve data from 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 a mapping from names to objects. Examples are the set of built-in
names (containing functions that are always accessible 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 the 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 dealing with real code examples.
For now, just keep in mind that namespaces are places where names are associated
with objects.

There is another concept, closely related to that of a namespace, which we would like
to mention briefly: **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 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. There are four different scopes that Python makes accessible (not necessarily all of them are 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 an 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 in the enclosing scope, and this continues until the built-in scope is searched. If a name has still not been found after searching the built-in scope, then Python raises a `NameError` **exception**, which basically means that the name hasn't been defined (seen 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 demonstrate local and enclosing namespaces, we will have to define a few functions. Don't worry if you are not familiar with their syntax for the moment—that will come in *Chapter 4, Functions, the Building Blocks of Code*. Just remember that in the following code, when you see `def`, it means we are defining a function:

```python
# scopes1.py
# Local versus Global
# we define a function, called local
def local():
    m = 7
    print(m)

# we define m within the global scope
m = 5

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

print(m)
```

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 `local` function). When we execute this program with the following command (have you activated your virtual environment?):

```
$ python scopes1.py
```

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

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 will do two things: it will set up a name to an object representing number 7 and will print 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. On the next line, there is a call to the function `local`. At this point, Python executes the function, so at this time, the binding `m = 7` happens in the local scope and is printed. Finally, there is a call to the `print` function, which is executed and will now print 5.

One very important thing to note is that the part of the code that belongs to the definition of the `local` function 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.

> In other languages, such as Java, C#, and C++, scopes are created by writing code within a pair of curly braces: { ... }. Therefore, in Python, indenting code corresponds to opening a curly brace, while outdenting code corresponds to closing a curly brace.

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 the number 5 printed twice on the console. Let's see what the code would look like in this case:

```python
# 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:

```
$ 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` is not 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:

```python
# 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:

```
$ 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 become more clear as we go through the examples in the book. The *Classes* section of the Python tutorial (`https://docs.python.org/3/tutorial/classes.html`) has an interesting paragraph about scopes and namespaces. Be sure you read it to gain a deeper understanding of the subject.

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

Objects and classes

When we introduced objects previously in the *A proper introduction* section of the chapter, we 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 into too much detail (we'll do that in *Chapter 6*, *OOP, Decorators, and Iterators*), we want to give you a brief explanation about classes and objects.

We have 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 (which is unique), a type, and a value.

But how do they come to life? How do we create them? How do 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**. 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 that 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 of the *concept* of bike: an abstract set of features and characteristics that together form something called a *bike*.

In computer programming, that is called a *class*. It's that simple. Classes are used to create objects. In other words, we all know what a bike is; we know the class. But then your friend has her own bike, which is an *instance* of the bike class. Her 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 create two bikes, one red and one blue. We'll keep the code very simple, but don't fret if everything is not clear; all you need to care about at this moment is to understand the difference between a class and an object (or instance of a class):

```python
# 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!
```

> We hope by this point that we do not need to tell you to run the file every time, right? The filename is indicated in the first line of each code block. To execute the code inside a Python module, just run `$ python filename.py`.
>
> Remember to have your virtual environment activated!

So many interesting things to notice here. First, the definition of a class happens with the `class` statement. 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 `print("Braking!")`.

After having defined the class, we are ready to create some 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 underscores, in Python, is called a **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. It contains only a `print` statement, of course—it's just an example.

So, two bikes were created: one has a red color and carbon fiber frame, and the other one has a blue color and a steel frame. We pass those values upon creation; afterwards, 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 `red_bike`.

One last thing to notice: remember how we said that the set of attributes of an object is considered to be a namespace? We hope it's clearer now what that 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 for writing good code

Writing good code is not as easy as it seems. As we have already said, good code exposes a long list of qualities that are difficult to combine. 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 that will make your code instantly better: **PEP 8**.

> A **Python Enhancement Proposal** (**PEP**) is a document that describes a newly proposed Python feature. PEPs are also used to document processes around Python language development and to provide guidelines and information more generally. You can find an index of all PEPs at `https://www.python.org/dev/peps`.

PEP 8 is perhaps the most famous of all PEPs. It lays out a simple but effective set of guidelines to define Python aesthetics so that we write beautiful Python code. If you take just one suggestion out of this chapter, please let it be this: use PEP 8. Embrace it. You will thank us later.

Coding today is no longer a check-in/check-out business. Rather, it's more of a social effort. Several developers collaborate on a piece of code through tools such as Git and Mercurial, and the result is code that is produced by many different hands.

> Git and Mercurial are the distributed revision control systems that are most commonly used 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 by PEP 8, it's not uncommon for any of them landing on a piece of code to think they wrote it themselves (it actually happens to Fabrizio all the time, because he forgets the code he writes).

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 PEP 8, we can avoid this. We are such fans of it that we won't sign off a code review if the code doesn't respect it. So, please take the time to study it; this is very important.

> Nowadays Python developers can leverage several different tools to automatically format their code, according to PEP 8 guidelines. One such tool is called *black*, which has become very popular in recent years. There are also other tools, called linters, which check if the code is formatted correctly, and issue warnings to the developer with instructions on how to fix errors. One very famous linter is *flake8*. We encourage you to use these tools, as they simplify the task of coding well-formatted software.

In the examples in this book, we will try to respect it as much as we can. Unfortunately, we don't have the luxury of 79 characters (which is the maximum line length suggested by PEP 8), and we will have to cut down on blank lines and other things, but we promise you we'll try to lay out our code so that it's as readable as possible.

Python culture

Python has been adopted widely in all coding industries. It is used by many different companies for different purposes, while also being an excellent education tool (it is excellent for that purpose due to its simplicity, making it easy to learn; it encourages good habits for writing readable code; it is platform-agnostic; and it supports modern object-oriented programming paradigms).

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 some of its most adopted web frameworks, such as Django.

Python's source 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 as easy to use (it can feel claustrophobic when one has to code in a language that is not Python, at times).

Anyway, over the years, this concept of being Pythonic has emerged and, the way we understand it, 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 being Pythonic, we 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 instruction:

```
>>> 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 IDEs

Just a few words about IDEs… To follow the examples in this book, you don't need one; any decent text editor will do fine. If you want to have more advanced features, such as syntax coloring and auto-completion, you will have to get yourself an IDE. You can find a comprehensive list of open-source IDEs (just Google "Python IDEs") on the Python website.

Fabrizio uses Visual Studio Code, from Microsoft. It's free to use and it provides an immense multitude of features, which one can expand by installing extensions.

After working for many years with several editors, including Sublime Text, this was the one that felt most productive to him.

Heinrich, on the other hand, is a hardcore Vim user. Although it might have a steep learning curve, Vim is a very powerful text editor that can also be extended with plugins. It also has the benefit of being installed in almost every system a software developer has to work on.

Two important pieces of advice:

- Whatever IDE you decide to use, try to learn it well so that you can exploit its strengths, but *don't depend on it too much*. Practice working 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 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, only touching upon 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 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 namespaces, scopes, classes, and objects. 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, such as `pip`, 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 us. All you need is enthusiasm, an activated virtual environment, this book, your fingers, and probably some coffee.

Try to follow the examples; we'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, in 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 to, the movies you stream, the PDFs you open. Even the source of the chapter you're reading at this very moment is just a file, which is data.

Data can be simple, whether it is 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. This is data that describes the design of other data structures, or data that describes application data or its context. In Python, *objects are our 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.

In this chapter, we are going to cover the following:

- Python objects' structures
- Mutability and immutability
- Built-in data types: numbers, strings, dates and times, sequences, collections, and mapping types
- The `collections` module
- Enumerations

Everything is an object

Before we delve into the specifics, we want you to be very clear about objects in Python, so let's talk a little bit more about them. 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*. This is exactly the same concept. Here is a screenshot of what it may look like (you may have to tweak the settings to get to the same view):

Figure 2.1: A name pointing to an object

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 using an example, so we'll come back to this later.

Mutable or immutable? That is the question

The first fundamental distinction that Python makes on data is about whether or not the value of an object can change. If the value can change, the object is called **mutable**, whereas 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; take this example:

```
>>> age = 42
>>> age
42
>>> age = 43   #A
>>> age
43
```

In the preceding code, on line `#A`, have we changed the value of `age`? Well, no. But now it's 43 (we hear what you are saying...). 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, in fact, we did not change that 42 to 43—we actually just pointed `age` to a different location, which is the new *int* object whose value is 43. Let's see the same code also printing the IDs:

```
>>> age = 42
>>> id(age)
4377553168
>>> age = 43
>>> id(age)
4377553200
```

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.

> If you reproduce these examples on your computer, you will notice that the IDs you get will be different. This is of course expected, as they are generated randomly by Python, and will be different every time.

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` (don't worry about the class declaration for now—it is there only for completeness):

```
>>> class Person:
...     def __init__(self, age):
...         self.age = age
...
>>> fab = Person(age=42)
>>> fab.age
42
>>> id(fab)
4380878496
>>> id(fab.age)
4377553168
>>> fab.age = 25  # I wish!
>>> id(fab)  # will be the same
4380878496
>>> id(fab.age)  # will be different
4377552624
```

In this case, we set up an object `fab` whose `type` is `Person` (a custom class). On creation, the object is given the age of 42. We then print it, along with the object ID, and the ID of `age` as well. Notice that, even after we change `age` to be 25, 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, as it's very important. We'll remind you about it throughout 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 an 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 is—as long as it can fit in your computer's memory, Python will take care of it.

Integer numbers can be positive, negative, or 0 (zero). They support all the basic mathematical operations, as shown in the following example:

```
>>> a = 14
>>> b = 3
>>> a + b  # addition
17
>>> a - b  # subtraction
11
>>> a * b  # multiplication
42
>>> a / b  # true division
4.666666666666667
>>> a // b  # integer division
4
>>> a % b  # modulo operation (reminder of division)
2
>>> a ** b  # power operation
2744
```

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 another one, the so-called **integer division** (//), which returns the *floored* quotient of the operands.

> It might be worth noting that in Python 2 the division operator / behaves differently than in Python 3.

Let's see how division behaves differently when we introduce negative numbers:

```
>>> 7 / 4  # true division
1.75
>>> 7 // 4  # integer division, truncation 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. Integer division in Python is *always rounded toward minus infinity*. If, instead of flooring, you want to truncate a number to an integer, you can use the built-in `int()` function, as shown in the following example:

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

Notice that the truncation is done toward *0*.

> The `int()` function can also return integer numbers from string representation in a given base:
>
> ```
> >>> int('10110', base=2)
> ```
> 22

It's worth noting that the power operator, ******, also has a built-in function counterpart, `pow()`, shown in the example below:

```
>>> pow(10, 3)
1000.0  # result is float
>>> 10 ** 3
1000  # result is int
>>> pow(10, -3)
0.001
>>> 10 ** -3
0.001
```

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

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

The `pow()` function allows a third argument to perform **modular exponentiation**. The form with three arguments now accepts a negative exponent in the case where the base is relatively prime to the modulus.

The result is the **modular multiplicative inverse** of the base (or a suitable power of that, when the exponent is negative, but not -1), modulo the third argument. Here's an example:

```
>>> pow(123, 4)
228886641
>>> pow(123, 4, 100)
41  # notice: 228886641 % 100 == 41
>>> pow(37, -1, 43)  # modular inverse of 37 mod 43
7
>>> 7 * 37 % 43  # proof the above is correct
1
```

One nice feature introduced in Python 3.6 is the ability to add underscores within number literals (between digits or base specifiers, but not leading or trailing). The purpose is to help make some numbers more readable, such as `1_000_000_000`:

```
>>> n = 1_024
>>> n
1024
>>> hex_n = 0x_4_0_0  # 0x400 == 1024
>>> hex_n
1024
```

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, so `True` and `False` 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*, *Conditionals and Iteration*.

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 peek 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 the 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 this example, `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*, *OOP, Decorators, and Iterators*.

Real numbers

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.

Several programming languages give coders two different formats: single and double precision. The former takes up 32 bits of memory, 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
63.617251235400005
```

> In the calculation of the area, we wrapped the `radius ** 2` within parentheses. Even though that wasn't necessary because the power operator has higher precedence than the multiplication one, we think the formula reads more easily like that. Moreover, should you get a slightly different result for the area, don't worry. It might depend on your OS, how Python was compiled, and so on. As long as the first few decimal digits are correct, you know it's a correct result.

The `sys.float_info` sequence holds information about how floating point numbers will behave on your system. This is an example of what you might see:

```
>>> 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 floating point numbers. This means we can represent at most 2^{64} (that is *18,446,744,073,709,551,616*) distinct numbers. Take a look at the `max` and `epsilon` values for the float numbers, and you will realize that 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 and try the following in your console:

```
>>> 0.3 - 0.1 * 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 are handling prices, or financial calculations, or any kind of data that need 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 moment.

Complex numbers

Python gives you **complex numbers** support out of the box. If you don't know what complex numbers are, 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 is perhaps unlikely that you will use them, unless you're coding something scientific. Nevertheless, let's see a small example:

```
>>> c = 3.14 + 2.73j
>>> c = complex(3.14, 2.73)  # same as above
>>> 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 simplified
>>> 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
>>> f.as_integer_ratio()
(5, 3)
```

The `as_integer_ratio()` method has also been added to integers and Booleans. This is helpful, as it allows you to use it without needing to worry about what type of number is being worked with.

Although `Fraction` objects can be very useful at times, it's not that common to spot them in commercial software. Instead, it is much more common to see decimal numbers being used in all those contexts where precision is everything; for example, in scientific and financial calculations.

> It's important to remember that arbitrary precision decimal numbers come at a price in terms of performance, of course. The amount of data to be stored for each number is greater than it is for `Fractions` or `floats`. The way they are handled also requires the Python interpreter to work harder behind the scenes. Another interesting thing to note 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')
>>> D('1.4').as_integer_ratio()  # 7/5 = 1.4 (isn't this cool?!)
(7, 5)
```

Notice that when we construct a `Decimal` number from a *float*, it takes on all the approximation issues a *float* may come with. On the other hand, when we create a `Decimal` from an integer or a string representation of a number, then the `Decimal` will have no approximation issues, and therefore no quirky behavior. When it comes to currency or situations in which precision is of utmost importance, use decimals.

This concludes our introduction to built-in numeric types. Let's now look at 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 when formatting, 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 storing textual data though, or sending it on the network, you will likely want to encode it, using an appropriate encoding for the medium you are using. The result of an encoding produces a **bytes** object, whose syntax and behavior is similar to that of strings. 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 multiple lines. An example will clarify this:

```
>>> # 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, and then explicitly, using the print() function. A good exercise would be to find out why they are different. Are you up to the challenge? (Hint: look up the str() and repr() functions.)

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

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

Python 3.9 has introduced two new methods that deal with the prefixes and suffixes of strings. Here's an example that explains the way they work:

```
>>> s = 'Hello There'
>>> s.removeprefix('Hell')
'o There'
>>> s.removesuffix('here')
'Hello T'
>>> s.removeprefix('Ooops')
'Hello There'
```

The nice thing about them is shown by the last instruction: when we attempt to remove a prefix or suffix which is not there, the method simply returns a copy of the original string. This means that these methods, behind the scenes, are checking if the prefix or suffix matches the argument of the call, and when that's the case, they remove it.

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 most widely used encoding for the web. Notice also that by adding the literal b in front of a string declaration, we're creating a *bytes* object:

```
>>> s = "This is üɲíc0de"  # 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 üɲíc0de'
```

```
>>> 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 access them at one precise position (**indexing**), or to get a sub-sequence out of them (**slicing**). When dealing with immutable sequences, both operations are read-only.

While indexing comes in one form—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, along with the *step*. They are separated with a colon (:) like this: my_sequence[start:stop:step]. All the arguments are optional; start is inclusive, and stop is exclusive. It's probably better to see an example, rather than try to explain them any further with 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.'
```

The last line is quite interesting. If you don't specify any of the parameters, Python will fill in the defaults for you. In this case, *start* will be the start of the string, *stop* will be the end of the string, and *step* will be the default: 1. This is an easy and quick way of obtaining a copy of the string s (the same value, but a different object). Can you think of a way to get the reversed copy of a string using slicing (don't look it up—find it for yourself)?

String formatting

One of the features strings have is the ability to be used as a template. There are several different ways of formatting a string, and for the full list of possibilities, we encourage you to look up the documentation. Here are some common examples:

```
>>> greet_old = 'Hello %s!'
>>> greet_old % 'Fabrizio'
'Hello Fabrizio!'
>>> greet_positional = 'Hello {}!'
>>> greet_positional.format('Fabrizio')
'Hello Fabrizio!'
>>> greet_positional = 'Hello {} {}!'
>>> greet_positional.format('Fabrizio', 'Romano')
'Hello Fabrizio Romano!'
>>> greet_positional_idx = 'This is {0}! {1} loves {0}!'
>>> greet_positional_idx.format('Python', 'Heinrich')
'This is Python! Heinrich loves Python!'
>>> greet_positional_idx.format('Coffee', 'Fab')
'This is Coffee! Fab loves Coffee!'
>>> keyword = 'Hello, my name is {name} {last_name}'
>>> keyword.format(name='Fabrizio', last_name='Romano')
'Hello, my name is Fabrizio Romano'
```

In the previous example, you can see four different ways of formatting strings. The first one, which relies on the `%` operator, is deprecated and shouldn't be used anymore. The current, modern way to format a string is by using the `format()` string method. You can see, from the different examples, that a pair of curly braces acts as a placeholder within the string. When we call `format()`, we feed it data that replaces the placeholders. We can specify indexes (and much more) within the curly braces, and even names, which implies we'll have to call `format()` using keyword arguments instead of positional ones.

Notice how `greet_positional_idx` is rendered differently by feeding different data to the call to `format`.

One last feature we want to show you was added to Python in version 3.6, and it's called **formatted string literals**. This feature is quite cool (and it is faster than using the `format()` method): strings are prefixed with `f`, and contain replacement fields surrounded by curly braces.

Replacement fields are expressions evaluated at runtime, and then formatted using the format protocol:

```
>>> name = 'Fab'
>>> age = 42
>>> f"Hello! My name is {name} and I'm {age}"
"Hello! My name is Fab and I'm 42"
>>> from math import pi
>>> f"No arguing with {pi}, it's irrational..."
"No arguing with 3.141592653589793, it's irrational..."
```

An interesting addition to f-strings, which was introduced in Python 3.8, is the ability to add an equals sign specifier within the f-string clause; this causes the expression to expand to the text of the expression, an equals sign, then the representation of the evaluated expression. This is great for self-documenting and debugging purposes. Here's an example that shows the difference in behavior:

```
>>> user = 'heinrich'
>>> password = 'super-secret'
>>> f"Log in with: {user} and {password}"
'Log in with: heinrich and super-secret'
>>> f"Log in with: {user=} and {password=}"
"Log in with: user='heinrich' and password='super-secret'"
```

Check out the official documentation to learn everything about string formatting and how truly powerful it can be.

Tuples

The last immutable sequence type we are going to look at here is the **tuple**. A tuple is a sequence of arbitrary Python objects. In a tuple declaration, items are separated by commas. Tuples are used everywhere in Python. They allow for patterns that are quite 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 objects (in several languages, it is common for a function to return only one object), and in the Python console, tuples can be used 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)  # braces are optional here
```

```
>>> 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 a comma after the item. The reason is that without the comma that item is wrapped in braces on its own, in what can be considered 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 first see the 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)
Now let's see how we would do it in Python:
>>> a, b = 0, 1
>>> a, b = b, a  # this is the Pythonic way to do it
>>> a, b
(1, 0)
```

Take a look at the line that shows you the Pythonic way of swapping two values. Do you remember what we wrote in *Chapter 1*, *A Gentle Introduction to Python*? 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 also means economy.

Because they are immutable, tuples can be used as keys for dictionaries (we'll see this shortly). To us, 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 a heterogeneous sequence of elements while, on the other hand, lists are, most of the time, homogeneous. Moreover, tuples are normally accessed via unpacking or indexing, while lists are usually iterated over.

Mutable sequences

Mutable sequences differ from their immutable counterparts in that they can be changed after creation. There are two mutable sequence types in Python: **lists** and **byte arrays**.

Lists

Python lists are very similar to tuples, but they don't have the restrictions of immutability. Lists are commonly used for storing collections of homogeneous objects, but there is nothing preventing you from storing 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, we showed you how to create a list using various techniques. We would like you to take a good look at the line with the comment *Python is magic*, which we don't expect you to fully understand at this point—especially if you are unfamiliar with Python. That is called a **list comprehension**: a very powerful functional feature of Python, which we will see in detail in *Chapter 5, Comprehensions and Generators*. We just wanted to spark your curiosity at this point.

Creating lists is good, but the real fun begins 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 `1s` 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 a list's main methods. We want to show you how powerful they are, using the method `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 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
>>> from math import prod
>>> prod(a)  # product of all values in the list
105
>>> 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]
```

Notice how easily we can perform the sum and the product of all values in a list. The function prod(), from the math module, is just one of the many new additions introduced in Python 3.8. Even if you don't plan to use it that often, it's always a good idea to check out the math module and be familiar with its functions, as they can be quite helpful.

The last two lines in the preceding code are also quite interesting, as they introduce us to a concept called **operator overloading**. In short, this 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 and see something a little more interesting. We want to show you how powerful the sorted 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 in this case). When we call `sorted(my_list)`, we get a sorted version of `my_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 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 (with `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 will see that they match.

For completeness, we included an example of sorting only on the elements at position 1, and then again, with the same sorting but in reverse order. If you have ever seen sorting in other languages, you should be quite impressed at this 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 score the same in the comparison, their original order is preserved. We've seen this in the result of `sorted(a, key=itemgetter(0))`, which yielded `[(1, 3), (1, 2), ...]`, in which the order of those two tuples had been preserved because they had the same value at position 0.

Bytearrays

To conclude our overview of mutable sequence types, let's spend a moment 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 in a bytearray are integers in the range [0, 256).

> When it comes to intervals, we are 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 bracket means that it is 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 (4, 5, and 6). Items in a bytearray type are integers between 0 and 256; 0 is included, 256 is not. One reason that 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 an example with the `bytearray` type:

```
>>> 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, 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 can prove to be very handy. On line #A, we created a `bytearray` named as `name` from the bytes literal `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.

> From the official documentation (`https://docs.python.org/3.9/glossary.html`): "An object is **hashable** if it has a hash value which never changes during its lifetime, and can be compared to other objects. [...] Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally. Most of Python's immutable built-in objects are hashable; mutable containers (such as lists or dictionaries) are not; immutable containers (such as tuples and frozensets) are only hashable if their elements are hashable. Objects which are instances of user-defined classes are hashable by default. They all compare unequal (except with themselves), and their hash value is derived from their id()."

Objects that compare equally must have the same hash value. Sets are very commonly used to test for membership; 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 an 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.

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

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 we added some duplication to emphasize that the resulting set won't have any. Let's see an example using 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)  # nor can we 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 with respect to 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 easily the most interesting. 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 also mutable objects. There are quite a few different ways to create a dictionary, so let us 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  # They are 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 five 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 (that is, that they have the same ID, not just the same value), but unless you have a good reason to use it, you should use the double equals instead. In the preceding code, we also used one nice function: `zip()`. It is named after the real-life zip, which glues together two parts, taking one element from each part at a time. Let us 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, we 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 we had to wrap the `list()` constructor around the `zip()` call (the reason is `zip()` returns an iterator, not a `list`, so if we want to see the result, we need to exhaust that iterator into something—a list in this case), 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?

Take a look at the zip of your suitcase, or a purse, or the cover of a pillow, and you will see it works exactly like the one in Python. 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 using square brackets. Do you remember strings, lists, and tuples? We were accessing elements at some position through square brackets as well, which is yet another example of Python's consistency.

Let's now look at 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.

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

```
>>> d = dict(zip('hello', range(5)))
>>> d
{'h': 0, 'e': 1, 'l': 3, 'o': 4}
```

```
>>> d.keys()
dict_keys(['h', 'e', 'l', 'o'])
>>> d.values()
dict_values([0, 1, 3, 4])
>>> d.items()
dict_items([('h', 0), ('e', 1), ('l', 3), ('o', 4)])
>>> 3 in d.values()
True
>>> ('o', 4) in d.items()
True
```

There are a few things to note here. First, notice how we are 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 the value 3), overwrites the first one (the one with the value 2). Another thing to notice is that when asking for any view, the original order in which items were added is now preserved, while before version 3.6 there was no guarantee of that.

As of Python 3.6, the `dict` type has been reimplemented to use a more compact representation. This resulted in dictionaries using 20% to 25% less memory when compared to Python 3.5. Moreover, since Python 3.6, as a side effect, dictionaries preserve the order in which keys were inserted. This feature has received such a welcome from the community that in 3.7 it has become an official feature of the language rather than an implementation side effect. Since Python 3.8, dictionaries are also reversible!

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
{'h': 0, 'e': 1, 'l': 3, 'o': 4}
>>> d.popitem()  # removes a random item (useful in algorithms)
('o', 4)
>>> d
{'h': 0, 'e': 1, '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
{'h': 0, 'e': 1, 'another': 'value', 'a': 13}
>>> 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 to return something else, but we'll see this when we explore functions. None is frequently used to represent the absence of a value, and it is quite commonly used as a default value for arguments in function declaration. Some inexperienced coders sometimes write code that returns either False or None. Both False and None evaluate to False in a Boolean context, so it may seem that there is not much difference between them. But actually, we would argue the contrary, that there is an important difference: False means that we have information, and the information we have is False. None means *no information*; no information is very different from information that 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 we really like about dictionaries is setdefault(). It behaves like get(), but also sets the key with the given value if it is not there. Let's see an 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}  # no override, as expected
```

This brings us to the end of this tour of dictionaries. Test your knowledge about them by trying to foresee what `d` looks like after this line:

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

Don't worry if you don't get it immediately. We just want to encourage you to experiment with dictionaries.

Python 3.9 sports a brand-new union operator available for `dict` objects, which was introduced by `PEP 584`. When it comes to applying union to `dict` objects, we need to remember that union for them is not commutative. This becomes evident when the two `dict` objects we're merging have one or more keys in common. Check out this example:

```
>>> d = {'a': 'A', 'b': 'B'}
>>> e = {'b': 8, 'c': 'C'}
>>> d | e
{'a': 'A', 'b': 8, 'c': 'C'}
>>> e | d
{'b': 'B', 'c': 'C', 'a': 'A'}
>>> {**d, **e}
{'a': 'A', 'b': 8, 'c': 'C'}
>>> {**e, **d}
{'b': 'B', 'c': 'C', 'a': 'A'}
>>> d |= e
>>> d
{'a': 'A', 'b': 8, 'c': 'C'}
```

Here, `dict` objects `d` and `e` have the key `'b'` in common. For the `dict` object, `d`, the value associated with `'b'` is `'B'`; whereas, for `dict` `e`, it's the number 8. This means that when we merge them with `e` on the righthand side of the union operator, `|`, the value in `e` overrides the one in `d`. The opposite happens, of course, when we swap the positions of those objects in relation to the union operator.

In this example, you can also see how the union can be performed by using the `**` operator to produce a **dictionary unpacking**. It's worth noting that union can also be performed as an augmented assignment operation (`d |= e`), which works in place. Please refer to PEP 584 for more information about this feature.

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

Data types

Python provides a variety of specialized data types, such as dates and times, container types, and enumerations. There is a whole section in the Python standard library titled *Data Types*, which deserves to be explored; it is filled with interesting and useful tools for each and every programmer's needs. You can find it here:

`https://docs.python.org/3/library/datatypes.html`

In this section, we are briefly going to take a look at dates and times, collections, and enumerations.

Dates and times

The Python standard library provides several data types that can be used to deal with dates and times. This realm may seem innocuous at first glance, but it's actually quite tricky: timezones, daylight saving time… There are a huge number of ways to format date and time information; calendar quirks, parsing, and localizing—these are just a few of the many difficulties we face when we deal with dates and times, and that's probably the reason why, in this particular context, it is very common for professional Python programmers to also rely on various third-party libraries that provide some much-needed extra power.

The standard library

We will start with the standard library, and finish the session with a little overview of what's out there in terms of the third-party libraries you can use.

From the standard library, the main modules that are used to handle dates and times are `datetime`, `calendar`, `zoneinfo`, and `time`. Let's start with the imports you'll need for this whole section:

```
>>> from datetime import date, datetime, timedelta, timezone
>>> import time
>>> import calendar as cal
>>> from zoneinfo import ZoneInfo
```

The first example deals with dates. Let's see how they look:

```
>>> today = date.today()
>>> today
datetime.date(2021, 3, 28)
>>> today.ctime()
'Sun Mar 28 00:00:00 2021'
>>> today.isoformat()
```

```
'2021-03-28'
>>> today.weekday()
6
>>> cal.day_name[today.weekday()]
'Sunday'
>>> today.day, today.month, today.year
(28, 3, 2021)
>>> today.timetuple()
time.struct_time(
    tm_year=2021, tm_mon=3, tm_mday=28,
    tm_hour=0, tm_min=0, tm_sec=0,
    tm_wday=6, tm_yday=87, tm_isdst=-1
)
```

We start by fetching the date for today. We can see that it's an instance of the
`datetime.date` class. Then we get two different representations for it, following the *C*
and the *ISO 8601* format standards, respectively. After that, we ask what day of the
week it is, and we get the number 6. Days are numbered 0 to 6 (representing Monday
to Sunday), so we grab the value of the sixth element in `calendar.day_name` (notice in
the code that we have substituted `calendar` with "`cal`" for brevity).

The last two instructions show how to get detailed information out of a date object.
We can inspect its `day`, `month`, and `year` attributes, or call the `timetuple()` method
and get a whole wealth of information. Since we're dealing with a date object, notice
that all the information about time has been set to 0.

Let's now play with time:

```
>>> time.ctime()
'Sun Mar 28 15:23:17 2021'
>>> time.daylight
1
>>> time.gmtime()
time.struct_time(
    tm_year=2021, tm_mon=3, tm_mday=28,
    tm_hour=14, tm_min=23, tm_sec=34,
    tm_wday=6, tm_yday=87, tm_isdst=0
)
>>> time.gmtime(0)
time.struct_time(
    tm_year=1970, tm_mon=1, tm_mday=1,
    tm_hour=0, tm_min=0, tm_sec=0,
    tm_wday=3, tm_yday=1, tm_isdst=0
```

```
)
>>> time.localtime()
time.struct_time(
    tm_year=2021, tm_mon=3, tm_mday=28,
    tm_hour=15, tm_min=23, tm_sec=50,
    tm_wday=6, tm_yday=87, tm_isdst=1
)
>>> time.time()
1616941458.149149
```

This example is quite similar to the one before, only here, we are dealing with time. We can see how to get a printed representation of time according to C format standard, and then how to check if daylight saving time is in effect. The function `gmtime` converts a given number of seconds from the epoch to a `struct_time` object in UTC. If we don't feed it any number, it will use the current time.

> The **epoch** is a date and time from which a computer system measures system time. You can see that on the machine used to run this code, the epoch is January 1st, 1970. This is the point in time used by both Unix and POSIX.
>
> Coordinated Universal Time or **UTC** is the primary time standard by which the world regulates clocks and time.

We finish the example by getting the `struct_time` object for the current local time and the number of seconds from the epoch expressed as a float number (`time.time()`).

Let's now see an example using `datetime` objects, which bring together dates and times.

```
>>> now = datetime.now()
>>> utcnow = datetime.utcnow()
>>> now
datetime.datetime(2021, 3, 28, 15, 25, 16, 258274)
>>> utcnow
datetime.datetime(2021, 3, 28, 14, 25, 22, 918195)
>>> now.date()
datetime.date(2021, 3, 28)
>>> now.day, now.month, now.year
(28, 3, 2021)
>>> now.date() == date.today()
True
```

```
>>> now.time()
datetime.time(15, 25, 16, 258274)
>>> now.hour, now.minute, now.second, now.microsecond
(15, 25, 16, 258274)
>>> now.ctime()
'Sun Mar 28 15:25:16 2021'
>>> now.isoformat()
'2021-03-28T15:25:16.258274'
>>> now.timetuple()
time.struct_time(
    tm_year=2021, tm_mon=3, tm_mday=28,
    tm_hour=15, tm_min=25, tm_sec=16,
    tm_wday=6, tm_yday=87, tm_isdst=-1
)
>>> now.tzinfo
>>> utcnow.tzinfo
>>> now.weekday()
6
```

The preceding example is rather self-explanatory. We start by setting up two instances that represent the current time. One is related to UTC (utcnow), and the other one is a local representation (now). It just so happens that we ran this code on the first day after daylight saving time was introduced in the UK in 2021, so now represents the current time in BST. BST is one hour ahead of UTC when daylight saving time is in effect, as can be seen from the code.

You can get date, time, and specific attributes from a datetime object in a similar way as to what we have already seen. It is also worth noting how both now and utcnow present the value None for the tzinfo attribute. This happens because those objects are **naive**.

> Date and time objects may be categorized as *aware* if they include time zone information, or *naïve* if they don't.

Let's now see how a duration is represented in this context:

```
>>> f_bday = datetime(
    1975, 12, 29, 12, 50, tzinfo=ZoneInfo('Europe/Rome')
    )
>>> h_bday = datetime(
    1981, 10, 7, 15, 30, 50, tzinfo=timezone(timedelta(hours=2))
    )
>>> diff = h_bday - f_bday
```

```
>>> type(diff)
<class 'datetime.timedelta'>
>>> diff.days
2109
>>> diff.total_seconds()
182223650.0
>>> today + timedelta(days=49)
datetime.date(2021, 5, 16)
>>> now + timedelta(weeks=7)
datetime.datetime(2021, 5, 16, 15, 25, 16, 258274)
```

Two objects have been created that represent Fabrizio and Heinrich's birthdays. This time, in order to show you the alternative, we have created **aware** objects.

There are several ways to include time zone information when creating a `datetime` object, and in this example, we are showing you two of them. One uses the brand-new `ZoneInfo` object from the `zoneinfo` module, introduced in Python 3.9. The second one uses a simple `timedelta`, an object that represents a duration.

We then create the `diff` object, which is assigned as the subtraction of them. The result of that operation is an instance of `timedelta`. You can see how we can interrogate the `diff` object to tell us how many days Fabrizio and Heinrich's birthdays are apart, and even the number of seconds that represent that whole duration. Notice that we need to use `total_seconds`, which expresses the whole duration in seconds. The `seconds` attribute represents the number of seconds assigned to that duration. So, a `timedelta(days=1)` will have seconds equal to 0, and `total_seconds` equal to 86,400 (which is the number of seconds in a day).

Combining a `datetime` with a duration adds or subtracts that duration from the original date and time information. In the last few lines of the example, we can see how adding a duration to a `date` object produces a `date` as a result, whereas adding it to a `datetime` produces a `datetime`, as it is fair to expect.

One of the more difficult undertakings to carry out using dates and times is parsing. Let's see a short example:

```
>>> datetime.fromisoformat('1977-11-24T19:30:13+01:00')
datetime.datetime(
    1977, 11, 24, 19, 30, 13,
    tzinfo=datetime.timezone(datetime.timedelta(seconds=3600))
)
>>> datetime.fromtimestamp(time.time())
datetime.datetime(2021, 3, 28, 15, 42, 2, 142696)
```

We can easily create `datetime` objects from ISO-formatted strings, as well as from timestamps. However, in general, parsing a date from unknown formats can prove to be a difficult task.

Third-party libraries

To finish off this subsection, we would like to mention a few third-party libraries that you will very likely come across the moment you will have to deal with dates and times in your code:

- **dateutil**: Powerful extensions to `datetime` (`https://dateutil.readthedocs.io/en/stable/`)
- **Arrow**: Better dates and times for Python (`https://arrow.readthedocs.io/en/latest/`)
- **pytz**: World time zone definitions for Python (`https://pythonhosted.org/pytz/`)

These three are some of the most common, and they are worth investigating.

Let's take a look at one final example, this time using the Arrow third-party library:

```
>>> import arrow
>>> arrow.utcnow()
<Arrow [2021-03-28T14:43:20.017213+00:00]>
>>> arrow.now()
<Arrow [2021-03-28T15:43:39.370099+01:00]>

>>> local = arrow.now('Europe/Rome')
>>> local
<Arrow [2021-03-28T16:59:14.093960+02:00]>
>>> local.to('utc')
<Arrow [2021-03-28T14:59:14.093960+00:00]>
>>> local.to('Europe/Moscow')
<Arrow [2021-03-28T17:59:14.093960+03:00]>
>>> local.to('Asia/Tokyo')
<Arrow [2021-03-28T23:59:14.093960+09:00]>
>>> local.datetime
datetime.datetime(
    2021, 3, 28, 16, 59, 14, 93960,
    tzinfo=tzfile('/usr/share/zoneinfo/Europe/Rome')
)
>>> local.isoformat()
'2021-03-28T16:59:14.093960+02:00'
```

Arrow provides a wrapper around the data structures of the standard library, plus a whole set of methods and helpers that simplify the task of dealing with dates and times. You can see from this example how easy it is to get the local date and time in the Italian time zone (*Europe/Rome*), as well as to convert it to UTC, or to the Russian or Japanese time zones. The last two instructions show how you can get the underlying `datetime` object from an Arrow one, and the very useful ISO-formatted representation of a date and time.

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 described in *Table 2.1*.

Data type	Description
`namedtuple()`	Factory function for creating tuple subclasses with named fields
`deque`	List-like container with fast appends and pops on either end
`ChainMap`	Dictionary-like class for creating a single view of multiple mappings
`Counter`	Dictionary subclass for counting hashable objects
`OrderedDict`	Dictionary subclass with methods that allow for re-ordering entries
`defaultdict`	Dictionary subclass that calls a factory function to supply missing values
`UserDict`	Wrapper around dictionary objects for easier dictionary subclassing
`UserList`	Wrapper around list objects for easier list subclassing
`UserString`	Wrapper around string objects for easier string subclassing

Table 2.1: Collections module data types

There isn't enough space here to cover them all, but you can find plenty of examples in the official documentation; here, we will just give a small example to show you `namedtuple`, `defaultdict`, and `ChainMap`.

namedtuple

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 fully-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 only 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 also refactor all the logic involved, which can be very tricky.

For example, say we are handling data about the left and right eyes 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 may look:

```
>>> 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` objects all of 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 it 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 that can be painful. We could have probably approached this a bit better from the beginning, by using a `namedtuple`. Let us show you what we 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 the left and right eyes using `vision.left` and `vision.right`, all we need to do to fix the new design issue is 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 correct
9.5
>>> vision.right  # still correct (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, as a wise man once wrote, *Explicit is better than implicit* (Can you recall where? Think *Zen* if you can'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 occur in a professional environment, and how painful it is to refactor in such cases.

defaultdict

The **defaultdict** data type is one of our favorites. It allows you to avoid checking whether 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 an `if` clause that runs to a length of four lines, and that we would have to write if dictionaries didn't have the `get()` method (we'll see all about `if` clauses in *Chapter 3*, *Conditionals and Iteration*):

```
>>> 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
```

Notice how we just need to instruct the `defaultdict` factory that we want an `int` number to be used if 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. To dig deeper, please refer to the official documentation.

ChainMap

ChainMap is an extremely useful 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. By 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
[{'port': 5678, 'host': 'packtpub.com'},
 {'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}
```

Isn't it just lovely that Python makes your life so 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 can just feed the `ChainMap` object to a `dict` constructor. If you have ever coded in other languages, such as Java or C++, you probably will be able to appreciate how precious this is, and how well Python simplifies some tasks.

Enums

Technically not a built-in data type, as you have to import them from the enum module, but definitely worth mentioning, are **enumerations**. They were introduced in Python 3.4, and though it is not that common to see them in professional code, we thought it would be a good idea to give you an example anyway for the sake of completeness.

The official definition of an enumeration is that it is *a set of symbolic names (members) bound to unique, constant values. Within an enumeration, the members can be compared by identity, and the enumeration itself can be iterated over.*

Say you need to represent traffic lights; in your code, you might resort to the following:

```
>>> GREEN = 1
>>> YELLOW = 2
>>> RED = 4
>>> TRAFFIC_LIGHTS = (GREEN, YELLOW, RED)
>>> # or with a dict
>>> traffic_lights = {'GREEN': 1, 'YELLOW': 2, 'RED': 4}
```

There's nothing special about this code. It's something, in fact, that is very common to find. But, consider doing this instead:

```
>>> from enum import Enum
>>> class TrafficLight(Enum):
...     GREEN = 1
...     YELLOW = 2
...     RED = 4
...
>>> TrafficLight.GREEN
<TrafficLight.GREEN: 1>
>>> TrafficLight.GREEN.name
'GREEN'
>>> TrafficLight.GREEN.value
1
>>> TrafficLight(1)
<TrafficLight.GREEN: 1>
>>> TrafficLight(4)
<TrafficLight.RED: 4>
```

Ignoring for a moment the (relative) complexity of a class definition, you can appreciate how this approach may be advantageous. The data structure is much cleaner, and the API it provides is much more powerful. We encourage you to check out the official documentation to explore all the great features you can find in the enum module. We think it's worth exploring, at least once.

Final considerations

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

Before we leap into *Chapter 3*, *Conditionals and Iteration*, we'd like to share some final considerations about different aspects that, to our minds, are important and not to be neglected.

Small value caching

While discussing 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
```

Uh-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 *performance*. Python caches short strings and small numbers to avoid having many copies of them clogging up the system memory. In the case of strings, caching or, more appropriately, interning them, also provides a significant performance improvement for comparison operations. Everything is handled properly under the hood, so you don't need to worry, 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 we know more about these customers, it's very hard to answer. What kind of access will we need? What sort of operations will we have to perform on each of them, and how many times? Will the collection change over time? Will we need to modify the customer dictionaries in any way? What is going to be the most frequent operation we 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 us draft these options for you:

```python
# 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? We probably wouldn't go with the tuple option, unless we wanted to highlight that the collection is not going to change. We would say that, usually, a list is better, as it allows for more flexibility.

Another factor to keep in mind is that tuples and lists are ordered collections. If you use a dictionary (prior to Python 3.6) or a set, you would lose the ordering, so you need to know if ordering is important in your application.

What about performance? For example, in a list, operations such as insertion and membership testing can take *O(n)* time, 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 search "**big O notation**". 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 $t \le c * f(n)$ 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 also be used to size other measurable quantities, of course).

Another way of understanding whether 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 to 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 in.

About indexing and slicing

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

There is one characteristic regarding Python indexing that we haven't mentioned before. We'll show you by way of an 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 list `a` has 10 elements, then due to 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 performing data manipulation. *Figure 2.2* displays a neat diagram about how indexing works on the string `"HelloThere"` (which is Obi-Wan Kenobi sarcastically greeting General Grievous in *Star Wars: Episode III – Revenge of the Sith*):

Positive Indexing →

0	1	2	3	4	5	6	7	8	9
H	**e**	**l**	**l**	**o**	**T**	**h**	**e**	**r**	**e**
-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

← Negative Indexing
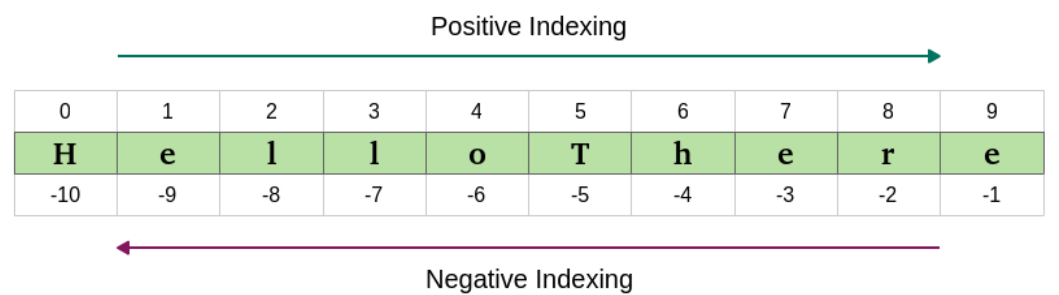
Figure 2.2: Python indexing

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

About names

You may have noticed that, in order to keep the examples as short as possible, we have named many objects using simple letters, like `a`, `b`, `c`, `d`, and so on. This is perfectly fine when debugging on the console or showing that `a + b == 7`, but it's bad practice when it comes to professional coding (or any type of coding, for that matter). We hope you will indulge us where we have done 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—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 datatype? We don't think so, at least in most cases. So, if you have an excellent reason to do so, go ahead; otherwise, don't. The reasoning behind this is that once `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 from the book *Clean Code* by *Robert C. Martin* is entirely dedicated to names. It's an amazing book that helped us improve our coding style in many different ways; it is 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 there are and how much can be achieved just by using them in different combinations.

We've seen number types, sequences, sets, mappings, dates, times, and collections (and a special guest appearance by enumerations). We have also seen that everything is an object, learned the difference between mutable and immutable, and we've also learned about slicing and indexing.

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

Most of all, we encourage you to try out all of 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, and when you massage strings. Play with all data types. Exercise them, break them, discover all their methods, enjoy them, and learn them very, very well. If your foundation is not rock solid, how good can your code be? Data is the foundation for everything; data shapes what dances around it.

The more you progress with the book, the more likely it is that you will find some discrepancies or maybe a small typo here and there in our code (or yours). You will get an error message, something will break. That's wonderful! When you code, things break and you have to debug them, 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, that's for sure, so you may as well start making your peace with them now.

The next chapter is about conditionals and iteration. We'll see how to actually put collections to use, and make decisions based on the data that 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, and break things—it's a very good way to learn.

3
Conditionals and Iteration

"Would you tell me, please, which way I ought to go from here?"

"That depends a good deal on where you want to get to."

— Lewis Carroll, from Alice's Adventures in Wonderland

In the previous chapter, we looked at Python's built-in data types. Now that you are 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 **flow of control**) is the order in which 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 the possible forms of those two constructs in a *documentation* fashion, we'd rather give you the basics, and then write a couple of small scripts with you. In the first one, we will see how to create a rudimentary prime number generator, while in the second, we'll see how to apply discounts to customers based on coupons. This way, you should get a better feeling for how conditional programming and looping can be used.

In this chapter, we are going to cover the following:

- Conditional programming
- Looping in Python
- Assignment expressions
- A quick peek at the `itertools` module

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 its basic function is to evaluate an expression and, based on the result, choose which part of the code to execute. As usual, let's look at 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 the code immediately after the `if` statement. Notice that the `print` instruction is indented, which means that 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 we set `late = False`, so when we 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 is evident by comparing the preceding two examples.

A specialized else: elif

Sometimes all you need is to do something if a condition is met (a simple `if` clause). At 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; since calling the manager (or not calling them) is a type of binary example (either you call or you don't), let's change the type of example and keep expanding. This time, we decide on tax percentages. If your income is less than $10,000, you don't need to pay any taxes. If it is between $10,000 and $30,000, you have to pay 20% in taxes. If it is between $30,000 and $100,000, you pay 35% in taxes, and if you're fortunate enough to earn over $100,000, you must pay 45% in taxes. Let's put this all down into beautiful Python code:

```python
# 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(f'You will pay: ${income * tax_coefficient} in taxes')
```

Executing the preceding code yields:

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

Let's go through the example one line at a time. We start by setting up the income value. In the example, your income is $15,000. We enter the `if` clause. Notice that this time we also introduced the `elif` clause, which is a contraction of `else-if`, and it's different from a bare `else` clause in that it also has its own condition. So, the `if` expression of `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 the `if` clause from now on). There is only one instruction after the `if` clause, the `print()` call, which tells us that you will pay $3000.0 in taxes this year (*15,000 * 20%*). Notice that the order is mandatory: `if` comes first, then (optionally) as many `elif` clauses as you may need, and then (optionally) a single `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 will (one per execution, of course). And then try the **boundaries**. This is crucial, as whenever you have conditions expressed as equalities or inequalities (==, !=, <, >, <=, >=), those numbers represent boundaries. It is essential to test boundaries thoroughly. Should we allow you to drive at 18 or 17? Are we checking your age with `age < 18`, or `age <= 18`? You can't imagine how many times we've 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 (10,000, 30,000, 100,000) as well as any value in between. See how the result changes, and get a good understanding of it before proceeding.

Let's now 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 email, we send it according to the severity of the error. If the alert system is anything other than console or email, 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 because of how silly it is. It shows us two nested `if` clauses (*outer* and *inner*). It also shows us that 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.

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 call the inner clause. In the inner `if` clause, according to `error_severity`, we send an email either to 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 this book, we included a trick to redirect that call to a regular `print()` function, just so you can experiment on the console without actually sending an email. Try changing the values and see how it all works.

The ternary operator

One last thing we would like to show you, before moving on, 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, it is sometimes easier and more readable to use the ternary operator instead of a proper `if` clause. For example, instead of:

```
# ternary.py
order_total = 247  # GBP

# classic if/else form
```

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

We could write:

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

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

> In some languages (like C or JavaScript) the ternary operator is even more concise. For example, the above could be written as:
>
> ```
> discount = order_total > 100 ? 25 : 0;
> ```
>
> Although Python's version is slightly more verbose, we think it more than makes up for that by being much easier to read and understand.

Are you clear on how the ternary operator works? Basically, `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 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 do serve different purposes, and therefore they're usually used in different contexts. We'll explore this difference thoroughly in this chapter.

The for loop

The `for` loop is used when looping over a sequence, such as a list, tuple, or collection of objects. Let's start with a simple example and expand on the concept to see what the Python syntax allows us to do:

```python
# 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 the order given), then the body of the loop is executed (the `print()` line). The `number` value 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:

```python
# 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:

```python
>>> 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 are only interested in understanding what values it will return to us. You can see that the behavior is the same as with slicing (which we described in the previous chapter): `start` is included, `stop` is 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

We now 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. We start from the innermost part of what we're trying to understand, and we expand outward. 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 the 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 value `2` in the third and final 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 such as 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. Let's change the example into a more Pythonic form:

```python
# 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 iteration. Running this code will print the three surnames, one at a time, which is much easier to read—right?

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

```python
# 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 two-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, such as `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 to designing Python so that it makes your life easier.

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

Iterators and iterables

According to the Python documentation (`https://docs.python.org/3/glossary.html`), 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__() method or with a __getitem__() method that implements Sequence semantics.*

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` 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, such as lists, tuples, dictionaries, and strings, while others, such as sets, do not. Python gives us the ability to iterate over iterables, using a type of object called an **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 do not fully understand all the preceding legalese, as you will in due course. We have put it here to serve 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 in which you write 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 the pair person/age on one line for each of them. Let's start with an example, which we will refine gradually:

```python
# multiple.sequences.py
people = ['Nick', 'Rick', 'Roger', 'Syd']
ages = [23, 24, 23, 21]
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
Nick 23
Rick 24
Roger 23
Syd 21
```

The code works, but it's not very Pythonic. It's rather cumbersome to have to get the length of `people`, construct a `range`, and then iterate over that. For some data structures it may also be expensive to retrieve items by their position. Wouldn't it be nice if we could use the same approach as for iterating over a single sequence? Let's try to improve it by using `enumerate()`:

```python
# multiple.sequences.enumerate.py
people = ['Nick', 'Rick', 'Roger', 'Syd']
ages = [23, 24, 23, 21]
for position, person in enumerate(people):
    age = ages[position]
    print(person, age)
```

That's better, but still not perfect. And it's 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:

```python
# multiple.sequences.zip.py
people = ['Nick', 'Rick', 'Roger', 'Syd']
ages = [23, 24, 23, 21]
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 we wanted to show this example is twofold. On the one hand, we wanted to give you an idea of how much shorter 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:

```python
# multiple.sequences.explicit.py
people = ['Nick', 'Rick', 'Roger', 'Syd']
ages = [23, 24, 23, 21]
instruments = ['Drums', 'Keyboards', 'Bass', 'Guitar']
for person, age, instrument in zip(people, ages, instruments):
    print(person, age, instrument)
```

In the preceding code we added the `instruments` list. Now that we feed three sequences to the `zip()` function, the `for` loop gets back a *three-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
Nick 23 Drums
Rick 24 Keyboards
Roger 23 Bass
Syd 21 Guitar
```

Sometimes, for reasons that may not be clear in a simple example such as 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:

```python
# multiple.sequences.implicit.py
people = ['Nick', 'Rick', 'Roger', 'Syd']
```

```python
ages = [23, 24, 23, 21]
instruments = ['Drums', 'Keyboards', 'Bass', 'Guitar']
for data in zip(people, ages, instruments):
    person, age, instrument = data
    print(person, age, instrument)
```

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 three-tuple `data` that comes from `zip(...)` is exploded within the body of the `for` loop into three variables: `person`, `age`, and `instrument`.

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 decide which looping construct to use, is that the `for` loop is best suited in cases where you need to iterate over the elements of some container or other iterable object.

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, such as 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. The difference is that the `while` loop doesn't loop over a sequence (it can, but you have to write the logic manually, which would make little sense as you would just 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 that will clarify everything for us. We want to print the binary representation of a positive number. In order to do so, we can use a simple algorithm that divides by `2` until we reach `0` and collects the remainders. When we reverse the list of remainders we collected, we get the binary representation of the number we started with:

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

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

```python
# 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

remainders.reverse()
print(remainders)
```

In the preceding code, we highlighted n > 0, which is the condition to keep looping. Notice how the code matches the algorithm we described: as long as n is greater than 0, we divide by 2 and add the remainder to a list. At the end (when n has reached 0) we reverse the list of remainders to get the binary representation of the original value of n.

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*:

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

remainders.reverse()
print(remainders)
```

In the preceding code, we have reassigned n to the result of the division by 2, along with 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 whether 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, check out the `bin()` function.

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

```python
# multiple.sequences.while.py
people = ['Nick', 'Rick', 'Roger', 'Syd']
ages = [23, 24, 23, 21]
position = 0
while position < len(people):
    person = people[position]
    age = ages[position]
    print(person, age)
    position += 1
```

In the preceding code, we have highlighted the *initialization*, *condition*, and *update* of the `position` variable, 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 is a bit of boilerplate you have to go through in order to achieve the same result. The opposite is also true, but unless you have a reason to do so, you ought to use the right tool for the job, and 99.9% of the time you'll be fine.

So, to recap, use a `for` loop when you need to iterate over an 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 us 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 as you want), or you can break out of the loop entirely. A common use case for skipping iterations is, for example, when you are 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 take a look at a couple of examples.

Let's say you want to apply a 20% discount to all products in a basket list for those that 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 stop execution of the body immediately and go to the next iteration, if any. This example will take us a little deeper down the rabbit hole, so be ready to jump:

```python
# 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'])
```

We start by importing the `date` and `timedelta` objects, then we set up our products. Those with `sku` as `1` and `3` have an expiration date of `today`, which means we want to apply a 20% discount on them. We loop over each `product` and 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 it 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
```

This 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 whether at least one of the elements in a list evaluates to `True` when fed to the `bool()` function. Given that we need to know whether 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:

```python
# 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 makes use of a very common programming pattern; you set up a **flag** variable before starting the inspection of the items. If you find an element that matches your criteria (in this example, that evaluates to `True`), 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 it also prevents any further iterations from running, effectively breaking out of the loop. The `continue` and `break` statements can be used together with no limitation in their numbers, both in the `for` and `while` looping constructs.

> There is no need to write code to detect whether there is at least one element in a sequence that evaluates to `True`. Just check out the built-in `any()` function.

A special else clause

One of the features we'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 useful 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. If 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. If we don't find at least one that satisfies the condition, we want to raise an **exception**. This means that 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, Exceptions and Context Managers*, so don't worry if you don't fully understand them for now. Just bear in mind that they will alter the regular flow of the code.

Let us 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:

```python
# 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 the 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 whether one was found. We kind of have the feeling that those kids would drive a very *metallic-a* car, but anyway, notice that if a driver is not found, `DriverException` is raised, signaling to 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 are not 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.

> In his *Transforming Code into Beautiful, Idiomatic Python* video, Raymond Hettinger suggests a much better name for the `else` statement associated with a `for` loop: `nobreak`. If you struggle with remembering how the `else` works for a `for` loop, simply remembering this fact should help you.

Assignment expressions

Before we look at some more complicated examples, we would like to briefly introduce you to a relatively new feature that was added to the language in Python 3.8, via PEP 572 (`https://www.python.org/dev/peps/pep-0572`). Assignment expressions allow us to bind a value to a name in places where normal assignment statements are not allowed. Instead of the normal assignment operator `=`, assignment expressions use `:=` (known as the **walrus operator** because it resembles the eyes and tusks of a walrus).

Statements and expressions

To understand the difference between normal assignments and assignment expressions, we need to understand the difference between statements and expressions. According to the Python documentation (`https://docs.python.org/3/glossary.html`), a **statement** is:

> *…part of a suite (a "block" of code). A statement is either an expression or one of several constructs with a keyword, such as* if, while *or* for.

An **expression**, on the other hand, is:

> *A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value.*

The key distinguishing feature of an expression is that it has a return value. Notice that an expression can be a statement, but not all statements are expressions. In particular, assignments like `name = "heinrich"` are not expressions, and so do not have a return value. This means that you cannot use an assignment statement in the conditional expression of a `while` loop or `if` statement (or any other place where a value is required).

> Have you ever wondered why the Python console doesn't print a value when you assign a value to a name? For example:
>
> ```
> >>> name = "heinrich"
> >>>
> ```
>
> Well, now you know! It's because what you've entered is a statement, which doesn't have a return value to print.

Using the walrus operator

Without assignment expressions, you would have to use two separate statements if you want to bind a value to a name and use that value in an expression. For example, it is quite common to see code similar to:

```python
# walrus.if.py
remainder = value % modulus
if remainder:
    print(f"Not divisible! The remainder is {remainder}.")
```

With assignment expressions, we could rewrite this as:

```python
# walrus.if.py
if remainder := value % modulus:
    print(f"Not divisible! The remainder is {remainder}.")
```

Assignment expressions allow us to write fewer lines of code. Used with care, they can also lead to cleaner, more understandable code. Let's look at a slightly bigger example to see how an assignment expression can really simplify a `while` loop.

In interactive scripts, we often need to ask the user to choose between a number of options. For example, suppose we are writing an interactive script that allows customers at an ice cream shop to choose what flavor they want. To avoid confusion when preparing orders, we want to ensure that the user chooses one of the available flavors. Without assignment expressions, we might write something like this:

```python
# menu.no.walrus.py
flavors = ["pistachio", "malaga", "vanilla", "chocolate", "strawberry"]
prompt = "Choose your flavor: "
print(flavors)
while True:
    choice = input(prompt)
    if choice in flavors:
        break
    print(f"Sorry, '{choice}' is not a valid option.")
print(f"You chose '{choice}'.")
```

Take a moment to read this code carefully. Notice the condition on that loop: `while True` means "loop forever." That's not what we really want, is it? We want to stop looping when the user inputs a valid flavor (`choice in flavors`). To achieve that, we've used an `if` statement and a `break` inside the loop. The logic to control the loop is not immediately obvious. In spite of that, this is actually quite a common pattern when the value needed to control the loop can only be obtained inside the loop.

> The `input()` function is very useful in interactive scripts. It allows you to prompt the user for input, and returns it as a string. You can read more about it in the official Python documentation.

How can we improve on this? Let us try to use an assignment expression:

```python
# menu.walrus.py
flavors = ["pistachio", "malaga", "vanilla", "chocolate", "strawberry"]
prompt = "Choose your flavor: "
print(flavors)
while (choice := input(prompt)) not in flavors:
    print(f"Sorry, '{choice}' is not a valid option.")
print(f"You chose '{choice}'.")
```

Now the loop's conditional expression says exactly what we want. That is much easier to understand. The code is also three lines shorter.

> Did you notice the parentheses around the assignment expression? We need them because the `:=` operator has lower precedence than the `not in` operator. Try removing the parentheses and see what happens.

We have seen examples of using assignment expressions in the conditional expressions of `if` and `while` statements. Besides these use cases, assignment expressions are also very useful in *lambda expressions* (which you will meet in *Chapter 4, Functions, the Building Blocks of Code*) as well as *comprehensions* and *generators* (which you will learn about in *Chapter 5, Comprehensions and Generators*).

A word of warning

The introduction of the walrus operator in Python was somewhat controversial. Some people feared that it would make it too easy to write ugly, non-Pythonic code. We think that these fears are not entirely justified. As you saw above, the walrus operator can *improve* code and make it easier to read. Like any powerful feature, it can however be abused to write *obfuscated* code. We would advise you to use it sparingly. Always think carefully about how it impacts the readability of your code.

Putting all this 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 look at those two examples we anticipated at the beginning of this chapter. We'll mix and match here, so you can see how you 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 we are going to write a very inefficient and rudimentary algorithm to detect primes. The important thing is to concentrate on those bits in the code that belong to this chapter's subject.

A prime generator

According to Wikipedia:

> A **prime number** (or a **prime**) is a natural number greater than 1 that is not a product of two smaller natural numbers. A natural number greater than 1 that is not prime 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, and 10 are not. In order to have a computer tell you whether a number, N, is prime, you can divide that number by the natural numbers in the range *[2, N)*. If any of those divisions yields zero as a remainder, then the number is not a prime. We will write two versions of this, the second of which will exploit the `for...else` syntax:

```python
# primes.py
primes = []  # this will contain the primes at 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)
```

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

So, there are 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 value of `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 loop, the outer one keeps on going as normal. 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 proceeds, 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 proceeding, we will pose the following: of all the iterations of the outer loop, one of them is different from all the others. Can you tell which one this is—and why? Think about it for a second, go back to the code, 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. We asked you to do it as a small exercise because this is 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. We'll tell you the answer now: the iteration that behaves differently from all others is the first one. The reason is that in the first iteration, `n` is `2`. Therefore the innermost `for` loop won't even run, because it's a `for` loop that 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 whether anything happens.

Now, from an algorithmic point of view, this code is inefficient; let's at least make it a bit easier on the eyes:

```python
# 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 better, right? The `is_prime` flag is 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?

Applying discounts

In this example, we want to show you a technique that we are very fond of. In many programming languages, besides the `if/elif/else` constructs, in whatever form or syntax they may come, you can find another statement, usually called `switch/case`, that is not in Python. It is the equivalent of a cascade of `if/elif/.../elif/else` clauses, with a syntax similar to this (warning, we are using JavaScript code here):

```javascript
/* 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 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 number—that is, not in [0, 6]. Python is perfectly capable of realizing such logic using `if`/`elif`/`else` statements:

```python
# 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. We raised the `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 we 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. We'll keep the logic down to a minimum here—remember that all we really care about is understanding conditionals and loops:

```python
# coupons.py
customers = [
```

```python
    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. We made up four different types of coupons: two are fixed and two are percentage-based. You can see that in the `if/elif/else` cascade we apply the discount accordingly, and we set it as a `'discount'` key in the `customer` dictionary.

At the end, we just print out part of the data to see whether our 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 conditional clauses are cluttering the logic. It's not easy to see what's going on at a first glance, which we don't like. In cases like this, you can exploit a dictionary to your advantage, like this:

```python
# 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 is now just three lines long, and very easy to understand. The concept here is to use a dictionary as a **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 we 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 two-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.

This technique is important, because it is also used in other contexts with functions where it becomes much more powerful than what we've seen in the preceding example. Another advantage of using it is that you can code it in such a way that the keys and values of the discounts dictionary are fetched dynamically (for example, from a database). This will allow the code to adapt to whatever discounts and conditions you have, without having to modify anything.

If you are still unclear as to how this works, we suggest you 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 a few words about the `itertools` module. If you are into iterating, this is a kind of heaven.

According to the Python official documentation (`https://docs.python.org/3/library/itertools.html`), the `itertools` module:

> *…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 we have the room here to show you all the goodies you can find in this module, so we encourage you to go check it out for yourself. We can promise that you will enjoy it, though. In a nutshell, it provides you with three broad categories of iterators. We shall 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, such as if it were a `while` loop:

```python
# 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 outputs:

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

The `count` factory class makes an iterator that simply goes on and on counting. It starts from 5 and keeps adding 3 to it. We need to break it manually 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, if any of them are shorter than the rest, the resulting iterator won't break, but will simply stop as soon as the shortest iterator is exhausted. This is very theoretical, we know, so let us 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 `1`. Let's see a simple example:

```python
# 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 in length, while `data` is only 10. `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 convenient way of selecting elements out of an iterable. The code is very simple, but 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, it puts all the values into a list and returns it.

Combinatoric generators

Last but not least is combinatoric generators. These are really fun, if you are into this kind of thing. Let's look at 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, there are six permutations of ABC: 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 ABC string, the permutations are *3! = 3 * 2 * 1 = 6*. Let's see this in Python:

```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 permutations. 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 toward expanding our Python vocabulary. We've seen how to drive the execution of code by evaluating conditions, along with 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 of how to shape it so that it does what we want, having it react 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 took a brief look at the itertools module, which is full of interesting iterators that can enrich our abilities with Python to a greater degree.

Now it's time to switch gears, take another step forward, and talk about functions. The next chapter is all about them, and they are extremely important. Make sure you're comfortable with what has been covered up to now. We want to provide you with interesting examples, so we'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? Functions and objects."

– Le Corbusier

In the previous chapters, we have seen that everything is an object in Python, and functions are no exception. But what exactly is a function? A function is *a sequence of instructions that perform a task, bundled together as a unit*. This unit can then be imported and used wherever it is needed. There are many advantages to using functions in your code, as we'll see shortly.

In this chapter, we are going to cover the following:

- Functions—what they are and why we should use them
- Scopes and name resolution
- Function signatures—input parameters and return values
- Recursive and anonymous functions
- Importing objects for code reuse

We believe the saying *a picture is worth a thousand words* is particularly true when explaining functions to someone who is new to this concept, so please take a look at the following diagram in *Figure 4.1*.
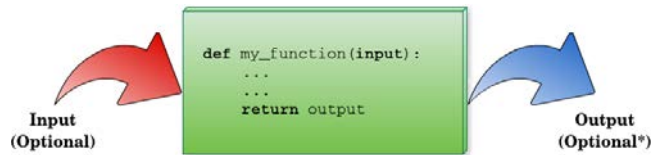
Figure 4.1: An example function

As you can see, a function is a block of instructions, packaged as a whole, like a box. Functions can accept input parameters 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 parentheses (which may or may not contain input parameters); a colon (`:`) then signals the end of the function definition line. Immediately afterward, 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 number of spaces suggested by PEP 8, and, in practice, it is the most widely used spacing measure.

A function may or may not return an output. If a function wants to return an output, it does so by using the `return` keyword, followed by the desired output. The eagle-eyed may have noticed the little * after *Optional* in the output section of the preceding diagram. 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, or no value is given to the `return` statement itself, the function returns `None`. The reasons behind this design choice are outside 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 us give you a few reasons why we need them:

- They reduce code duplication in a program. By having a specific task be 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 us now look at a few examples to get a better understanding of each point.

Reducing code duplication

Imagine that you are writing a piece of scientific software, and you need to calculate prime numbers up to a certain limit—as we did in the previous chapter. You have a nice algorithm to calculate them, so you copy and paste it to wherever you need. One day, though, your friend, B. Riemann, gives you a better algorithm to calculate primes, which will save you a lot of time. At this point, you need to go over your whole code base and replace the old code with the new one.

This is actually a bad way to go about it. It's error-prone, you never know what lines you are chopping out or leaving in by mistake or when you might be cutting and pasting code into other code, and you may also risk missing one of the places where prime calculation is done, leaving your software in an inconsistent state where the same action is performed in different places in different ways. What if, instead of replacing code with a better version of it, you need to fix a bug and you miss one of the places? That would be even worse. What if the names of the variables in the old algorithm are different than those used in the new one? That will also complicate things.

So, what should you do? Simple! You write the function `get_prime_numbers(upto)` and use it anywhere you need a list of primes. When **B. Riemann** 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, nor will undetected bugs be left behind due to copy-and-paste failures or oversights. Use functions, and you'll only gain from it.

Splitting a complex task

Functions are also very useful for splitting long or complex tasks into smaller ones. The end result is that the code benefits from it in several ways—for example, through its readability, testability, and reusability.

To give you a simple example, imagine that you are preparing a report. Your code needs to fetch data from a data source, parse it, filter it, and polish it, and then a whole series of algorithms needs to be run against it, in order to produce the results that will feed the Report class. It's not uncommon to read procedures like this that are just one big do_report(data_source) function. There are tens or hundreds of lines of code that end with return report.

These situations are slightly more common in scientific code, which tends to be brilliant from an algorithmic point of view but sometimes lacks the touch of experienced programmers when it comes to the style in which they are written. Now, picture a few hundred lines of code. It's very hard to follow through, to find the places where things are changing context (such as 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:

```python
# 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 that you need to suspend). Code like this is easier to deal with.

Hiding implementation details

Let's stay with the preceding example to talk about this point as well. We can see that, by going through the code of the do_report() function, we 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 we don't need to delve into the details, we are not forced to, in the way that we would be if do_report was just one big, fat function. In order to understand what was going on, we would have to read every single line of code. With functions, we don't need to. This reduces the time we 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 to a minimum.

Improving readability

Programmers 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, like in the example below:

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

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:

```
# 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 of a and b in the second example, and it's much easier to read through the code. If we don't need to modify that multiplication logic, we don't even need to go into the implementation details. Therefore, readability is improved here, while, in the first snippet, we would have to spend time trying to understand what that complicated list comprehension is doing.

> Don't worry if you don't understand list comprehensions, as we will study them in *Chapter 5*, *Comprehensions and Generators*.

Improving traceability

Imagine that we have coded for an e-commerce website. We have displayed the product prices all over the pages. Imagine that the prices in the database are stored with no VAT (sales tax), but we want to display them on the website with VAT at 20%. Here are a few ways of calculating the VAT-inclusive price from the VAT-exclusive price:

```python
# 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
```

These four different ways of calculating a VAT-inclusive price are all perfectly acceptable; we have encountered all of them in the professional code that we have worked on over the years. Now, imagine that we have started selling products in different countries, and some of them have different VAT rates, so we need to refactor the code (throughout the website) in order to make that VAT calculation dynamic.

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

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

```python
# vat.function.py
def calculate_price_with_vat(price, vat):
    return price * (100 + vat) / 100
```

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

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

Scopes and name resolution

Do you remember when we talked about scopes and namespaces in *Chapter 1, A Gentle Introduction to Python*? We're going to expand on that concept now. Finally, we can talk in terms of functions, and this will make everything easier to understand. Let's start with a very simple example:

```python
# 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)
```

We have defined the `test` name 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 `my_function()` function (`test = 1`). If we execute the code, we will see this:

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

It's clear that `test = 1` shadows the `test = 0` assignment 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 `test` name again in the function body, and we set it to point to an integer of value 1. Both of the two `test` names therefore exist: one in the global scope, pointing to an int object with a value of 0, the other in the `my_function()` scope, pointing to an `int` object with a value of 1. Let's comment out the line with `test = 1`. Python searches for the `test` name in the next enclosing namespace (recall the **LEGB** rule: **local**, **enclosing**, **global**, **built-in**, described in *Chapter 1, A Gentle Introduction to Python*) 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:

```python
# 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 `test = 1` line. Can you figure out what the result will be? Well, when reaching the `print('outer:', test)` line, 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 whether you understand what happens and whether the **LEGB** rule is clear, before proceeding.

Another thing to note is that Python gives us 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

In 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 `inner()` function, we overwrite test neither in the `outer()` function 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 `inner()` function and see what happens:

```python
# scoping.level.2.nonlocal.py
def outer():
    test = 1  # outer scope

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

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


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

Notice how in the body of the `inner()` function, we 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 `inner()` function, we actually get to bind the `test` name to the one declared in the `outer` function. If we removed the `nonlocal test` line from the `inner()` function and tried the same trick in the `outer()` function, 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:

```python
# 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 `test` name 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 `test = 2` assignment 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, and get comfortable with scopes and name resolution—it's very important. Also, can you tell what would happen if you defined `inner()` outside `outer()` in the preceding examples?

Input parameters

At the beginning of this chapter, we saw that a function can take input parameters. Before we delve into all the possible types of parameters, let's make sure you have a clear understanding of what passing an argument 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

Before we explore the topic of arguments any further, please allow us to clarify the terminology a little. According to the official Python documentation:

> "*Parameters are defined by the names that appear in a function definition, whereas arguments are the values actually passed to a function when calling it. Parameters define what types of arguments a function can accept.*"

We will try to be precise when referring to parameters and arguments, but it is worth noting that they are sometimes used synonymously as well. Let's now look at some examples.

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 finally we call it, passing x:

```python
# key.points.argument.passing.py
x = 3
def func(y):
    print(y)

func(x)  # prints: 3
```

When func() is called with x, within its local scope, a name, y, is created, and it's pointed to the same object that x is pointing to. This is better clarified in *Figure 4.2* (don't worry about the fact that this example was run with Python 3.6—this is a feature that hasn't changed).
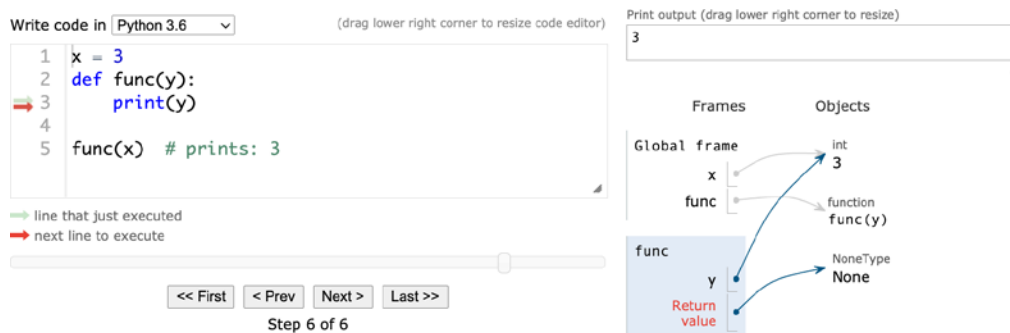
Figure 4.2: Understanding argument-passing with Python Tutor

The right-hand side of *Figure 4.2* 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 respectively to an **int** (with a value of 3) and to a function object. Right beneath 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 on the left side of the figure), y is pointing to the same object that x is. 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 (but perhaps 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 previously in this chapter.

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

Assignment to parameter names

Assignment to parameter names doesn't affect the caller. This is something that can be tricky to understand at first, so let's look at an example:

```python
# 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 we call the function with func(x), the instruction x = 7 is executed within the local scope of the func() function; the name, x, is pointed to an integer with a value of 7, leaving the global x unaltered.

Changing a mutable object

Changing a mutable object affects the caller. This is the final point, and it's very important because Python *apparently* behaves differently with mutable objects (just apparently, though). Let's look at an example:

```python
# key.points.mutable.py
x = [1, 2, 3]
def func(x):
```

```
    x[1] = 42  # this affects the 'x' argument!

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; within the body of the function, we are not changing x, in that we're not changing its reference, or, in other words, we are not changing which object x is pointing to. We are merely accessing the element at position 1 in that object, and changing its value.

Remember *point 2* the *Input parameters* section: *Assigning an object to an parameter 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 original 'x' argument!
    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 we have highlighted. At first, like before, 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 'something else' string. This leaves the caller unaltered 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. Once again, the Python Tutor website (`http://www.pythontutor.com/`) will help you a lot by giving you a visual representation of these concepts.

Now that we have a good understanding of input parameters and how they behave, let's look at the different ways of passing arguments to functions.

Passing arguments

There are four different ways of passing arguments to a function:

- Positional arguments
- Keyword arguments
- Iterable unpacking
- Dictionary unpacking

Let's take a look at them one by one.

Positional arguments

When we call a function, each positional argument is assigned to the parameter in the corresponding *position* in the function definition:

```python
# arguments.positional.py
def func(a, b, c):
    print(a, b, c)

func(1, 2, 3)  # prints: 1 2 3
```

This is the most common way of passing arguments to functions (and in some programming languages this is also the only way of passing arguments).

Keyword arguments

Keyword arguments in a function call are assigned to parameters using the `name=value` syntax:

```python
# arguments.keyword.py
def func(a, b, c):
    print(a, b, c)

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

When we use keyword arguments, the order of the arguments does not need to match the order of the parameters in the function definition. This can make our code easier to read and debug. We don't need to remember (or look up) the order of parameters in a function definition. We can look at a function call and immediately know which argument corresponds to which parameter.

You can also use both positional and keyword arguments at the same time:

```python
# arguments.positional.keyword.py
def func(a, b, c):
    print(a, b, c)


func(42, b=1, c=2)
```

Keep in mind, however, that positional arguments always have to be listed before any keyword arguments. For example, if you try something like this:

```python
# arguments.positional.keyword.py
func(b=1, c=2, 42)  # positional argument after keyword arguments
```

You will get an error:

```
$ python arguments.positional.keyword.py
  File "arguments.positional.keyword.py", line 7
    func(b=1, c=2, 42)  # positional argument after keyword arguments
                   ^
SyntaxError: positional argument follows keyword argument
```

Iterable unpacking

Iterable unpacking uses the syntax *iterable_name to pass the elements of an *iterable* as positional arguments to a function:

```python
# arguments.unpack.iterable.py
def func(a, b, c):
    print(a, b, c)


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

This is a very useful feature, particularly when we need to programmatically generate arguments for a function.

Dictionary unpacking

Dictionary unpacking is to keyword arguments what iterable unpacking is to positional arguments. We use the syntax **dictionary_name to pass keyword arguments, constructed from the keys and values of a dictionary, to a function:

```python
# arguments.unpack.dict.py
```

```python
def func(a, b, c):
    print(a, b, c)

values = {'b': 1, 'c': 2, 'a': 42}
func(**values)  # equivalent to func(b=1, c=2, a=42)
```

Combining argument types

We've already seen that positional and keyword arguments can be used together, as long as they are passed in the proper order. As it turns out, we can also combine unpacking (of both kinds) with normal positional and keyword arguments. We can even unpack multiple iterables and multiple dictionaries!

Arguments must be passed in the following order:

- First, positional arguments: both ordinary (`name`) and iterable unpacking (`*name`)

- Next come keyword arguments (`name=value`), which can be mixed with iterable unpacking (`*name`)

- Finally, there is dictionary unpacking (`**name`), which can be mixed with keyword arguments (`name=value`)

This will be much easier to understand with an example:

```python
# arguments.combined.py
def func(a, b, c, d, e, f):
    print(a, b, c, d, e, f)

func(1, *(2, 3), f=6, *(4, 5))
func(*(1, 2), e=5, *(3, 4), f=6)
func(1, **{'b': 2, 'c': 3}, d=4, **{'e': 5, 'f': 6})
func(c=3, *(1, 2), **{'d': 4}, e=5, **{'f': 6})
```

All the calls to `func()` above are equivalent. Play around with this example until you are sure you understand it. Pay close attention to the errors you get when you get the order wrong.

> The ability to unpack multiple iterables and dictionaries was introduced to Python by PEP 448. This PEP also introduced the ability to use unpacking in contexts other than just function calls. You can read all about it at `https://www.python.org/dev/peps/pep-0448/`.

When combining positional and keyword arguments, it is important to remember that each parameter can only appear once in the argument list:

```
# arguments.multiple.value.py
def func(a, b, c):
    print(a, b, c)

func(2, 3, a=1)
```

Here, we are passing two values for parameter a: the positional argument 2 and the keyword argument a=1. This is illegal, so we get an error when we try to run it:

```
$ python arguments.multiple.value.py
Traceback (most recent call last):
  File "arguments.multiple.value.py", line 5, in <module>
    func(2, 3, a=1)
TypeError: func() got multiple values for argument 'a'
```

Defining parameters

Function parameters can be classified into five groups.

- Positional or keyword parameters: allow both positional and keyword arguments
- Variable positional parameters: collect an arbitrary number of positional arguments in a tuple
- Variable keyword parameters: collect an arbitrary number of keyword arguments in a dictionary
- Positional-only parameters: can only be passed as positional arguments
- Keyword-only parameters: can only be passed as keyword arguments

All the parameters in the examples we have seen so far in this chapter are normal positional or keyword parameters. We've seen how they can be passed as both positional and keyword arguments. There's not much more to say about them, so let's look at the other categories. Before we do though, let's briefly look at optional parameters.

Optional parameters

Apart from the categories we've looked at here, parameters can also be classified as either *required* or *optional*. **Optional parameters** have a default value specified in the function definition. The syntax is `name=value`:

```python
# parameters.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
func(42, 43, 44)       # prints: 42, 43, 44
```

Here, `a` is required, while `b` has the default value `4` and `c` has the default value `88`. It's important to note that, with the exception of keyword-only parameters, required parameters must always be to the left of all optional parameters in the function definition. Try removing the default value from `c` in the above example and see for yourself what happens.

Variable positional parameters

Sometimes you may prefer not to specify the exact number of positional parameters to a function; Python provides you with the ability to do this by using **variable positional parameters**. Let's look at a very common use case, the `minimum()` function. This is a function that calculates the minimum of its input values:

```python
# parameters.variable.positional.py
def minimum(*n):
    # print(type(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 define a parameter with an asterisk, *, prepended to its name, we are telling Python that this parameter will collect a variable number of positional arguments when the function is called. Within the function, n is a tuple. Uncomment print(type(n)) to see for yourself, and play around with it for a bit.

Note that a function can have at most one variable positional parameter—it wouldn't make sense to have more. Python would have no way of deciding how to divide up the arguments between them. You are also unable to specify a default value for a variable positional parameter. The default value is always an empty tuple.

> Have you noticed how we checked whether n wasn't empty with a simple if n:? This is because collection objects evaluate to True when non-empty, and otherwise False, in Python. This is the case 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 parameters, instead of silently doing nothing. In this context, we're not concerned about making this function robust, but rather understanding variable positional parameters.

Did you notice that the syntax for defining variable positional parameters looks very much like the syntax for iterable unpacking? This is no coincidence. After all, the two features mirror each other. They are also frequently used together, since variable positional parameters save you from worrying whether the length of the iterable you're unpacking matches the number of parameters in the function definition.

Variable keyword parameters

Variable keyword parameters are very similar to variable positional parameters. The only difference is the syntax (** instead of *) and the fact that they are collected in a dictionary:

```python
# parameters.variable.keyword.py
def func(**kwargs):
    print(kwargs)

func(a=1, b=42)  # prints {'a': 1, 'b': 42}
func()  # prints {}
func(a=1, b=46, c=99)  # prints {'a': 1, 'b': 46, 'c': 99}
```

You can see that adding ** in front of the parameter name in the function definition tells Python to use that name to collect a variable number of keyword parameters. As in the case of variable positional parameters, each function can have at most one variable keyword parameter—and you cannot specify a default value.

Just like variable positional parameters resemble iterable unpacking, variable keyword parameters resemble dictionary unpacking. Dictionary unpacking is also often used to pass arguments to functions with variable keyword parameters.

The reason why being able to pass a variable number of keyword arguments 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 to the function the appropriate parameters. Before you read on, try to spend a couple of minutes figuring out a solution by yourself:

```python
# parameters.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 that, in the function, we can prepare a dictionary of connection parameters (conn_params) using default values as fallbacks, 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 right now. Running the preceding code yields the following result:

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

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

Positional-only parameters

Starting from Python 3.8, PEP 570 (`https://www.python.org/dev/peps/pep-0570/`) introduced **positional-only parameters**. There is a new function parameter syntax, `/`, indicating that a set of the function parameters must be specified positionally and *cannot* be passed as keyword arguments. Let's see a simple example:

```python
# parameters.positional.only.py
def func(a, b, /, c):
    print(a, b, c)

func(1, 2, 3)  # prints: 1 2 3
func(1, 2, c=3)  # prints 1 2 3
```

In the preceding example, we define a function `func()`, which specifies three parameters: `a`, `b`, and `c`. The `/` in the function signature indicates that `a` and `b` must be passed positionally, that is, not by keyword.

The last two lines in the example show that we can call the function passing all three arguments positionally, or we can pass `c` by keyword. Both cases work fine, as `c` is defined after the `/` in the function signature. If we try to call the function by passing `a` or `b` by keyword, like so:

```python
func(1, b=2, c=3)
```

This produces the following traceback:

```
Traceback (most recent call last):
  File "arguments.positional.only.py", line 7, in <module>
    func(1, b=2, c=3)
TypeError: func() got some positional-only arguments
passed as keyword arguments: 'b'
```

The preceding example show us that Python is now complaining about how we called `func()`. We have passed `b` by keyword, but we are not allowed to do that.

Positional-only parameters can also be optional:

```python
# parameters.positional.only.optional.py
def func(a, b=2, /):
    print(a, b)
```

```
func(4, 5)  # prints 4 5
func(3)  # prints 3 2
```

Let's see what this feature brings to the language with a few examples borrowed from the official documentation. One advantage is the ability to fully emulate behaviors of existing C-coded functions:

```
def divmod(a, b, /):
    "Emulate the built in divmod() function"
    return (a // b, a % b)
```

Another important use case is to preclude keyword arguments when the parameter name is not helpful:

```
len(obj='hello')
```

In the preceding example, the obj keyword argument impairs readability. Moreover, if we wish to refactor the internals of the len function, and rename obj to the_object (or any other name), the change is guaranteed not to break any client code, because there won't be any call to the len() function involving the now stale obj parameter name.

Finally, using positional-only parameters implies that whatever is on the left of / remains available for use in variable keyword arguments, as shown by the following example:

```
def func_name(name, /, **kwargs):
    print(name)
    print(kwargs)

func_name('Positional-only name', name='Name in **kwargs')

# Prints:
# Positional-only name
# {'name': 'Name in **kwargs'}
```

The ability to retain parameter names in function signatures for use in **kwargs can lead to simpler and cleaner code.

Let us now explore the mirror version of positional-only: keyword-only parameters.

Keyword-only parameters

Python 3 introduced **keyword-only parameters**. 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 parameters, or after a bare *. Let's see an example of both:

```python
# parameters.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 parameters (a) and a keyword-only one, 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 one, c. You can uncomment the third call to see the error that is produced.

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 different parameter types in the same function (in fact it is often very useful to do so). As in the case of combining different types of arguments in the same function call, there are some restrictions on ordering:

- Positional-only parameters come first, followed by a /.
- Normal parameters go after any positional-only parameters.

- Variable positional parameters go after normal parameters.
- Keyword-only parameters go after variable positional parameters.
- Variable keyword parameters always go last.
- For positional-only and normal parameters, any required parameters have to be defined before any optional parameters. This means that if you have an optional positional-only parameter, all your normal parameters must be optional too. This rule doesn't affect keyword-only parameters.

These rules can be a bit tricky to understand without an example, so let's look at a couple:

```python
# parameters.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')
```

Note the order of the parameters in the function definition. The execution of this yields the following:

```
$ python parameters.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 parameters:

```python
# parameters.all.pkwonly.py
def allparams(a, /, b,  c=42, *args, d=256, e, **kwargs):
    print('a, b, c:', a, b, c)
    print('d, e:', d, e)
    print('args:', args)
    print('kwargs:', kwargs)

allparams(1, 2, 3, 4, 5, 6, e=7, f=9, g=10)
```

Note that we have both positional-only and keyword-only parameters in the function declaration: a is positional-only, while d and e are keyword-only. They come after the *args variable positional argument, and it would be the same if they came right after a single * (in which case there wouldn't be any variable positional parameter). The execution of this yields the following:

```
$ python parameters.all.pkwonly.py
a, b, c: 1 2 3
d, e: 256 7
args: (4, 5, 6)
kwargs: {'f': 9, 'g': 10}
```

One other thing to note is the names we gave to the variable positional and keyword parameters. You're free to choose differently, but be aware that `args` and `kwargs` are the conventional names given to these parameters, at least generically.

More signature examples

To briefly recap on function signatures that use the positional- and keyword-only specifiers, here are some further examples. Omitting the variable positional and keyword parameters, for brevity, we are left with the following syntax:

```
def func_name(positional_only_parameters, /,
    positional_or_keyword_parameters, *,
    keyword_only_parameters):
```

First, we have positional-only, then positional or keyword parameters, and finally keyword-only ones.

Some other valid signatures are presented below:

```
def func_name(p1, p2, /, p_or_kw, *, kw):
def func_name(p1, p2=None, /, p_or_kw=None, *, kw):
def func_name(p1, p2=None, /, *, kw):
def func_name(p1, p2=None, /):
def func_name(p1, p2, /, p_or_kw):
def func_name(p1, p2, /):
```

All of the above are valid signatures, while the following would be invalid:

```
def func_name(p1, p2=None, /, p_or_kw, *, kw):
def func_name(p1=None, p2, /, p_or_kw=None, *, kw):
def func_name(p1=None, p2, /):
```

You can read about the grammar specifications in the official documentation:

https://docs.python.org/3/reference/compound_stmts.html#function-definitions

A useful exercise for you at this point would be to implement any of the above example signatures, printing out the values of those parameters, like we have done in previous exercises, and play around passing arguments in different ways.

Avoid the trap! Mutable defaults

One thing to be aware of, in Python, is that default values are created at definition 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:

```python
# parameters.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()
```

Both parameters 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 parameters.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—when using **memoization** techniques, for example. Even more interesting is what happens when, between the calls, we introduce one that doesn't use defaults, such as this:

```python
# parameters.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 parameters.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:

```python
# parameters.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, we always get a brand new, empty list.

After a thorough exposition of input parameters, it's now time to look at the other side of the coin, output parameters.

Return values

The return values of functions are one of those things where Python is ahead of the competition. In most other languages, functions are usually allowed to return only one object but, in Python, you can return a tuple—which implies that you can return whatever you want. This feature allows a programmer to write software that would be much harder to write in other languages, 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, or we invoke a bare `return` statement, the function will return `None`. This behavior is harmless when it's not needed, but allows for interesting patterns, and confirms Python as a very consistent language.

We say it's harmless because you are never forced to collect the result of a function call. We'll show you what we mean with an example:

```python
# 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 composed only of the `pass` statement. As the official documentation tells us, `pass` is a null operation, as, 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 brackets ({}), 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 to `func()` returns a value (`None`) that we don't collect. As we mentioned before, collecting the return value of a function call is not mandatory.

That's all well, but not very interesting, so how about we write an interesting function? Remember that, in *Chapter 1*, *A Gentle Introduction to Python*, we talked about the *factorial* function. Let's write our own implementation here (for simplicity, we will assume the function is always called correctly with appropriate values, so we won't need to sanity-check the input argument):

```python
# 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, we return 1. Otherwise, we perform the required calculation and return `result`.

> In Python it's common to use the `in` operator to do a membership check, as we did in the preceding example, instead of the more verbose:
>
> ```
> if n == 0 or n == 1:
>
>     ...
> ```

Let's now try to write this function a little bit more succinctly:

```python
# 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
```

This simple example shows how Python is both elegant and concise. This implementation is readable even if we have never seen `reduce()` or `mul()`. If you can't read or understand it, set aside a few minutes and do some research in 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 that every developer needs to be able to perform, so take this as a challenge.

> To this end, make sure you look up the `help()` function, which proves quite helpful when exploring with the console.

Returning multiple values

As we mentioned before, 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 boosting your ability to be creative.

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)
```

We could have wrapped the part that is in bold in the preceding code in brackets, 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.

> In the source code for this example, we have left a simple example of a test function to make sure the code is doing the correct calculation.

A few useful tips

When writing functions, it's very useful to follow guidelines so that you write them well. We'll quickly point some of them out.

Functions should do one thing

Functions that do one thing are easy to describe in one short sentence; functions that do multiple things can be split into smaller functions that do one thing. These smaller functions are usually easier to read and understand.

Functions should be small

The smaller they are, the easier it is to test and write them so that they do one thing.

The fewer input parameters, the better

Functions that take a lot of parameters quickly become hard to manage (among other issues).

Functions should be consistent in their return values

Returning `False` and returning `None` are 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 that return in a consistent way, no matter what happens in their logic.

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 we'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 right 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 will write better functions, which will serve you well.

> Chapter 3 of *Clean Code*, by Robert C. Martin, is dedicated to functions, and it's one of the best sets of guidelines we have 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.

The body of a recursive function usually has two sections: one where the return value depends on a subsequent call to itself, and one where it doesn't (called the **base case**).

As an example, we can consider the (hopefully now familiar) **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 this:

```
5! = 1 * 2 * 3 * 4 * 5 = (1 * 2 * 3 * 4) * 5 = 4! * 5
```

Let's write this down in code:

```python
# recursive.factorial.py
def factorial(n):
    if n in (0, 1):  # base case
        return 1
    return factorial(n - 1) * n  # recursive case
```

Recursive functions are often used when writing algorithms, and they can be really fun to write. As an exercise, try to solve a couple of simple problems using both a recursive and an iterative approach. Good candidates for practice might be calculating Fibonacci numbers, or the length of a string—things like that.

> When writing recursive functions, always consider how many nested calls you make, since there is a limit. For further information on this, check out `sys.getrecursionlimit()` and `sys.setrecursionlimit()`.

Anonymous functions

One last type of function that we 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 we wanted a list of all the numbers up to a certain value of *N* that are also multiples of five. We could use the `filter()` function for this, which will require a function and an iterable as input. The return value is a filter object that, when you iterate over it, yields the elements from the input iterable for which the function returns `True`. Without using an anonymous function, we might do something like this:

```python
# 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)))
```

Note how we use `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)))
```

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 both 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 we 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.

Function attributes

Every function is a fully fledged object and, as such, it has 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 a part of them and how to display their value for an example function:

```python
# func.attributes.py
def multiplication(a, b=1):
    """Return a multiplied by b. """
    return a * b


if __name__ == "__main__":
    special_attributes = [
        "__doc__", "__name__", "__qualname__", "__module__",
        "__defaults__", "__code__", "__globals__", "__dict__",
        "__closure__", "__annotations__", "__kwdefaults__",
    ]

    for attribute in special_attributes:
        print(attribute, '->', getattr(multiplication, attribute))
```

We 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 dynamically get an attribute at runtime, taking the name of the attribute from a variable (as in this example). 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 0x10fb599d0,
            file "func.attributes.py", line 2>
__globals__ -> {... omitted ...}
__dict__ -> {}
__closure__ -> None
__annotations__ -> {}
__kwdefaults__ -> None
```

We have omitted the value of the `__globals__` attribute, as it was too big. An explanation of the meaning of this attribute can be found in the *Callable types* section of the *Python Data Model* documentation page:

`https://docs.python.org/3/reference/datamodel.html#the-standard-type-hierarchy`

Should you want to see all the attributes of an object, just call `dir(object_name)` and you will be given a list of all of its attributes.

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, we don't have the room to go through all of them here. We've already seen some of them, 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 about at least once. Get familiar with them, experiment, write a small piece of code for each of them, and make sure you have them at your fingertips so that you can use them when needed.

You can find a list of built-in functions in the official documentation, here: `https://docs.python.org/3/library/functions.html`.

Documenting your code

We are big fans of code that doesn't need documentation. When we program correctly, choose the right names, and take care of the details, the code should come out as self-explanatory, with documentation being almost unnecessary. Sometimes a comment is very useful though, and so is some documentation. You can find the guidelines for documenting Python in PEP 257 -- Docstring conventions:

`https://www.python.org/dev/peps/pep-0257/`, but we'll show you the basics here.

Python is documented with strings, which are aptly called **docstrings**. Any object can be documented, and we can use either one-line or multi-line docstrings. One-liners are very simple. They should not provide another signature for the function, but instead state its purpose:

```python
# 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.

Multiline 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, we have documented a fictitious `connect()` function,
using the **Sphinx** notation, in the following example:

```
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
```

> Sphinx is one of the most widely used tools for creating Python
> documentation—in fact, the official Python documentation was
> written with it. It's definitely worth spending some time checking
> it out.

The `help()` built-in function, which is intended for interactive use, creates
a documentation page for an object using its docstring.

Importing objects

Now that we know a lot about functions, let's look at how to use them. The whole point of writing functions is to be able to reuse them later, and in Python, this translates to importing them into the namespace where you need them. There are many different ways to import objects into a namespace, but the most common ones are `import module_name` and `from module_name import function_name`. Of course, these are quite simplistic examples, but bear with us for the time being.

The `import module_name` form finds the `module_name` module and defines a name for it in the local namespace, where the `import` statement is executed. The `from module_name import identifier` form 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:

```
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 one of Fabrizio's projects (notice that the blank lines between blocks of imports follow the guidelines from PEP 8 at https://www.python.org/dev/peps/pep-0008/#imports: standard library, third party, and local code):

```python
# imports.py
from datetime import datetime, timezone  # two imports on the same line
from unittest.mock import patch  # single import

import pytest  # third party library

from core.models import (  # multiline import
    Exam,
    Exercise,
    Solution,
)
```

When we have a structure of files starting in the root of our project, we can use the dot notation to get to the object we want to import into our 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; this is frowned upon for several reasons though, relating to performance and the risk of silently shadowing other names. You can read all that there is to know about imports in the official Python documentation but, before we leave the subject, let us give you a better example.

Imagine that we have defined a couple of functions, `square(n)` and `cube(n)`, in a module, `funcdef.py`, which is in the `lib` folder. We want to use them in a couple of modules that are at the same level as 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
│   ├── __init__.py
│   └── funcdef.py
```

Before we 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 an `__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:

```python
# lib/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 type of import we've seen so far is called an **absolute import**; that is, it defines the whole path of either 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 a **relative import**. 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 such as this:

```
from .mymodule import myfunc
```

Relative imports are quite useful when restructuring projects. Not having the full path in the imports allows the developer to move things around without having to rename too many of those paths.

For a complete explanation of relative imports, refer to PEP 328:

`https://www.python.org/dev/peps/pep-0328/`

In later chapters, we will create projects using different libraries and use several different types of imports, including relative ones, so make sure you take a bit of time to read up about them in the official Python documentation.

One final example

Before we finish off this chapter, let's go through one last example. We could write a function to generate a list of prime numbers up to a limit; we've already seen the code for this in *Chapter 3*, so let's make it a function and, to keep it interesting, let's optimize it a bit.

It turns out that we don't need to divide by all numbers from *2* to *N-1* to decide whether a number, *N*, is prime. We can stop at $\sqrt{N}$ (the square root of *N*). Moreover, we don't need to test the division for all numbers from *2* to $\sqrt{N}$, as we can just use the primes in that range. We leave it up to you to figure out why this works, if you're interested in the beauty of mathematics.

Let's see how the code changes:

```python
# 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 = 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 that 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 `primelist` result list to get the primes for the division and 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 also served our purpose, the formula we chose is cleaner and requires a couple of imports, which is what we wanted to show. Check out the functions in the `math` module—they are very interesting!

Summary

In this chapter, we explored the world of functions. They are very important and, from now on, we'll use them in virtually everything we do. 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 inputs and may produce outputs. We can feed input arguments to a function in many different ways, using positional and keyword arguments, and using variable syntax for both types.

You should now know how to write a function, document it, import it into your code, and call it.

In the next chapter we will be picking up the pace a little a bit, so we suggest you take any opportunity you get to consolidate and enrich the knowledge you've gathered so far by putting your nose into the Python official documentation.

5
Comprehensions
and Generators

"It's not the daily increase but daily decrease. Hack away at the unessential."

– Bruce Lee

We love this quote from Bruce Lee. He was such a wise man! The second part in particular, "hack away at the unessential," is to us 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 wouldn't we?

Sometimes, there are valid reasons for not pushing our code up to the maximum limit: for example, sometimes, in order to achieve a negligible improvement, we have to sacrifice 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 reasonable to try to 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 seconds saved over thousands 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 enable you to write efficient, elegant code that reads well, runs fast, and doesn't waste resources in an obvious way.

In this chapter, we are going to cover the following:

- The `map()`, `zip()`, and `filter()` functions
- Comprehensions
- Generators

We will perform several measurements and comparisons and cautiously draw some conclusions. Please do keep in mind that on a different machine with a different setup or operating system, results may vary. Take a look at this code:

```python
# 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 that we 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 such as 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, it's likely to be a sporadic operation. You don't need to concern yourself with saving a fraction of a microsecond 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 one-tenth of a second on a function called one million times saves you 100,000 seconds, which is about 27.7 hours—that makes a big difference! 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 the iterator and iterable. Simply put, this is the ability of 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 *Chapter 6, OOP, Decorators, and Iterators*.

Some of the objects we're going to explore in this chapter are iterators, which save memory by only operating on a single element of a collection at a time rather than creating a modified copy. As a result, some extra work is needed if we just want to show the result of the operation. We will often resort to wrapping the iterator in a `list()` constructor. This is because passing an iterator to `list(...)` exhausts it and puts all the generated items in a newly created list, which we can easily print to show you its content. Let's see an example of using the technique on a `range` object:

```
# list.iterable.py
>>> range(7)
range(0, 7)
>>> list(range(7))  # put all elements in a list to view them
[0, 1, 2, 3, 4, 5, 6]
```

We've highlighted the result of typing `range(7)` into a Python console. Notice that it doesn't show the contents of the `range`, because `range` never actually loads the entire sequence of numbers into memory. The second highlighted line shows how wrapping the `range` in a `list()` allows us to see the numbers it generated.

The map, zip, and filter functions

We'll start by reviewing `map()`, `filter()`, and `zip()`, which are the main built-in functions you can employ when handling collections, and then we'll learn how to achieve the same results using two very important constructs: **comprehensions** and **generators**.

map

According to the official Python documentation:

> *map(function, iterable, …)*
>
> *Return 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:

```
# map.example.py
>>> map(lambda *a: a, range(3))  # 1 iterable
<map object at 0x10acf8f98>  # Not useful! Let's use list
```

```
>>> list(map(lambda *a: a, range(3)))  # 1 iterable
[(0,), (1,), (2,)]
>>> 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 we have to wrap calls in list(...). Without it, we 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, nor does it 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 in older Python versions, when sorting did not support the use of *key functions*. Nowadays, it is not needed as often, but it's a cool trick that still comes in handy 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 as 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. In this context, 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. This technique has nothing to do with Python decorators, which we will explore later on in the book.

After sorting, we revert the decorated objects to get the original ones from them. This is referred to as **undecorating**.

```python
# 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))
```

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 `credits`. The value of `credits` is also a dictionary in which there are three subject/grade key/value pairs. As you may recall from our visit to the data structures world, calling `dict.values()` returns an iterable object, with only the dictionary's values. Therefore, `sum(student['credits'].values())` for the first student is equivalent to `sum((9, 6, 7))`.

Let's print the result of calling `decorate` with the first student:

```
>>> decorate(students[0])
(22, {'credits': {'history': 7, 'math': 9, 'physics': 6}, 'id': 0})
```

If we decorate all the students like this, we can sort them on their total number of credits by just sorting the list of tuples. To apply the decoration to each item in `students`, we call `map(decorate, students)`. 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}]
```

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/3.9/howto/sorting.html#the-old-way-using-decorate-sort-undecorate`

One thing to notice about the sorting part is what happens when two or more students share the same total sum. The sorting algorithm would then proceed to sort 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 three-tuple instead of a two-tuple, having the sum of credits in the first position, the position of the `student` object in the `students` list in second place, and the `student` object itself in third place. 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.

zip

We've already covered `zip()` in the previous chapters, so let's just define it properly, after which we 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:

```python
# zip.grades.py
>>> grades = [18, 23, 30, 27]
>>> avgs = [22, 21, 29, 24]
>>> list(zip(avgs, grades))
[(22, 18), (21, 23), (29, 30), (24, 27)]
>>> list(map(lambda *a: a, avgs, grades))  # equivalent to zip
[(22, 18), (21, 23), (29, 30), (24, 27)]
```

Here, we're zipping together the average and the grade for the last exam for each student. Notice how easy it is to reproduce `zip()` using `map()` (the last two instructions of the example). Here as well, in order to visualize the results, we have to use `list()`.

A simple example of the combined use of `map()` and `zip()` could be a way of calculating the element-wise maximum among sequences; that is, the maximum of the first element of each sequence, then the maximum of the second one, and so on:

```python
# 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 maximum values of three sequences. `zip()` is not strictly needed of course—we could just use `map()`. Sometimes it's hard, when showing a simple example, to grasp why using a technique might be good or bad. We forget that we aren't always in control of the source code; we might have to use a third-party library that we can't change the way we want. Having different ways to work with data is therefore really 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:

```python
# 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]
```

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 manipulate sequences very effectively. But these functions are not the only way to do it. Let's look at one of the nicest features of Python: comprehensions.

Comprehensions

A comprehension is a concise notation for performing some operation on each element of a collection of objects, and/or selecting a subset of elements that satisfy some condition. They are borrowed from the functional programming language Haskell (https://www.haskell.org/) and, together with iterators and generators, contribute to giving Python a functional flavor.

Python offers different types of comprehensions: list, dictionary, and set. We'll concentrate mainly on list comprehensions; once you understand those, the other types will be quite easy to grasp.

Let's start with a very simple example. We 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:

```python
# squares.map.py
# If you code like this you are not a Python dev! ;)
```

```
>>> squares = []
>>> for n in range(10):
...        squares.append(n ** 2)
...
>>> 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. Now, 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 placed a `for` loop within square brackets. Let's now filter out the odd squares. We'll show you how to do it with `map()` and `filter()` first, before then using a list comprehension again:

```
# even.squares.py
# using map and filter
sq1 = list(
    map(lambda n: n ** 2, filter(lambda n: not 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
```

We think that the difference in readability is now evident. The list comprehension reads much better. It's almost English: give us 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 does, too, 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:

```python
# pairs.for.loop.py
items = 'ABCD'
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:

```
$ python pairs.for.loop.py
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'B'), ('B',
'C'), ('B', 'D'), ('C', 'C'), ('C', 'D'), ('D', 'D')]
```

All the tuples with the same letter are those where `b` is at the same position as `a`. Now, let's see how we can translate this to a list comprehension:

```python
# pairs.list.comprehension.py
items = 'ABCD'
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.

> Another way of achieving the same result is to use the `combinations_with_replacement()` function from the `itertools` module (which we briefly introduced in *Chapter 3, Conditionals and Iteration*). You can read more about it in the official Python documentation.

Filtering a comprehension

We can also apply filtering to a comprehension. Let's first do it with `filter()`, and 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 similar to the one we saw in the previous example:

```python
# pythagorean.triple.py
from math import sqrt
# this will generate all possible pairs
mx = 10
triples = [(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
triples = list(
    filter(lambda triple: triple[2].is_integer(), triples))

print(triples)  # prints: [(3, 4, 5.0), (6, 8, 10.0)]
```

> A **Pythagorean triple** is a triple (a, b, c) of integer numbers satisfying the equation $a^2 + b^2 = c^2$.

In the preceding code, we generated a list of *three-tuples*, `triples`. 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` is 3 and `b` is 4, the tuple will be `(3, 4, 5.0)`, and when `a` is 5 and `b` is 7, the tuple will be `(5, 7, 8.602325267042627)`.

After generating all the `triples`, we need to filter out all those where the hypotenuse is not 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 we just showed you, the one with hypotenuse `5.0` will be retained, while the one with the `8.602325267042627` hypotenuse will be discarded.

This is good, but we don't like the fact that the triple has two integer numbers and a float—they are all supposed to be integers. Let's use `map()` to fix this:

```python
# pythagorean.triple.int.py
from math import sqrt
mx = 10
triples = [(a, b, sqrt(a**2 + b**2))
```

```
        for a in range(1, mx) for b in range(a, mx)]
 triples = filter(lambda triple: triple[2].is_integer(), triples)
 # this will make the third number in the tuples integer
 triples = list(
     map(lambda triple: triple[:2] + (int(triple[2]), ), triples))

 print(triples)  # prints: [(3, 4, 5), (6, 8, 10)]
```

Notice the step we added. We take each element in `triples` and we slice it, taking only the first two elements in it. Then, we concatenate the slice with a one-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
 triples = [(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
 triples = [(a, b, int(c)) for a, b, c in triples if c.is_integer()]
 print(triples)  # prints: [(3, 4, 5), (6, 8, 10)]
```

That's much better! It's clean, readable, and shorter. It's not quite as elegant as it could have been, though. We're still wasting memory by constructing a list with a lot of triples that we end up discarding. We can fix that by combining the two comprehensions into one:

```
 # pythagorean.triple.walrus.py
 from math import sqrt
 # this step is the same as before
 mx = 10
 # We can combine generating and filtering in one comprehension
 triples = [(a, b, int(c))
     for a in range(1, mx) for b in range(a, mx)
     if (c := sqrt(a**2 + b**2)).is_integer()]
 print(triples)  # prints: [(3, 4, 5), (6, 8, 10)]
```

Now that really is elegant. By generating the triples and filtering them in the same list comprehension, we avoid keeping any triple that doesn't pass the test in memory. Notice that we used an **assignment expression** to avoid needing to compute the value of `sqrt(a**2 + b**2)` twice.

> We're going quite fast here, as anticipated in the *Summary* of *Chapter 4, Functions, the Building Blocks of Code.* Are you playing with this code? If not, we suggest you do. It's very important that you play around, break things, change things, and see what happens. Make sure you have a clear understanding of what is going on.

Dictionary comprehensions

Dictionary comprehensions work exactly like list comprehensions, but to construct dictionaries. There is only a slight difference in the syntax. The following example will suffice to explain everything you need to know:

```python
# dictionary.comprehensions.py
from string import ascii_lowercase
lettermap = {c: k for k, c in enumerate(ascii_lowercase, 1)}
```

If you print `lettermap`, you will see the following (we omitted the intermediate results, but you get the gist):

```
$ python dictionary.comprehensions.py
{'a': 1,
 'b': 2,
 ...
 'y': 25,
 'z': 26}
```

In the preceding code, we are enumerating the sequence of all lowercase ASCII letters (using the `enumerate` function). We then construct a dictionary with the resulting letter/number pairs as keys and values. Notice how the syntax is similar to the familiar dictionary syntax.

There is also another way to do the same thing:

```python
lettermap = dict((c, k) for k, c in enumerate(ascii_lowercase, 1))
```

In this case, we are feeding a generator expression (we'll talk more about these later in this chapter) to the `dict` constructor.

Dictionaries do not allow duplicate keys, as shown in the following example:

```python
# dictionary.comprehensions.duplicates.py
word = 'Hello'
swaps = {c: c.swapcase() for c in word}
print(swaps)  # prints: {'H': 'h', 'e': 'E', 'l': 'L', 'o': 'O'}
```

We create a dictionary with the letters of the string `'Hello'` as keys and the same letters, but with the case swapped, as values. Notice that there is only one `'l': 'L'` pair. The constructor doesn't complain; it simply reassigns duplicates to the last value. Let's make this clearer with another example that assigns 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: {'H': 0, 'e': 1, 'l': 3, 'o': 4}
```

Notice the value associated with the letter `'l': 3`. The `'l': 2` pair isn't there; it has been overridden by `'l': 3`.

Set comprehensions

Set comprehensions are very similar to list and dictionary ones. Let's see one quick example:

```
# set.comprehensions.py
word = 'Hello'
letters1 = {c for c in word}
letters2 = set(c for c in word)
print(letters1)  # prints: {'H', 'o', 'e', 'l'}
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 `letters1` is very similar to that of a dictionary comprehension. You can spot the difference only by the fact that dictionaries require keys and values, separated by colons, while sets don't. For `letters2`, we fed a generator expression to the `set()` constructor.

Generators

Generators are very powerful tools. They are based on the concept 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 behave like regular functions in all respects, except for one difference: instead of collecting results and returning them at once, they are automatically turned into iterators that yield results one at a time when you call `next` on them. Generator functions are automatically turned into their own iterators by Python.

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 we asked you to count out loud from 1 to 1,000,000. You start, and at some point, we ask you to stop. After some time, we 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 we stopped you after 31,415, you will just go on with 31,416, and so on.

The point is that you don't need to remember all the numbers you said before 31,415, 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:

```python
# 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 two `print` statements 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 `print` statements return the same result is because we fed `get_squares_gen()` to the `list()` constructor, which exhausts the generator completely by asking for the next element until a `StopIteration` is raised. Let's see this in detail:

```python
# 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 0x10dd...>
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))
```

Each time we call `next` on the generator object, we either start it (the 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), 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. 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 into every iteration aspect of Python makes generators even more powerful because once we've written them, we'll be able to plug them into whatever iteration mechanism we want.

At this point, you're probably asking yourself why you would want to use a generator instead of a regular function. The answer is to save time and (especially) memory.

We'll talk more about performance later, but for now, let's concentrate on one 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 a length of N, then the number of its permutations is $N!$. This means that if the sequence is 10 elements long, the number of permutations is 3,628,800. But a sequence of 20 elements would have 2,432,902,008,176,640,000 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 seconds, but for 20 elements there is simply no way that it could be done (it would take thousands of years and require billions of gigabytes of memory).

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 process them all—there are too many—but at least you'll be able to work with some of them. Sometimes 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 that which otherwise wouldn't be.

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 cause 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's consistency prevents this, and allows us great ease when coding. Let's see a quick example:

```python
# 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, we told you that generator objects are based on the
iteration protocol. We'll see in *Chapter 6, OOP, Decorators, and Iterators*, a complete
example of how to write a custom iterator/iterable object. For now, we 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. `__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.

> If you recall, 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:

```python
# 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 the same as the previous example, only this time instead of using the next(squares) proxy call, we're directly calling squares.__next__().

Generator objects also have three other methods that allow us to control their behavior: send(), throw(), and close(). send() allows us to communicate a value back to the generator object, while throw() and close(), respectively, allow us to raise an exception within the generator and close it. Their use is quite advanced and we won't be covering them here in detail, but we want to spend a few words on send(), with a simple example:

```
# 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, and 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. But what if you wanted to stop it at some point? One solution is to use a variable to control the while loop, as in something such as 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 following call to `next` will raise a `StopIteration` exception. This trick works, but we don't like it. The function depends 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 into 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 15, in <module>


    print(c.send('Q')) # F
StopIteration
```

We think it's worth going through this code line by line, as if we were executing it, to see whether 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 as 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, result is set to 'Wow!' (still #A), and then its type and value are printed on the console (#B). result is not 'Q', so 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, the yield n expression (#A) returns None (the behavior is exactly the same as when we call a function that does not return anything). result is therefore set to None, and its type and value are 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', and so 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, which 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 come back to this example later.

Using send() allows for interesting patterns, and it's worth noting that send() can also be used to start the execution of a generator (provided you call it with None).

The yield from expression

Another interesting construct is the `yield from` expression. This expression allows you to yield values from a sub-iterator. Its use allows for quite advanced patterns, so let's see a very quick example of it:

```python
# 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, and 16 on the console (on separate lines). By now, we 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:

```python
# 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, `yield from` is actually running a sub-iterator, `(n ** 2 ...)`. The `yield from` expression returns to the caller each value the sub-iterator is producing. It's shorter and reads better.

Generator expressions

Let's now talk about the other technique 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 brackets. 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 0x103fb5a98>
>>> 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 we 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. First, map():

```
# gen.map.py
def adder(*n):
    return sum(n)
s1 = sum(map(adder, 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 we find the generator expression to be much more readable. Now, for filter():

```
# 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 this example, `odd_cubes1` and `odd_cubes2` are the same: they generate a sequence of odd cubes. Yet again, we prefer the generator syntax. This should be evident when things get a little more complicated:

```python
# 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 two generators, `cubes1` and `cubes2`. They are exactly the same and return two-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, as we did in this example, the superiority of the generator syntax is evident. It's shorter, simpler, and more elegant.

Now, let us ask you: what is the difference between the following lines of code?

```python
# 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 brackets 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 one million times.

Do you see where we're losing 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 over these details that make our code very different. If you don't believe us, check out the following code:

```python
# sum.example.2.py
s = sum([n**2 for n in range(10**9)])  # this is killed
# s = sum(n**2 for n in range(10**9))    # this succeeds
print(s)  # prints: 333333332833333333500000000
```

Try running the example. If we run the first line on an old Linux machine with 6 GB RAM, this is what we get:

```
$ python sum.example.2.py
Killed
```

On the other hand, if we comment out the first line, and uncomment the second one, this is the result:

```
$ python sum.example.2.py
333333332833333333500000000
```

Sweet generator expressions. The difference between the two lines is that in the first one, a list with the squares of the first billion numbers must be made before being able to sum them up. That list is huge, and we ran out of memory (at least, Heinrich's machine did; if yours doesn't, try a bigger number), so Python kills the process for us.

But when we remove the square brackets, we no longer have a list. The `sum` function receives 0, 1, 4, 9, and so on until the last one, and sums them up. No problems.

Some performance considerations

So, we've seen that we have many different ways of achieving the same result. We can use any combination of `map()`, `zip()`, and `filter()`, or choose to go with a comprehension or a generator. We may even decide to go with `for` loops; when the logic to apply to each running parameter isn't simple, these may be the best option.

Besides readability concerns, though, let's also talk about performance. When it comes to performance, usually there are two factors that 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 whether a simple generator function would work instead.

If the answer is yes to the latter, go with the generator, as it will save a lot of space. The same goes for functions: if you don't actually need them to return a list or tuple, then you can transform them into 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 more complicated 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 on average, `map()` exhibits performance similar to comprehensions and generator expressions, while `for` loops are consistently slower.

In order to appreciate the reasoning behind these statements fully, we need to understand how Python works, which is a bit outside the scope of this book as it's quite technical in detail. Let's just say that `map()` and 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 CPython (`https://github.com/python/cpython`), which is written in C. C is one of the most powerful and popular programming languages still used today.

How about we do a small exercise and try to find out whether the claims we made are accurate? We will write a small piece of code that collects the results of `divmod(a, b)` for a certain set of integer pairs, `(a, b)`. We will use the `time()` function from the `time` module to calculate the elapsed time of the operations that we will perform:

```python
# performance.py
from time import time
mx = 5000

t = time()  # start time for the for loop
floop = []
for a in range(1, mx):
    for b in range(a, mx):
        floop.append(divmod(a, b))
print('for loop: {:.4f} s'.format(time() - t))  # elapsed time
```

```python
t = time()  # start time for the list comprehension
compr = [
    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
gener = list(
    divmod(a, b) for a in range(1, mx) for b in range(a, mx))
print('generator expression: {:.4f} s'.format(time() - t))
```

As you can see, we're creating three lists: `floop`, `compr`, and `gener`. Running the code produces the following:

```
$ python performance.py
for loop: 2.3652 s
list comprehension: 1.5173 s
generator expression: 1.5289 s
```

The list comprehension runs in ~64% of the time taken by the `for` loop. That's impressive. The generator expression came very close to that, with ~65%. The difference in time between the list comprehension and generator expression is hardly significant, and if you re-run the example a few times, you will probably also see the generator expression take less time than the list comprehension.

An interesting result is to notice that, within the body of the `for` loop, we're appending data to a list. This implies that Python does the work, behind the scenes, of resizing it every now and then, allocating space for items to be appended. We guessed that creating a list of zeros, and simply filling it with the results, might have sped up the `for` loop, but we were wrong. Check it for yourself; you just need `mx * (mx - 1) // 2` elements to be pre-allocated.

> The approach we used here for timing execution is rather naïve. In *Chapter 11*, *Debugging and Profiling*, we will look at better ways of profiling code and timing execution.

Let's see a similar example that compares a `for` loop and a `map()` call:

```python
# performance.map.py
from time import time
mx = 2 * 10 ** 7
```

```python
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))
```

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 `divmod()`, and we have only one loop instead of two nested ones. Execution gives the following result:

```
$ python performance.map.py
for loop: 2.3240 s
list comprehension: 1.0891 s
map: 0.5070 s
```

And `map` wins the race: it took ~47% of the time required by the list comprehension, and ~21% of the time needed by the `for` loop. Take these results with a pinch of salt, however, as the result might be different according to various factors, such as OS and Python version. But in general, we think it's safe to say that these results are good enough for having an idea when it comes to coding for performance.

Apart from the little case-by-case differences though, it's quite clear that the `for` loop option is the slowest one, so let's see why we still want to use it.

Don't overdo comprehensions and generators

We've seen how powerful comprehensions and generator expressions can be. And they are, don't get us wrong, but the feeling that we have when we 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 maintain or change.

If you check the Zen of Python again, there are a few lines that we think are worth keeping in mind when dealing with optimized code:

```
>>> import this
...
Explicit is better than implicit.
Simple is better than complex.
...
Readability counts.
...
If the implementation is hard to explain, it's a bad idea.
...
```

Comprehensions and generator expressions are more implicit than explicit, can be quite difficult to read and understand, and can be hard to explain. Sometimes, you have to break them apart using the inside-out technique to understand what's going on.

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 how to calculate them in the *Filtering a comprehension* section, but we did it in a very inefficient way. We were scanning all pairs of numbers below a certain threshold, calculating the hypotenuse, and filtering out those that were not valid Pythagorean triples.

A better way to get a list of Pythagorean triples is to generate them directly. There are many different formulas you can use to do this; here we will use the **Euclidean formula**. This formula says that any triple (a, b, c), where $a = m^2 - n^2$, $b = 2mn$ and $c = m^2 + n^2$, 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 $(6, 8, 10)$, which is like $(3, 4, 5)$, only all the numbers are 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. The 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 among their divisors, that is, when 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 $m \leq N^{1/2}$.

To recap: *m* must be greater than *n*, they must also be coprime, and their difference *m* - *n* must be odd. Moreover, to avoid useless calculations, we'll put the upper bound for *m* at *floor(sqrt(N)) + 1*.

> The `floor` function 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 *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 any numbers.

Let's put all of this into code, step by step. We start by writing a simple `gcd()` function that uses **Euclid's algorithm**:

```python
# 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 we won't spend any time talking about it here as we need to focus on the generator expression. The next step is to use the knowledge we gathered before to generate a list of primitive Pythagorean triples:

```python
# 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=sum                                # 7
)
```

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 creates 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. It's worth noting how we calculated *sqrt(N)*, that is, N**.5, which is just another way to do it that we 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.

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 our personal preference; there is no mathematical reason behind it.

So, what do you think? Was it straightforward to read? We don't think so. And believe us, this is still a simple example; we have both seen much worse in our careers. This kind of code is difficult to understand, debug, and modify. It should have no place in a professional environment.

Let's see whether we can rewrite this code into something more readable:

```python
# 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

sorted(gen_triples(50), key=sum)                   # 9
```

This is so much better. Let's go through it, line by line. You'll see how much 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 do we calculate a and b, and yield the resulting tuple. We could have calculated the values of a and b earlier, but by delaying until we know all conditions for a valid triple are satisfied, we avoid wasting time and CPU. On the last line, we apply sorting with the same key we were using in the generator expression example.

We hope you agree that this example is easier to understand. If we ever need to modify the code, this will be much easier, and less error-prone to work with than the generator expression.

If you print the results of both examples (they are the same), you will get this:

```
[(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 to use comprehensions and generator expressions as much as you can, but if the code starts to become complicated to modify or read, you may want to refactor it into something more readable. Your colleagues will thank you.

Name localization

Now that we are familiar with all types of comprehensions and generator expressions, let's talk about name localization within them. Python 3 localizes loop variables in all four forms of comprehensions: list, dictionary, set, and generator expressions. This behavior is therefore different from that of the for loop. Let's look at some simple examples to show all the cases:

```python
# 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 = {A: 2 * A for A in range(5)}
```

```
print(A)  # prints: 100

ex4 = {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 exercise list, dictionary, and set comprehensions and a generator expression. 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 in the same way with any other type of comprehension or a generator expression. 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
{'__name__': '__main__', '__doc__': None, ..., 's': 10, 'A': 4}
```

Together, with a lot of other boilerplate stuff that we have omitted, we can spot `'A': 4`.

Generation behavior in built-ins

Generator-like behavior is quite common among the built-in types and functions. This is a major difference between Python 2 and Python 3. In Python 2, functions such as `map()`, `zip()`, and `filter()` returned lists instead of iterable objects. 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 returned a list, and there was another function called `xrange()` that behaved like the `range()` function now behaves in Python 3.

The idea of functions and methods that return iterable objects is quite widespread. You can find it in the `open()` function, which is used to operate on file objects (we'll see it in *Chapter 8*, *Files and Data Persistence*), 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 to reduce the memory footprint by avoiding wasting space wherever possible, especially in those functions and methods that are used extensively in most situations. At the beginning of this chapter, we said that it makes more sense to optimize the performance 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. That is exactly what Python itself is doing here.

One last example

Before we finish this chapter, we'll show you a simple problem that Fabrizio used to submit to candidates for a Python developer role in a company he used to work for.

The problem is the following: write a function that returns the terms of the sequence *0 1 1 2 3 5 8 13 21 ...*, 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 for testing 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.

Let's start with a rudimentary version, and then improve on it:

```python
# 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 value in the sequence. 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 we 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, Fabrizio 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 to what you'll find here (an excellent candidate would have started with it!):

```python
# 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 he was given. We don't know why he kept it, but we're glad he did so we can show it to you. Now, the `fibonacci()` function is a *generator function*. First, we yield 0, and 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 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 wrap the call with `list()`, as usual. But what about elegance? We can't leave it like that, can we? Let's try the following:

```python
# 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, or 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.

Summary

In this chapter, we explored the concepts of iteration and generation a bit more deeply. We looked at the `map()`, `zip()`, and `filter()` functions in detail, and learned how to use them as an alternative to a regular `for` loop approach.

Then we covered the concept of comprehensions for lists, dictionaries, and sets. We explored their syntax and how to use them as an alternative to both the classic `for` loop approach and the use of the `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 so if we used a conventional approach based on lists.

We talked about performance, and saw that `for` loops come 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()`, and comprehensions, can be much faster.

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. It is structurally similar to this one, in that we won't explore many different subjects—just a few of them—but we'll try to delve deeper into them.

Make sure you understand the concepts of this chapter before moving on to the next one. We're building a wall brick by brick, and if the foundation is not solid, you won't get very far.

6

OOP, Decorators, and Iterators

La classe non è acqua. (Class will out.)

– Italian saying

We could probably write a whole book about **object-oriented programming** (**OOP**) and classes. In this chapter, we face the hard challenge of finding the balance between breadth and depth. There are simply too many things to talk about, and plenty of them would take more than this whole chapter if we described them in depth. Therefore, we will try to give you what we 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.

In this chapter, we are going to cover the following topics:

- Decorators
- OOP with Python
- Iterators

Decorators

In *Chapter 5, Comprehensions and Generators*, we measured the execution time of various expressions.

If you recall, we 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. We also printed it on the console after each measurement. That was tedious.

Every time we find ourselves repeating things, an alarm bell should go off. Can we 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: 0.3001396656036377

t = time()
g()
print('g took:', time() - t)   # g took: 0.5039339065551758
```

In the preceding code, we defined two functions, `f()` and `g()`, which do nothing but sleep (for 0.3 and 0.5 seconds, respectively). We used the `sleep()` function to suspend the execution for the desired amount of time. Notice how the time 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: 0.30434322357177734
measure(g)  # g took: 0.5048270225524902
```

Ah, much better now. The whole timing mechanism has been encapsulated in a function so we don't repeat code. We print the function name dynamically and it's easy enough to code. What if we needed to pass any 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: 0.30056095123291016
measure(f, 0.2)  # f took: 0.2033553123474121
```

Now, `f()` is expecting to be fed `sleep_time` (with a default value of 0.1), so we don't need `g()` anymore. We also had to change the `measure()` function so that it now accepts 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 `func()` we do inside.

This is very good, but we can push it a little bit further. Let's say we somehow want to have that timing behavior built into 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: 0.20372915267944336
f(sleep_time=0.3)  # f took: 0.30455899238586426
print(f.__name__)  # wrapper <- ouch!
```

The preceding code is probably not so straightforward. 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() (you can witness this in the last line of code). Since the wrapper() inside is calling func(), which is f(), we are actually closing the loop.

The wrapper() function is, not surprisingly, a wrapper. It takes variable positional and keyword arguments and calls f() with them. It also does the time measurement calculation around the call.

This technique is called **decoration**, and measure() is, effectively, a **decorator**. This paradigm became so popular and widely used that, in version 2.4, Python added a special syntax for it. You can read the specifics in PEP 318 (`https://www.python.org/dev/peps/pep-0318/`). In Python 3.9, the decorator syntax was slightly amended, to relax some grammar restrictions; this change was brought about in PEP 614 (`https://www.python.org/dev/peps/pep-0614/`).

Let's now explore three cases: one decorator, two decorators, and one decorator that takes arguments. First, the single decorator case:

```
# 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:

```python
# 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, it is important to pay attention to the order. 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 is to the function, the sooner it is applied.*

Some decorators can take arguments. This technique is generally used to produce another decorator (in which case, the object would be called a **decorator factory**). Let's look at the syntax, and then we'll see an example of it:

```python
# decorators/syntax.py
def func(arg1, arg2, ...):
    pass
func = decoarg(arg_a, arg_b)(func)

# is equivalent to the following:

@decoarg(arg_a, arg_b)
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 we give you another example, let's fix one thing that is bothering us. Take a look at this code from our previous example:

```python
# decorators/time.measure.deco1.py

def measure(func):
    def wrapper(*args, **kwargs):
        …
```

```
    return wrapper

f = measure(f)  # decoration point
print(f.__name__)  # wrapper <- ouch!
```

We don't want to lose the original function's name and docstring when we decorate it. But because inside our decorator we return `wrapper`, the decorated function, `f()`, is reassigned to it and therefore its original attributes are lost, replaced with the attributes of `wrapper`. There is an easy fix for that from the beautiful `functools` module. We will fix the last example, and we will also rewrite its syntax to use the `@` operator:

```python
# 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: 0.3010902404785156
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.

> For the full list of function attributes that are reassigned by `func()`, please check the official documentation for the `functools.update_wrapper()` function, here: `https://docs.python.org/3/library/functools.html?#functools.update_wrapper`.

Let's see another example. We want a decorator that prints an error message when the result of a function is greater than a certain threshold. We will also take this opportunity to show you how to apply two decorators at once:

```python
# decorators/two.decorators.py
from time import 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(
                f'Result is too big ({result}). '
                'Max allowed is 100.'
            )
        return result
    return wrapper

@measure
@max_result
def cube(n):
    return n ** 3

print(cube(2))
print(cube(5))
```

We 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.

We 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: 3.0994415283203125e-06
8

Result is too big (125). Max allowed is 100.
cube took: 1.0013580322265625e-05
125
```

For your convenience, we have separated the results of the two calls with a blank line. In the first call, the result is 8, which passes the threshold check. 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 we decorated the `cube()` function with the same two decorators but in a different order, the order of the printed messages would also have been different.

A decorator factory

Let's simplify this example now, going back to a single decorator: `max_result()`. We want to make it so that we can decorate different functions with different thresholds, as we don't want to write one decorator for each threshold. Let's therefore amend `max_result()` so that it allows us to decorate functions by specifying the threshold dynamically:

```python
# 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(
                    f'Result is too big ({result}). '
```

```
                    f'Max allowed is {threshold}.'
                )
            return result
        return wrapper
    return decorator


@max_result(75)
def cube(n):
    return n ** 3


print(cube(5))
```

The 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 `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.

The `wrapper()` function returns `result`, `decorator()` returns `wrapper()`, and `max_result()` returns `decorator()`. This means that our `cube = max_result(75)(cube)` call actually becomes `cube = decorator(cube)`. Not just any `decorator()` though, but one for which `threshold` has a value of `75`. This is achieved by a mechanism called **closure**.

> Dynamically created functions that are returned by other functions are called **closures**. Their main feature is that they have full access to the variables and names defined in the local namespace where they were created, even though the enclosing function has returned and finished executing.

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 us to use the `max_result()` decorator with different thresholds, like this:

```python
# 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 make the code simpler, and more elegant.

Object-oriented programming (OOP)

It's been quite a long and hopefully nice journey and, by now, we should be ready to explore OOP. We'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 a core feature of the language.

The two main players in OOP are **objects** and **classes**. Classes are used to create objects (objects are instances of the classes from 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

We will start with the simplest class you could ever write in Python:

```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) is Simplest)  # There's a better way to do this
```

Let's run the preceding code and explain it line by line:

```
$ python simplest.class.py
<class 'type'>
<class '__main__.Simplest'>
True
```

The `Simplest` class we defined has only the `pass` instruction in its body, which means it doesn't have any custom attributes or methods. Brackets after the name are optional if empty. We will print its type (`__main__` is the name of the scope in which top-level code executes), and we are aware that, in the highlighted comment, we wrote *object* instead of *class*. It turns out that, as you can see by the result of that `print` statement, *classes are actually objects*. To be precise, they are instances of `type`. Explaining this concept would lead us to a talk about **metaclasses** and **metaprogramming**, advanced concepts that require a solid grasp of the fundamentals to be understood and are beyond the scope of this chapter. As usual, we mentioned it to leave a pointer for you, for when you are ready to explore more deeply.

Let's go back to the example: we created `simp`, an instance of the `Simplest` class. You can see that the syntax to create an instance is the same as the syntax for calling a function. Next, we print what type `simp` belongs to and we verify that `simp` is, in fact, an instance of `Simplest`. We'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 in order to walk a namespace, all we need to do is to use the dot (.) operator.

Let's look at another example:

```python
# 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, we have defined a **class attribute** called `species`. Any name defined in the body of a class becomes an attribute that belongs to that class. In the code, we 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` also has two attributes that belong to its own namespace and are therefore called **instance attributes**: `name` and `surname`.

> **Class attributes** are shared among 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 will be specific to each individual object.

Attribute shadowing

When you search for an attribute on an object, if it is not found, Python extends the search to the attributes on 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:

```python
# 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 of `Point`, `p`, you can see that we can print both `x` and `y` from the `p` 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 (check out the section about scopes in *Chapter 4*, *Functions, the Building Blocks of Code*, for a refresher). We know that `x = 12` won't affect the global one, and for class and instance attributes, it is exactly the same.

After assigning `p.x = 12`, when we print it, the search doesn't need to reach the class attributes because x is found on the instance, so we get 12 printed out. We also print `Point.x`, which refers to x in the class namespace, to show it's still 10.

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 will no longer be found in the instance.

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? What if we created another instance of `Point`? Would that help to show why instance attributes can be very useful?

The self argument

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:

```python
# 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 take the name `self`, or you can use a more comfortable syntax, `sq.area()`, and Python will translate that for you behind the scenes.

Let's look at a better example:

```python
# 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 that the method will be bound to. We don't need to necessarily call it `self`, but it's the convention, and this is one of the few cases where it's very important to abide by it.

Initializing an instance

Have you noticed how, before calling `p1.final_price(...)` in the code above, 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; that is a technique that is mostly used when writing metaclasses which, as we mentioned, is a fairly advanced topic that we won't explore in the book. Let's now see an example of how to initialize objects in Python:

```python
# oop/class.init.py
class Rectangle:
    def __init__(self, side_a, side_b):
        self.side_a = side_a
        self.side_b = side_b

    def area(self):
        return self.side_a * self.side_b

r1 = Rectangle(10, 4)
```

```
print(r1.side_a, r1.side_b)  # 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, we wrote 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 side_a for r1 and r2, respectively, while 4 and 3 will be side_b. 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 more convenient.

In this example, we also declared attributes at the instance level, rather than at the class level, because it made sense to do so.

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

This is just half of the story though: *OOP is much more powerful than just this*. We have two main design constructs to use: 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. Let's declare a few engine types:

```
# oop/class_inheritance.py
class Engine:
    def start(self):
        pass

    def stop(self):
        pass
```

```python
class ElectricEngine(Engine):  # Is-A Engine
    pass

class V8Engine(Engine):  # Is-A Engine
    pass
```

Then we want to declare some car types that will use those engines:

```python
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
    pass  # engine_cls same as parent

car = Car()
racecar = RaceCar()
citycar = CityCar()
f1car = F1Car()
cars = [car, racecar, citycar, f1car]
```

```
for car in cars:
    car.start()
```

Running the above prints the following:

```
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 brackets 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. The *parent* and *base* classes are synonyms, and so are *child of* and *derived from*. Also, we say that a class *inherits* from another class, or that it *extends* it.

This is the inheritance mechanism.

Let us now 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 the `engine` instance attribute.

It makes sense to have `engine_cls` shared among all class instances because it's quite likely that all instances of the same car class will have the same kind of engine. On the other hand, it wouldn't be good to have a single engine (an instance of any `Engine` class) as a class attribute because we would be sharing one engine among 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 aspect 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 important to describe objects in this way so that we can use inheritance and composition correctly, to structure our code in the best way.

> Notice how we had to avoid having dots in the
> `class_inheritance.py` script name, as dots in module names
> make imports difficult. Most modules in the source code of the
> book are meant to be run as standalone scripts, so we chose to
> add dots to enhance readability when possible, but in general,
> you want to avoid dots in your module names.

Before we leave this paragraph, let's verify the correctness of what we stated above, with another example:

```python
# oop/class.issubclass.isinstance.py
from class_inheritance import Car, RaceCar, F1Car

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). Similarly, 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 whether an object is an instance of a class, use the `isinstance` function. It is recommended over sheer type comparison (`type(object) is Class`).

> Notice we have left out the prints you get when instantiating the cars. We saw them in the previous example.

Let's also check inheritance. The same setup, but different logic in the `for` loops:

```python
# 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 we 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 in snake case, like *this_way_is_correct*. Also, when in the code you want to use a name that clashes with a Python-reserved keyword or a built-in function or class, the convention is to add a trailing underscore to the name. In the first *for loop* example, we are 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 PEP 8, right?

To help you picture the difference between *Is-A* and *Has-A*, take a look at the following diagram:
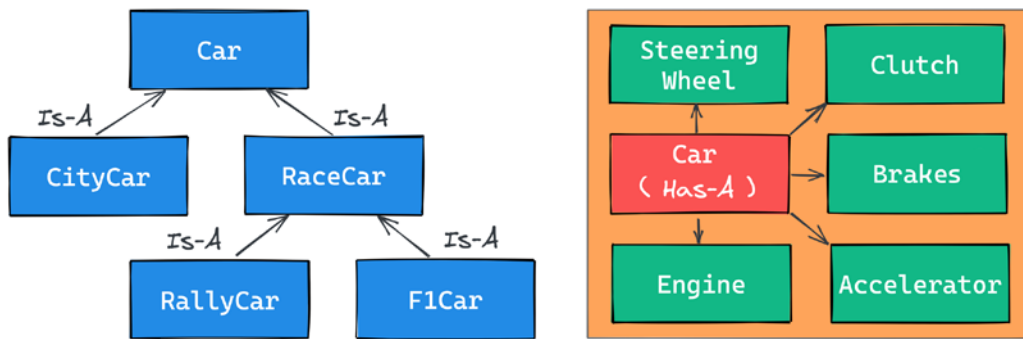
Figure 6.1: Is-A versus Has-A relationships

Accessing a base class

We've already seen class declarations, such as `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`. Please remember that, if you don't specify a base class, brackets are optional and in practice are never used.

Therefore, writing `class A: pass` or `class A(): pass` or `class A(object): pass` is exactly the same thing. The `object` class is a special class in that it hosts 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:

```python
# 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. Three of the input parameters for `Book` are duplicated in `Ebook`. 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 be reflected in `Ebook`. We know that `Ebook` *Is-A* `Book`, and therefore we probably want changes to be reflected in the child classes.

Let's see one way to fix this issue:

```python
# 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(
    'Learn Python Programming', 'Packt Publishing', 500, 'PDF')
print(ebook.title)  # Learn Python Programming
print(ebook.publisher)  # Packt Publishing
print(ebook.pages)  # 500
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; we feed self to that 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 automatically adapt to the change.

This approach is good, but we can still do a bit better. Say that we change the name of Book to Liber, because we've fallen in love with Latin. We would then have to change the __init__() method of Ebook to reflect that change. This can be avoided by using super:

```python
# 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(
    'Learn Python Programming', 'Packt Publishing', 500, 'PDF')
print(ebook.title)  # Learn Python Programming
print(ebook.publisher)  # Packt Publishing
print(ebook.pages)  # 500
print(ebook.format_)  # PDF
```

super() is a function that returns a proxy object that delegates method calls to a parent or sibling class.

Two classes are siblings if they share the same parents.

In this case, super() will delegate that call to __init__() of the Book class, and the beauty of this approach is that now we're 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 its 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:
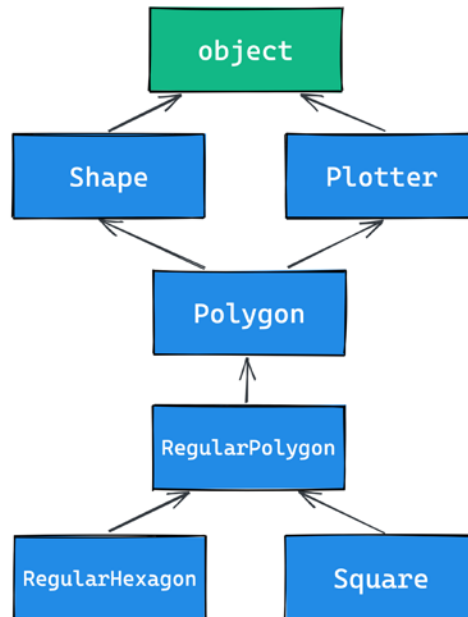
Figure 6.2: A class inheritance diagram

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 RegularPolygon. Note also that Shape and Plotter implicitly inherit from object, so we therefore have what is known as 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 the diagram into code:

```
# oop/multiple.inheritance.py
class Shape:
    geometric_type = 'Generic Shape'
    def area(self):  # This acts as placeholder for the interface
```

```python
        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 `Shape` class has one attribute, `geometric_type`, and two methods: `area()` and `get_geometric_type()`. It's quite common to use base classes (such as `Shape`, in our example) to define an *interface*, a set of methods for which children must provide an implementation. There are different and better ways to do this, but we 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`. For a regular polygon, where all sides are equal, we can implement a simple `__init__()` method, which just takes the length of the side. 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 we call the `area()` method on `hexagon` and `square`, we get the correct area for both. This is because they both provide the correct implementation logic for it. Also, we 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 will explore briefly in *Chapter 14, Introduction to API Development*), 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 of 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 we ask for `someobject.attribute` and `attribute` is not found on that object, Python starts searching in the class that `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 made of single-inheritance steps, which means that classes have only one parent, all the way up to `object`. 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 the order in which classes are searched on attribute lookup: the **method resolution order** (**MRO**).

> The MRO is the order in which base classes are searched for a member during lookup. Since 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 did not inherit 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 regard to the previous example, let's see 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 used `Square.__mro__`, or `Square.mro()` directly, but if you have to do it from an instance, you'll have to derive its class dynamically.

Note that the only point of doubt is the bifurcation 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, consider the following code:

```
# oop/mro.simple.py
class A:
    label = 'a'

class B(A):
    label = 'b'

class C(A):
    label = 'c'
```

```python
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 either 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 among the base classes of D. But what happens if we remove the label attribute from B? This would be a confusing situation: will the algorithm go all the way up to A or will it get to C first? Let's find out:

```python
# 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 that when we ask for d.label, we get **'c'**, which is correct.

In day-to-day programming, it is not common to have to deal with the MRO, but we felt it was important to at least mention it in this paragraph so that, should you get entangled in a complex mixins structure, you will be able to find your way out of it.

Class and static methods

So far, 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. Unlike instance methods, they are not passed any special argument, and therefore we don't need to create an instance of the class in order to call them. Let's look at an example of an imaginary StringUtil class:

```python
# oop/static.methods.py
class StringUtil:

    @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(StringUtil.is_palindrome(
    'Radar', case_insensitive=False))  # False: Case Sensitive
print(StringUtil.is_palindrome('A nut for a jar of tuna'))  # True
print(StringUtil.is_palindrome('Never Odd, Or Even!'))  # True
print(StringUtil.is_palindrome(
    'In Girum Imus Nocte Et Consumimur Igni')  # Latin! Show-off!
)  # True
```

```
print(StringUtil.get_unique_words(
    'I love palindromes. I really really love them!'))
# {'them!', 'palindromes.', 'I', 'really', '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, `StringUtil`, that 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 neither letters nor 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 to take any character from s if the character is either alphanumeric or a number. This is because, in palindrome sentences, we want to discard anything that is not a character or a number.

We then lowercase s if `case_insensitive` is `True`, and then we proceed to check whether it is a palindrome. 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. The measure `len(s) // 2` reaches half of s, and if s is an odd number of characters long, the middle one won't be checked (for instance, in *RaDaR*, *D* is not checked), but we don't care; it would be compared to itself, so it's always passing that check.

The `get_unique_words()` method 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, so we don't need to do anything else.

The `StringUtil` class provides us with a nice container namespace for methods that are meant to work on strings. We could have coded a similar example with a `MathUtil` class, and some static methods to work on numbers, but we wanted to show you something different.

Class methods

Class methods are slightly different from static methods in that, like instance methods, they also take a special first argument, but in this case, it is the class object itself, rather than the instance. A very common use case for coding class methods is to provide factory capability to a class, which means to have alternative ways to create instances of the class. Let's see an example:

```python
# 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, we show 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 each class method, the `cls` argument refers to the `Point` class. As with the instance method, which takes `self` as the first argument, the class method takes 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 professional Python coder would change; it is so strong a convention that plenty of tools, such as parsers, linters, and the like, rely on it.

Class and static methods play well together. Static methods are actually quite helpful in breaking up the logic of a class method to improve its layout.

Let's see an example by refactoring the `StringUtil` class:

```python
# oop/class.methods.split.py
class StringUtil:

    @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(StringUtil.is_palindrome('A nut for a jar of tuna'))  # True
print(StringUtil.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 by using the name of the class itself, like so: `StringUtil._strip_string(...)` and `StringUtil._is_palindrome(...)`, which is not good practice, because we would hardcode the class name in the `is_palindrome()` method, thereby putting ourselves in the position of having to modify it whenever we want to change the class name. Using `cls` means it will act as the class name, which means our code won't need any modifications should the class name change.

Notice how the new logic reads much better than the previous version. Moreover, notice that, by naming the *factored-out* methods with a leading underscore, we are 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#, or C++, 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, for privacy, 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 modify it, or call it from the outside. A very common use case for private attributes is helper methods that are supposed to be used by public ones (possibly in call chains in conjunction with other methods), and internal data, such as scaling factors, or any other data that we would ideally 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 both privacy and constraints. To be honest, in our professional experience with Python, we've never heard anyone screaming *"oh my God, we have a terrible bug because Python lacks private attributes!"* Not once, we swear.

That said, the call for privacy actually makes sense because without it, you risk introducing bugs into your code for real. Let us show you what we mean:

```python
# 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 so 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 is reflected 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, such as `__my_attr`, is replaced with a name that includes an underscore and the class name before the actual name, such as `_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 ensures 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 the 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, such as [18, 99]. You could write accessor methods, such as 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 set its value after checking its validity. 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 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 method 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 were 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:

```python
# 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, age):
        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 the decoration shown, which allows 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, such as Java, it's customary to define them as `get_age()` and `set_age(int value)`, but we 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 (by not writing the setter counterpart) and for special actions when the attribute is deleted. Please refer to the official documentation to dig deeper.

The cached_property decorator

One convenient use of properties is when we need to run some code in order to set up the object we want to use. For example, say we needed to connect to a database (or to an API).

In both cases, we might have to set up a client object that knows how to talk to the database (or to the API). It is quite common to use a property, in these cases, so that we can hide away the complexity of having to set the client up, and we can just use it. Let us show you a simplified example:

```python
class Client:
    def __init__(self):
        print("Setting up the client...")

    def query(self, **kwargs):
        print(f"Performing a query: {kwargs}")

class Manager:
    @property
    def client(self):
```

```
        return Client()

    def perform_query(self, **kwargs):
        return self.client.query(**kwargs)
```

In the preceding example, we have a dummy `Client` class, which prints the string *"Setting up the client…"* every time we create a new instance. It also has a pretend `query` method, that prints a string as well. We then have a class, `Manager`, which has a `client` property, which creates a new instance of `Client` every time it is called (for example, by a call to `perform_query`).

If we were to run this code, we would notice that every time we call `perform_query` on the manager, we see the string *"Setting up the client…"* being printed. When creating a client is expensive, this code would be wasting resources, so it might be better to cache that client, like this:

```
class ManualCacheManager:
    @property
    def client(self):
        if not hasattr(self, '_client'):
            self._client = Client()
        return self._client
```

The `ManualCacheManager` class is a bit smarter: the `client` property first checks if the attribute `_client` is on the class, by calling the built-in `hasattr` function. If not, it assigns `_client` to a new instance of `Client`. Finally, it simply returns it. Repeatedly accessing the `client` property on this class will only create one instance of `Client`, the first time. From the second call on, `_client` is returned with no need to create a new one.

This is such a common need that, in Python 3.8, the `functools` module added the `cached_property` decorator. The beauty of using that, instead of our manual solution, is that in case we need to refresh the client, we can simply delete the `client` property, and the next time we call it, it will recreate a brand new `Client` for us. Let's see an example:

```
from functools import cached_property

class CachedPropertyManager:
    @cached_property
    def client(self):
        return Client()

    def perform_query(self, **kwargs):
```

```
        return self.client.query(**kwargs)

manager = CachedPropertyManager()
manager.perform_query(object_id=42)
manager.perform_query(name_ilike='%Python%')
del manager.client  # This causes a new Client on next call
manager.perform_query(age_gte=18)
```

Running this code gives the following result:

```
$ python cached.property.py
Setting up the client...                        # New Client
Performing a query: {'object_id': 42}           # first query
Performing a query: {'name_ilike': '%Python%'}  # second query
Setting up the client...                        # Another Client
Performing a query: {'age_gte': 18}             # Third query
```

As you can see, it's only after we manually delete the `manager.client` that we get a new one when we invoke `manager.perform_query` again.

Python 3.9 also introduces a `cache` decorator, which can be used in conjunction with the `property` decorator, to cover scenarios for which `cached_property` is not suitable. As always, we encourage you to read up on all the details in the official Python documentation and experiment.

Operator overloading

We 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 a[k] call on a dictionary roughly translates to type(a).__getitem__(a, k). We can override these special methods for our purposes.

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 that corresponds to that of the stored string:

```
# oop/operator.overloading.py
class Weird:
```

```python
    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 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, such as 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 specific interface, of course).

In Python, things are different. Polymorphism is implicit, and nothing prevents you from calling 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 section on operator overloading. This is the ability of an operator to change shape according to the type of data it is fed.

Polymorphism also allows Python programmers to simply use the interface (methods and properties) exposed from an object rather than having to check which class it was instantiated from. This allows the code to be more compact and feel more natural.

We cannot spend too much time on polymorphism, but we encourage you to check it out by yourself; it will expand your understanding of OOP. Good luck!

Data classes

Before we leave the OOP realm, there is one last thing we want to mention: data classes. Introduced in Python 3.7 by PEP 557 (`https://www.python.org/dev/peps/pep-0557/`), they can be described as *mutable named tuples with defaults*. You can brush up on named tuples in *Chapter 2*, *Built-In Data Types*. Let's dive straight into an example:

```python
# oop/dataclass.py
from dataclasses import dataclass

@dataclass
class Body:
    '''Class to represent a physical body.'''
    name: str
    mass: float = 0.   # Kg
    speed: float = 1.   # m/s

    def kinetic_energy(self) -> float:
        return (self.mass * self.speed ** 2) / 2


body = Body('Ball', 19, 3.1415)
print(body.kinetic_energy())  # 93.755711375 Joule
print(body)  # Body(name='Ball', mass=19, speed=3.1415)
```

In the previous code, we have created a class to represent a physical body, with one method that allows us to calculate its kinetic energy (using the renowned formula $E_k = \frac{1}{2}mv^2$). Notice that `name` is supposed to be a string, while `mass` and `speed` are both floats, and both are given a default value. It's also interesting that we didn't have to write any `__init__()` method; it's done for us by the `dataclass` decorator, along with methods for comparison and for producing the string representation of the object (implicitly called on the last line by `print`).

You can read all the specifications in PEP 557 if you are curious, but for now, just remember that data classes might offer a nicer, slightly more powerful alternative to named tuples, in case you need it.

Writing a custom iterator

Now we have all the tools to appreciate how we can write our own custom iterator. Let's first define 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, and dictionaries are all iterables. Custom objects that define either of the __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 the __iter__() method that returns the object itself, and an implementation for the __next__() method that 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 the __iter__() and __next__() methods 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:

```python
# 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!')
```

```python
print(''.join(c for c in oddeven))  # TIICO!hssol

oddeven = OddEven('CiAo')  # or manually...
it = iter(oddeven)  # this calls oddeven.__iter__ internally
print(next(it))  # C
print(next(it))  # A
print(next(it))  # i
print(next(it))  # o
```

So, we needed to provide an implementation for `__iter__()` that returned the object itself, and then one for `__next__()`. Let's go through it. What needs to happen is the return of `_data[0]`, `_data[2]`, `_data[4]`, `...`, `_data[1]`, `_data[3]`, `_data[5]`, `...` until we have returned every item in the data. To do that, we prepare a list of indexes, such as [0, 2, 4, 6, ..., 1, 3, 5, ...], and while there is at least an element in it, we pop the first one out 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 that the end result works for all edge cases, empty sequences, sequences of lengths of 1, 2, and so on.

Summary

In this chapter, we looked at decorators, discovered the reasons for having them, and covered 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 have scratched the surface of object-oriented programming in Python. We covered all the basics, so you should now be able to understand the code that will come in future chapters. We talked about all kinds of methods and attributes that you 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 understand generators more deeply.

In the next chapter, we're going to learn about exceptions and context managers.

7
Exceptions and Context Managers

The best-laid schemes o' mice an' men
Gang aft agley

– Robert Burns

These famous lines by Robert Burns should be etched into the mind of every programmer. Even if our code is correct, errors will happen. If we don't deal with them properly, they can cause our best-laid schemes to go awry.

Unhandled errors can cause software to crash or misbehave. If you are lucky, this just results in an irritated user. If you're unlucky, your business can end up losing money (an e-commerce website that keeps crashing is not likely to be very successful). Therefore, it is important to learn how to detect and handle errors. It is also a good idea to cultivate the habit of always thinking about what errors can occur and how your code should respond when they do.

This chapter is all about errors and dealing with the unexpected. We'll be learning about **exceptions**, which are Python's way of signaling that an error or other exceptional event has occurred. We'll also talk about **context managers**, which provide a mechanism to encapsulate and re-use error handling code.

In this chapter, we are going to cover the following:

- Exceptions
- Context managers

Exceptions

Even though we haven't formally introduced them to you, by now we 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 met `IndexError` when we tried accessing a list at a position that was outside the valid range. We 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.

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 whether 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 into the 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 us see an example of a few exceptions:

```python
# 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_name)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'undefined_name' 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 `Traceback`, so that we have information about the error, but the shell itself still runs normally. This is a special behavior; a regular program or a script would normally exit immediately if nothing were done to handle exceptions. Let's see a quick example:

```
# exceptions/unhandled.py
1 + "one"
print("This line will never be reached")
```

If we run this code, we get the following output:

```
$ python exceptions/unhandled.py
Traceback (most recent call last):
  File "exceptions/unhandled.py", line 2, in <module>
    1 + "one"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Because we did nothing to handle the exception, Python immediately exits once an exception occurs (after printing out information about the error).

Raising exceptions

The exceptions we've seen so far were raised by the Python interpreter when it detected an error. However, you can also raise exceptions yourself, when a situation occurs that your own code considers to be an error. To raise an exception, use the `raise` statement. For example:

```
# exceptions/raising.py
>>> raise NotImplementedError("I'm afraid I can't do that")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NotImplementedError: I'm afraid I can't do that
```

You can use any exception type you want, but it's a good idea to choose the exception type that best describes the particular error condition that has occurred. You can even define your own exception types (we'll see how to do that in a moment). Notice that the argument we passed to the `Exception` class is printed out as part of the error message.

> Python has too many built-in exceptions to list here, but they are all documented at `https://docs.python.org/3.9/library/exceptions.html#bltin-exceptions`.

Defining your own exceptions

As we mentioned above, you can define your own custom exceptions. To do that, you just have to define a class that inherits from any other exception class. Ultimately, all exceptions derive from `BaseException`; however, this class is not intended to be directly subclassed, and your custom exceptions should inherit from `Exception` instead. In fact, nearly all built-in exceptions also inherit from `Exception`. Exceptions that do not inherit from `Exception` are meant for internal use by the Python interpreter.

Tracebacks

The **traceback** that Python prints might initially look quite intimidating, but it is extremely useful for understanding what happened to cause the exception. Let's look at a traceback and see what it can tell us:

```
# exceptions/trace.back.py
def squareroot(number):
    if number < 0:
```

```
        raise ValueError("No negative numbers please")
    return number ** .5

def quadratic(a, b, c):
    d = b ** 2 - 4 * a * c
    return ((-b - squareroot(d)) / (2 * a),
            (-b + squareroot(d)) / (2 * a))

quadratic(1, 0, 1)  # x**2 + 1 == 0
```

Here we defined a function called `quadratic()`, which uses the famous quadratic formula to find the solution of a quadratic equation. Instead of using the `sqrt()` function from the `math` module, we wrote our own version (`squareroot()`), which raises an exception if the number is negative. When we call `quadratic(1, 0, 1)` to solve the equation $x^2+1=0$, we will get a `ValueError`, because `d` is negative. When we run this, we get:

```
$ python exceptions/trace.back.py
Traceback (most recent call last):
  File "exceptions/trace.back.py", line 12, in <module>
    quadratic(1, 0, 1)  # x**2 + 1 == 0
  File "exceptions/trace.back.py", line 9, in quadratic
    return ((-b - squareroot(d)) / (2 * a),
  File "exceptions/trace.back.py", line 4, in squareroot
    raise ValueError("No negative numbers please")
ValueError: No negative numbers please
```

It is often useful to read tracebacks from bottom to top. On the very last line, we have the error message, telling us what went wrong: `ValueError: No negative numbers please`. The preceding lines tell us where the exception was raised (line 4 of `exceptions/trace.back.py` in the `squareroot()` function). We can also see the sequence of function calls that got us to the point where the exception was raised: `squareroot()` was called from line 9 in the function `quadratic()`, which was called from line 12, at the top level of the module. As you can see, the traceback is like a map that shows us the path through the code to where the exception happened. Following that path and examining the code in each function along the way is often very useful when you want to understand why an exception happened.

Handling exceptions

To handle an exception in Python, you use the `try` statement. When you enter the `try` clause, 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 composed of the `try` clause, which opens the statement, followed by one or more `except` clauses that define what to do when an exception is caught. The `except` clauses may optionally be followed by an `else` clause, which is executed when the `try` clause is exited without any exception raised. After `except` and `else` clauses we can have a `finally` clause (also optional), whose code is executed regardless of whatever happened in the other clauses. The `finally` clause is typically used to clean up resources. You are also allowed to omit the `except` and `else` clauses and only have a `try` clause followed by a `finally` clause. This is helpful if we want exceptions to be propagated and handled elsewhere, but we do have some cleanup code that must be executed regardless of whether an exception occurs.

The order of the clauses is important. It must be `try`, `except`, `else`, `finally`. Also, remember that `try` must be followed by at least one `except` clause or a `finally` clause. Let us see an example:

```python
# exceptions/try.syntax.py
def try_syntax(numerator, denominator):
    try:
        print(f'In the try block: {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))
```

This example defines a simple `try_syntax()` function. We perform the division of two numbers. We are prepared to catch a `ZeroDivisionError` exception, which will occur if we call the function with `denominator = 0`. Initially, the code enters the `try` block. If `denominator` is not `0`, `result` is calculated and, after leaving the `try` block, execution 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. Our attempt to calculate `numerator / denominator` raises a `ZeroDivisionError`. As a result, we enter the `except` block and print `zde`.

The `else` block is not 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 whether it makes sense to you:

```
$ python try.syntax.py
In the try block: 12/4      # try
The result is: 3.0          # else
Exiting                     # finally
3.0                         # return within else

In the try block: 11/0      # try
division by zero            # except
Exiting                     # finally
None                        # implicit return end of function
```

When you execute a `try` block, you may want to catch more than one exception. For example, when calling the `divmod()` function, you can get a `ZeroDivisionError` if the second argument is `0`, or `TypeError` if either argument is not a number. If you want to handle both in the same way, you can structure your code like this:

```
# exceptions/multiple.py
values = (1, 2)

try:
    q, r = divmod(*values)
except (ZeroDivisionError, TypeError) as e:
    print(type(e), e)
```

This code will catch both `ZeroDivisionError` and `TypeError`. Try changing `values = (1, 2)` to `values = (1, 0)` or `values = ('one', 2)`, and you will see the output change.

If you need to handle different exception types differently, you can just add more `except` clauses, like this:

```
# exceptions/multiple.py
try:
    q, r = divmod(*values)
except ZeroDivisionError:
    print("You tried to divide by zero!")
except TypeError as e:
    print(e)
```

Keep in mind that an exception is handled in the first block that matches that exception class or any of its base classes. 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.

> Python does also allow you to use an `except` clause without specifying any exception type (this is equivalent to writing `except BaseException`). Doing so is generally not a good idea as it means you will also capture exceptions that are intended for internal use by the interpreter. They include the so-called *system-exiting exceptions*. These are `SystemExit`, which is raised when the interpreter exits via a call to the `exit()` function, and `KeyboardInterrupt`, which is raised when the user terminates the application by pressing *Ctrl + C* (or *Delete* on some systems).

You can also raise exceptions from within an `except` clause. For example, you might want to replace a built-in exception (or one from a third-party library) with your own custom exception. This is quite a common technique when writing libraries, as it helps shield users from implementation details of the library. Let's see an example:

```python
# exceptions/replace.py
>>> class NotFoundError(Exception):
...     pass
...
>>> vowels = {'a': 1, 'e': 5, 'i': 9, 'o': 15, 'u': 21}
>>> try:
...     pos = vowels['y']
... except KeyError as e:
...     raise NotFoundError(*e.args)
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyError: 'y'


During handling of the above exception, another exception occurred:
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
__main__.NotFoundError: y
```

By default, Python assumes that an exception that happens within an `except` clause is an unexpected error and helpfully prints out tracebacks for both exceptions. We can tell the interpreter that we are deliberately raising the new exception by using a `raise from` statement:

```python
# exceptions/replace.py
>>> try:
...     pos = vowels['y']
... except KeyError as e:
...     raise NotFoundError(*e.args) from e
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyError: 'y'

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
__main__.NotFoundError: y
```

The error message has changed but we still get both tracebacks, which is actually very handy for debugging. If you really wanted to completely suppress the original exception, you could use `from None` instead of `from e` (try this yourself).

> You can also use `raise` by itself, without specifying a new exception, to re-raise the original exception. This is sometimes useful if you want to log the fact that an exception has occurred, without actually suppressing or replacing the exception.

Programming with exceptions can be very tricky. You could inadvertently hide bugs by trapping exceptions that would have alerted you to their presence. Play it safe by keeping these simple guidelines in mind:

- Keep the `try` clause as short as possible. It should contain only the code that may cause the exception(s) that you want to handle.

- Make the `except` clauses as specific as you can. It may be tempting to just write `except Exception`, but if you do you will almost certainly end up catching exceptions you did not actually intend to.

- Use tests to ensure that your code handles both expected and unexpected errors correctly. We shall talk more about writing tests in *Chapter 10*, *Testing*.

If you follow these suggestions, you will minimize the chance of getting it wrong.

Not only for errors

Before we talk about context managers, we want to show you an unconventional use of exceptions, just to give you something to help you expand your views on them. Exceptions can be used for more than just errors:

```python
# 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 this example, the condition is a trivial linear equation, but imagine something much cooler than that. What bugs us is having to check whether 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 breakout logic interferes with the rest of the code and we don't like it, so we came up with a different solution for this. Take a look at it, and see whether you can adapt it to other cases too:

```python
# 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.args)  # (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 `ExitLoopException` with the values that satisfy our condition, and immediately the control is given to the `except` clause that handles it. Notice that we can use the `args` attribute of the exception to get the values that were passed to the constructor.

Context managers

When working with external resources or global state, we often need to perform some cleanup steps, like releasing the resources or restoring the original state, when we are done. Failing to clean up properly could result in all manner of bugs. Therefore, we need to ensure that our cleanup code will be executed even if an exception happens. We could use `try`/`finally` statements, but this is not always convenient and could result in a lot of repetition, as we often have to perform similar cleanup steps whenever we work with a particular type of resource. **Context managers** solve this problem by creating an execution context in which we can work with a resource or modified state and automatically perform any necessary cleanup when we leave that context, even if an exception was raised.

One example of global state that we may want to modify temporarily is the precision for decimal computations. For example, suppose we need to perform a particular computation to a specific precision, but we want to retain the default precision for the rest of our computations. We might do something like the following:

> You may recall that the `Decimal` class allows us to perform arbitrary precision computations with decimal numbers. If not, you may want to review the relevant section of *Chapter 2*, *Built-In Data Types* now.

```python
# context/decimal.prec.py
from decimal import Context, Decimal, getcontext, setcontext

one = Decimal("1")
three = Decimal("3")

orig_ctx = getcontext()
ctx = Context(prec=5)
setcontext(ctx)
print(ctx)
print(one / three)
setcontext(orig_ctx)
print(one / three)
```

Notice that we store the current context, set a new context (with a modified precision), perform our calculation, and finally restore the original context. Running this produces the following output:

```
$ python decimal.prec.py
Context(prec=5, rounding=ROUND_HALF_EVEN, Emin=-999999,
Emax=999999, capitals=1, clamp=0, flags=[],
traps=[InvalidOperation, DivisionByZero, Overflow])
0.33333
0.3333333333333333333333333333
```

This seems fine, but what if an exception happened before we could restore the original context? We would be stuck with the wrong precision and the results of all subsequent computations would be incorrect! We can fix this by using a `try/finally` statement:

```python
# context/decimal.prec.py
orig_ctx = getcontext()
ctx = Context(prec=5)
setcontext(ctx)
try:
    print(ctx)
    print(one / three)
finally:
    setcontext(orig_ctx)
print(one / three)
```

That is much safer. Now we can rest assured that regardless of what happens in that `try` block, we will always restore the original context. It is not very convenient to have to keep writing `try/finally` like that, though. This is where context managers come to the rescue. The `decimal` module provides the `localcontext` context manager, which handles setting and restoring the context for us:

```python
# context/decimal.prec.py
from decimal import localcontext

with localcontext(Context(prec=5)) as ctx:
    print(ctx)
    print(one / three)
print(one / three)
```

That is much easier to read (and type)! The `with` statement is used to enter a runtime context defined by a context manager. When exiting the code block delimited by the `with` statement, any cleanup operation defined by the context manager (in this case, restoring the decimal context) is executed automatically.

It is also possible to combine multiple context managers in one `with` statement. This is quite useful for situations where you need to work with multiple resources at the same time:

```python
# context/decimal.prec.py
with localcontext(Context(prec=5)), open("out.txt", "w") as out_f:
    out_f.write(f"{one} / {three} = {one / three}\n")
```

Here, we enter a local context and open a file (which acts as a context manager) in one `with` statement. We perform a calculation and write the result to the file. When we're done, the file is automatically closed and the default decimal context is restored. Don't worry too much about the details of working with files for now; we will learn all about that in *Chapter 8*, *Files and Data Persistence*.

Apart from decimal contexts and files, many other objects in the Python standard library can be used as context managers. For example:

- Socket objects, which implement a low-level networking interface, can be used as context managers to automatically close network connections.
- The lock classes used for synchronization in concurrent programming use the context manager protocol to automatically release locks.

In the rest of this chapter, we will show you how you can implement your own context managers.

Class-based context managers

Context managers work via two magic methods: `__enter__()` is called just before entering the body of the `with` statement and `__exit__()` is called when exiting the `with` statement body. This means that you can easily create your own context manager simply by writing a class that implements these methods:

```python
# context/manager.class.py
class MyContextManager:
    def __init__(self):
        print("MyContextManager init", id(self))
```

```python
    def __enter__(self):
        print("Entering 'with' context")
        return self
    def __exit__(self, exc_type, exc_val, exc_tb):
        print(f"{exc_type=} {exc_val=} {exc_tb=}")
        print("Exiting 'with' context")
        return True
```

Here, we have defined a very simple context manager class called `MyContextManager`. There are a few interesting things to note about this class. Notice that our `__enter__()` method returns `self`. This is quite common, but by no means required: you can return whatever you want from `__enter__()`, even `None`. The return value of the `__enter__()` method will be assigned to the variable named in the `as` clause of the `with` statement. Also notice the `exc_type`, `exc_val`, and `exc_tb` parameters of the `__exit__()` function. If an exception is raised within the body of the `with` statement, the interpreter will pass the *type*, *value*, and *traceback* of the exception as arguments through these parameters. If no exception is raised, all three arguments will be `None`.

Also notice that our `__exit__()` method returns `True`. This will cause any exception raised within the `with` statement body to be suppressed (as if we had handled it in a `try`/`except` statement). Had we returned `False` instead, such an exception would continue to be propagated after our `__exit__()` method has executed. The ability to suppress exceptions means that a context manager can be used as an exception handler. The benefit of this is that we can write our exception handling logic once and reuse it wherever we need it. This is just one more way in which Python helps us to apply the **DRY** principle to our code.

Let us see our context manager in action:

```python
# context/manager.class.py
ctx_mgr = MyContextManager()
print("About to enter 'with' context")
with ctx_mgr as mgr:
    print("Inside 'with' context")
    print(id(mgr))
    raise Exception("Exception inside 'with' context")
    print("This line will never be reached")
print("After 'with' context")
```

Here, we have instantiated our context manager in a separate statement, before the `with` statement. We did this to make it easier for you to see what is happening; however, it is much more common for those steps to be combined like `with MyContextManager() as mgr`. Running this code produces the following output:

```
$ python context/manager.class.py
MyContextManager init 140340228792272
About to enter 'with' context
Entering 'with' context
Inside 'with' context
140340228792272
exc_type=<class 'Exception'> exc_val=Exception("Exception inside
'with' context") exc_tb=<traceback object at 0x7fa3817c5340>
Exiting 'with' context
After 'with' context
```

Study this output carefully to make sure you understand what is happening. We
have printed some IDs to help verify that the object assigned to mgr is really the same
object that we returned from __enter__(). Try changing the return values from the
__enter__() and __exit__() methods and see what effect that has.

Generator-based context managers

If you are implementing a class that represents some resource that needs to be
acquired and released, it makes sense to implement that class as a context manager.
Sometimes, however, we want to implement context manager behavior, but we
do not have a class that it makes sense to attach that behavior to. For example, we
may just want to use a context manager to re-use some error handling logic. In such
situations, it would be rather tedious to have to write an additional class purely
to implement the desired context manager behavior. Fortunately for us, Python
has a solution.

The contextlib module in the standard library provides a handy contextmanager
decorator that takes a *generator function* and converts it into a context manager
(if you don't remember how generator functions work, you should review *Chapter
5, Comprehensions and Generators*). Behind the scenes, the decorator wraps the
generator in a context manager object. The __enter__() method of this object starts
the generator and returns whatever the generator yields. If an exception occurs
within the with statement body, the __exit__() method passes the exception into the
generator (using the generator's throw method). Otherwise, __exit__() simply calls
next on the generator. Note that the generator must only yield once; a RuntimeError
will be raised if the generator yields a second time. Let us translate our previous
example into a generator-based context manager:

```python
# context/generator.py
from contextlib import contextmanager

@contextmanager
```

```python
def my_context_manager():
    print("Entering 'with' context")
    val = object()
    print(id(val))
    try:
        yield val
    except Exception as e:
        print(f"{type(e)=} {e=} {e.__traceback__=}")
    finally:
        print("Exiting 'with' context")

print("About to enter 'with' context")
with my_context_manager() as val:
    print("Inside 'with' context")
    print(id(val))
    raise Exception("Exception inside 'with' context")
    print("This line will never be reached")
print("After 'with' context")
```

The output from running this is very similar to the previous example:

```
$ python context/generator.py
About to enter 'with' context
Entering 'with' context
139768531985040
Inside 'with' context
139768531985040
type(e)=<class 'Exception'> e=Exception("Exception inside 'with'
context") e.__traceback__=<traceback object at 0x7f1e65a42800>
Exiting 'with' context
After 'with' context
```

Most context manager generators have a similar structure to `my_context_manager()` in this example. They have some setup code, followed by a `yield` statement inside a `try` statement. Here, we yielded an arbitrary object, so that you can see that the same object is made available via the `as` clause of the `with` statement. It is also quite common to have just a bare `yield` with no value (in which case `None` is yielded). In such cases, the `as` clause of the `with` statement will typically be omitted.

One very useful feature of generator-based context managers is that they can also be used as function decorators. This means that if the entire body of a function needs to be inside a `with` statement context, you could save a level of indentation and just decorate the function instead.

In addition to the `contextmanager` decorator, the `contextlib` module also contains many very useful context managers. The documentation also provides several helpful examples of using and implementing context managers. Make sure you read it at `https://docs.python.org/3/library/contextlib.html`.

The examples we gave in this section were deliberately quite simple. They needed to be simple, to make it easier to see how context managers work. Study these examples carefully until you are confident that you understand them completely. Then, start writing your own context managers (both as classes and generators). Try to convert the `try`/`except` statement for breaking out of a nested loop that we saw earlier in this chapter into a context manager. The `measure` decorator that we wrote in *Chapter 6, OOP, Decorators, and Iterators*, is also a good candidate for converting to a context manager.

Summary

In this chapter, we looked at exceptions and context managers.

We saw that exceptions are Python's way of signaling that an error has occurred. We showed you how to catch exceptions so that your program does not fail when errors inevitably do happen. We also showed you how you can raise exceptions yourself when your own code detects an error, and that you can even define your own exception types. We ended our exploration of exceptions by seeing that they are not only useful for signaling errors, but can also be used as a flow-control mechanism.

We ended the chapter with a brief overview of context managers. We saw how to use the `with` statement to enter a context defined by a context manager that performs cleanup operations when we exit the context. We also showed you how to create your own context managers, either as part of a class or by using a generator function.

We will see more context managers in action in the next chapter on files and data persistence.

8

Files and Data Persistence

"It's not that I'm so smart, it's just that I stay with problems longer."

– Albert Einstein

In the previous chapters, we have explored several different aspects of Python. As the examples have a didactic purpose, we've run them in a simple Python shell or in the form of a Python module. They ran, maybe printed something on the console, and then they terminated, leaving no trace of their brief existence.

Real-world applications, though, are rather different. Naturally, they still run in memory, but they interact with networks, disks, and databases. They also exchange information with other applications and devices, using formats that are suitable for the situation.

In this chapter, we are going to start closing in on the real world by exploring the following:

- Files and directories
- Compression
- Networks and streams
- The JSON data-interchange format
- Data persistence with `pickle` and `shelve`, from the standard library
- Data persistence with SQLAlchemy

As usual, we will try to balance breadth and depth so that by the end of the chapter, you will have a solid grasp of the fundamentals and will know how to fetch further information on the web.

Working with files and directories

When it comes to files and directories, Python offers plenty of useful tools. In particular, in the following examples, we will leverage the `os`, `pathlib`, and `shutil` modules. As we'll be reading and writing on the disk, we will be using a file, `fear.txt`, which contains an excerpt from *Fear*, by Thich Nhat Hanh, as a guinea pig for some of our examples.

Opening files

Opening a file in Python is very simple and intuitive. In fact, we just need to use the `open()` function. Let's see a quick example:

```python
# files/open_try.py
fh = open('fear.txt', 'rt')  # r: read, t: text

for line in fh.readlines():
    print(line.strip())  # remove whitespace and print

fh.close()
```

The previous code is very simple. We call `open()`, passing the filename, and telling `open()` that we want to read it in text mode. There is no path information before the filename; therefore, `open()` will assume the file is in the same folder the script is run from. This means that if we run this script from outside the `files` folder, then `fear.txt` won't be found.

Once the file has been opened, we obtain a file object back, `fh`, which we can use to work on the content of the file. In this case, we use the `readlines()` method to iterate over all the lines in the file, and print them. We call `strip()` on each line to get rid of any extra spaces around the content, including the line termination character at the end, since `print` will already add one for us. This is a quick and dirty solution that works in this example, but should the content of the file contain meaningful spaces that need to be preserved, you will have to be slightly more careful in how you sanitize the data. At the end of the script, we close the stream.

Closing a file is very important, as we don't want to risk failing to release the handle we have on it. When that happens, you can encounter issues such as memory leaks, or the annoying "you can't delete this file" pop-up that informs you that some software is still using it. Therefore, we need to apply some precautions, and wrap the previous logic in a `try/finally` block. This means that, whatever error might occur when we try to open and read the file, we can rest assured that `close()` will be called:

```python
# files/open_try.py

fh = open('fear.txt', 'rt')

try:
    for line in fh.readlines():
        print(line.strip())
finally:
    fh.close()
```

The logic is exactly the same, but now it is also safe.

> If you are not familiar with the `try/finally` block, make sure you go back to *Chapter 7*, *Exceptions and Context Managers*, and study it.

We can simplify the previous example further, this way:

```python
# files/open_try.py

fh = open('fear.txt')  # rt is default

try:
    for line in fh:  # we can iterate directly on fh
        print(line.strip())
finally:
    fh.close()
```

As you can see, `rt` is the default mode for opening files, so we don't need to specify it. Moreover, we can simply iterate on `fh`, without explicitly calling `readlines()` on it. Python is very nice and gives us shorthands to make our code shorter and simpler to read.

All the previous examples produce a print of the file on the console (check out the source code to read the whole content):

```
An excerpt from Fear - By Thich Nhat Hanh

The Present Is Free from Fear

When we are not fully present, we are not really living. We're not
really there, either for our loved ones or for ourselves. If we're not
there, then where are we? We are running, running, running, even during
our sleep. We run because we're trying to escape from our fear.
...
```

Using a context manager to open a file

Let's admit it: the prospect of having to disseminate our code with `try`/`finally` blocks is not one of the best. As usual, Python gives us a much nicer way to open a file in a secure fashion: by using a context manager. Let's see the code first:

```python
# files/open_with.py
with open('fear.txt') as fh:
    for line in fh:
        print(line.strip())
```

This example is equivalent to the previous one, but reads so much better. The `open()` function is capable of producing a file object when invoked by a context manager, but the true beauty of it lies in the fact that `fh.close()` will be called automatically for us, even in the case of errors.

Reading and writing to a file

Now that we know how to open a file, let's see a couple of different ways in which we can read and write to it:

```python
# files/print_file.py
with open('print_example.txt', 'w') as fw:
    print('Hey I am printing into a file!!!', file=fw)
```

A first approach uses the `print()` function, which we've seen plenty of times in the previous chapters. After obtaining a file object, this time specifying that we intend to write to it (`'w'`), we can tell the call to `print()` to direct its output to the file, instead of to the **standard output** stream as it normally does.

In Python, the standard input, output, and error streams are represented by the file objects `sys.stdin`, `sys.stdout`, and `sys.stderr`. Unless input or output is redirected, reading from `sys.stdin` usually corresponds to reading from the keyboard and writing to `sys.stdout` or `sys.stderr` usually prints to the console screen.

The previous code has the effect of creating the `print_example.txt` file if it doesn't exist, or truncating it in case if, and writes the line `Hey I am printing into a file!!!` into it.

Truncating a file means erasing its contents without deleting it. After truncation, the file still exists on the filesystem, but it's empty.

This is all nice and easy, but not what we typically do when we want to write to a file. Let's see a much more common approach:

```python
# files/read_write.py
with open('fear.txt') as f:
    lines = [line.rstrip() for line in f]

with open('fear_copy.txt', 'w') as fw:
    fw.write('\n'.join(lines))
```

In this example, we first open `fear.txt` and collect its content into a list, line by line. Notice that this time, we are calling a different method, `rstrip()`, as an example, to make sure we only strip the whitespace on the right-hand side of every line.

In the second part of the snippet, we create a new file, `fear_copy.txt`, and we write to it all the lines from the original file, joined by a newline, \n. Python is gracious and works by default with **universal newlines**, which means that even though the original file might have a newline that is different to \n, it will be translated automatically for us before the line is returned. This behavior is, of course, customizable, but normally it is exactly what you want. Speaking of newlines, can you think of one that might be missing in the copy?

Reading and writing in binary mode

Notice that by opening a file passing `t` in the options (or omitting it, as it is the default), we're opening the file in text mode. This means that the content of the file is treated and interpreted as text.

If you wish to write bytes to a file, you can open it in binary mode. This is a common requirement when you deal with files that don't just contain raw text, such as images, audio/video, and, in general, any other proprietary format.

In order to handle files in binary mode, simply specify the `b` flag when opening them, as in the following example:

```python
# files/read_write_bin.py
with open('example.bin', 'wb') as fw:
    fw.write(b'This is binary data...')

with open('example.bin', 'rb') as f:
    print(f.read())  # prints: b'This is binary data...'
```

In this example, we are still using text as binary data, for simplicity, but it could be anything you want. You can see it's treated as binary by the fact that you get the `b'This ...'` prefix in the output.

Protecting against overwriting an existing file

As we have seen, Python gives us the ability to open files for writing. By using the `w` flag, we open a file and truncate its content. This means the file is overwritten with an empty file, and the original content is lost. If you wish to only open a file for writing if it doesn't already exist, you can use the `x` flag instead, as in the following example:

```python
# files/write_not_exists.py
with open('write_x.txt', 'x') as fw:  # this succeeds
    fw.write('Writing line 1')

with open('write_x.txt', 'x') as fw:  # this fails
    fw.write('Writing line 2')
```

If you run this snippet, you will find a file called `write_x.txt` in your directory, containing only one line of text. The second part of the snippet, in fact, fails to execute. This is the output we get on our console (the file path has been shortened for editorial purposes):

```
$ python write_not_exists.py
Traceback (most recent call last):
  File "/…/ch08/files/write_not_exists.py", line 6, in <module>
    with open('write_x.txt', 'x') as fw:
FileExistsError: [Errno 17] File exists: 'write_x.txt'
```

Checking for file and directory existence

If you want to make sure a file or directory exists (or doesn't), the `pathlib` module is what you need. Let's see a small example:

```python
# files/existence.py
from pathlib import Path

p = Path('fear.txt')
path = p.parent.absolute()

print(p.is_file())        # True
print(path)               # /Users/fab/srv/lpp3e/ch08/files
print(path.is_dir())      # True

q = Path('/Users/fab/srv/lpp3e/ch08/files')
print(q.is_dir())         # True
```

The preceding snippet is quite interesting. We create a `Path` object that we set up with the name of the text file we want to inspect. We use the `parent()` method to retrieve the folder in which the file is contained, and we call the `absolute()` method on it, to extract the absolute path information.

We check if `'fear.txt'` is a file, and the folder in which it is contained is indeed a folder (or directory, which is equivalent).

The old way to do these operations was to use the `os.path` module from the standard library. While `os.path` works on strings, `pathlib` offers classes representing filesystem paths with semantics appropriate for different operating systems. Hence, we suggest using `pathlib` whenever possible, and reverting to the old way of doing things only when there is no alternative.

Manipulating files and directories

Let's see a couple of quick examples on how to manipulate files and directories. The first example manipulates the content:

```python
# files/manipulation.py
from collections import Counter
from string import ascii_letters

chars = ascii_letters + ' '
```

```python
def sanitize(s, chars):
    return ''.join(c for c in s if c in chars)

def reverse(s):
    return s[::-1]

with open('fear.txt') as stream:
    lines = [line.rstrip() for line in stream]

# let's write the mirrored version of the file
with open('raef.txt', 'w') as stream:
    stream.write('\n'.join(reverse(line) for line in lines))

# now we can calculate some statistics
lines = [sanitize(line, chars) for line in lines]
whole = ' '.join(lines)

# we perform comparisons on the lowercased version of 'whole'
cnt = Counter(whole.lower().split())

# we can print the N most common words
print(cnt.most_common(3))
```

This example defines two functions: `sanitize()` and `reverse()`. They are simple functions whose purpose is to remove anything that is not a letter or space from a string, and produce the reversed copy of a string, respectively.

We open `fear.txt` and we read its content into a list. Then we create a new file, `raef.txt`, which will contain the horizontally-mirrored version of the original one. We write all the content of `lines` with a single operation, using `join` on a newline character. Maybe more interesting is the bit at the end. First, we reassign `lines` to a sanitized version of itself by means of a list comprehension. Then we put the lines together in the `whole` string, and finally, we pass the result to a `Counter` object. Notice that we split the lowercased version of the string into a list of words. This way, each word will be counted correctly, regardless of its case, and, thanks to `split()`, we don't need to worry about extra spaces anywhere. When we print the three most common words, we realize that, truly, Thich Nhat Hanh's focus is on others, as `we` is the most common word in the text:

```
$ python manipulation.py
[('we', 17), ('the', 13), ('were', 7)]
```

Let's now see an example of manipulation more oriented to disk operations, in which we put the `shutil` module to use:

```python
# files/ops_create.py
import shutil
from pathlib import Path

base_path = Path('ops_example')

# let's perform an initial cleanup just in case
if base_path.exists() and base_path.is_dir():
    shutil.rmtree(base_path)

# now we create the directory
base_path.mkdir()

path_b = base_path / 'A' / 'B'
path_c = base_path / 'A' / 'C'
path_d = base_path / 'A' / 'D'

path_b.mkdir(parents=True)
path_c.mkdir()  # no need for parents now, as 'A' has been created

# we add three files in 'ops_example/A/B'
for filename in ('ex1.txt', 'ex2.txt', 'ex3.txt'):
    with open(path_b / filename, 'w') as stream:
        stream.write(f'Some content here in {filename}\n')

shutil.move(path_b, path_d)

# we can also rename files
ex1 = path_d / 'ex1.txt'
ex1.rename(ex1.parent / 'ex1.renamed.txt')
```

In the preceding code, we start by declaring a base path, which will safely contain all the files and folders we're going to create. We then use `mkdir()` to create two directories: `ops_example/A/B` and `ops_example/A/C`. Notice we don't need to specify `parents=True` when calling `path_c.mkdir()`, since all the parents have already been created by the previous call on `path_b`.

We use the `/` operator to concatenate directory names; `pathlib` takes care of using the right path separator for us, behind the scenes.

After creating the directories, we use a simple `for` loop to create three files in directory `B`. Then, we move directory `B` and its contents to a different name: `D`. And finally, we rename `ex1.txt` to `ex1.renamed.txt`. If you open that file, you'll see it still contains the original text from the `for` loop logic. Calling `tree` on the result produces the following:

```
$ tree ops_example/
ops_example/
└── A
    ├── C
    └── D
        ├── ex1.renamed.txt
        ├── ex2.txt
        └── ex3.txt
```

Manipulating pathnames

Let's explore the abilities of `pathlib` a little more by means of a simple example:

```python
# files/paths.py
from pathlib import Path

p = Path('fear.txt')

print(p.absolute())
print(p.name)
print(p.parent.absolute())
print(p.suffix)

print(p.parts)
print(p.absolute().parts)

readme_path = p.parent / '..' / '..' / 'README.rst'
print(readme_path.absolute())
print(readme_path.resolve())
```

Reading the result is probably a good enough explanation for this simple example:

```
/Users/fab/srv/lpp3e/ch08/files/fear.txt
fear.txt
```

```
/Users/fab/srv/lpp3e/ch08/files
.txt
('fear.txt',)
('/', 'Users', 'fab', 'srv', 'lpp3e', 'ch08', 'files', 'fear.txt')
/Users/fab/srv/lpp3e/ch08/files/../../README.rst
/Users/fab/srv/lpp3e/README.rst
```

Note how, in the last two lines, we have two different representations of the same path. The first one (`readme_path.absolute()`) shows two `'..'`, a single one of which, in path terms, indicates changing to the parent folder. So, by changing to the parent folder twice in a row, from …`/lpp3e/ch08/files/` we go back to …`/lpp3e/`. This is confirmed by the last line in the example, which shows the output of `readme_path.resolve()`.

Temporary files and directories

Sometimes, it's very useful to be able to create a temporary directory or file when running some code. For example, when writing tests that affect the disk, you can use temporary files and directories to run your logic and assert that it's correct, and to be sure that at the end of the test run, the test folder has no leftovers. Let's see how to do it in Python:

```python
# files/tmp.py
from tempfile import NamedTemporaryFile, TemporaryDirectory

with TemporaryDirectory(dir='.') as td:
    print('Temp directory:', td)
    with NamedTemporaryFile(dir=td) as t:
        name = t.name
        print(name)
```

The preceding example is quite straightforward: we create a temporary directory in the current one (`"."`), and we create a named temporary file in it. We print the filename, as well as its full path:

```
$ python tmp.py
Temp directory: ./tmpz5i9ne20
/Users/fab/srv/lpp3e/ch08/files/tmpz5i9ne20/tmp2e3j8p78
```

Running this script will produce a different result every time. After all, it's a temporary random name we're creating here, right?

Directory content

With Python, you can also inspect the contents of a directory. We will show you two ways of doing this. This is the first:

```python
# files/listing.py
from pathlib import Path

p = Path('.')
for entry in p.glob('*'):
    print('File:' if entry.is_file() else 'Folder:', entry)
```

This snippet uses the `glob()` method of a `Path` object, applied from the current directory. We iterate on the results, each of which is an instance of a subclass of `Path` (`PosixPath` or `WindowsPath`, according to which OS we are running). For each `entry`, we inspect if it is a directory, and print accordingly. Running the code yields the following (we omitted a few results for brevity):

```
$ python listing.py
File: existence.py
File: fear.txt
…
Folder: compression
…
File: walking.pathlib.py
…
```

An alternative way to scan a directory tree is given to us by `os.walk`. Let's see an example:

```python
# files/walking.py
import os

for root, dirs, files in os.walk('.'):
    abs_root = os.path.abspath(root)
    print(abs_root)

    if dirs:
        print('Directories:')
        for dir_ in dirs:
```

```
                print(dir_)
            print()

        if files:
            print('Files:')
            for filename in files:
                print(filename)
            print()
```

Running the preceding snippet will produce a list of all the files and directories in the current one, and it will do the same for each sub-directory.

File and directory compression

Before we leave this section, let us give you an example of how to create a compressed file. In the source code of the book, we have two examples: one creates a *.zip* file, while the other one creates a *tar.gz* file. Python allows you to create compressed files in several different ways and formats. Here, we are going to show you how to create the most common one, **ZIP**:

```python
# files/compression/zip.py
from zipfile import ZipFile

with ZipFile('example.zip', 'w') as zp:
    zp.write('content1.txt')
    zp.write('content2.txt')
    zp.write('subfolder/content3.txt')
    zp.write('subfolder/content4.txt')

with ZipFile('example.zip') as zp:
    zp.extract('content1.txt', 'extract_zip')
    zp.extract('subfolder/content3.txt', 'extract_zip')
```

In the preceding code, we import `ZipFile`, and then, within a context manager, we write into it four files (two of which are in a sub-folder, to show how ZIP preserves the full path). Afterward, as an example, we open the compressed file and extract a couple of files from it into the `extract_zip` directory. If you are interested in learning more about data compression, make sure you check out the *Data Compression and Archiving* section on the standard library (`https://docs.python.org/3.9/library/archiving.html`), where you'll be able to learn all about this topic.

Data interchange formats

Modern software architecture tends to split an application into several components. Whether you embrace the service-oriented architecture paradigm, or you push it even further into the microservices realm, these components will have to exchange data. But even if you are coding a monolithic application whose codebase is contained in one project, chances are that you still have to exchange data with APIs, other programs, or simply handle the data flow between the frontend and backend parts of your website, which very likely won't speak the same language.

Choosing the right format in which to exchange information is crucial. A language-specific format has the advantage that the language itself is very likely to provide you with all the tools to make **serialization** and **deserialization** a breeze. However, you will lose the ability to talk to other components that have been written in different versions of the same language, or in different languages altogether. Regardless of what the future looks like, going with a language-specific format should only be done if it is the only possible choice for the given situation.

According to Wikipedia (`https://en.wikipedia.org/wiki/Serialization`):

> *In computing, serialization is the process of translating a data structure or object state into a format that can be stored (for example, in a file or memory data buffer) or transmitted (for example, over a computer network) and reconstructed later (possibly in a different computer environment).*

A much better approach is to choose a format that is language-agnostic, and can be spoken by all (or at least most) languages. Fabrizio used to lead a team of programmers from England, Poland, South Africa, Spain, Greece, India, and Italy, to mention just a few. They all spoke English, so regardless of their native tongue, they could all understand each other (well... mostly!).

In the software world, some popular formats have become the de facto standard for data interchange. The most famous ones probably are **XML**, **YAML**, and **JSON**. The Python standard library features the `xml` and `json` modules, and, on PyPI (`https://pypi.org/`), you can find a few different packages to work with YAML.

In the Python environment, JSON is perhaps the most commonly used one. It wins over the other two because of being part of the standard library, and for its simplicity. If you have ever worked with XML, you know what a nightmare it can be.

Moreover, when working with a database like PostgreSQL, the ability to use native JSON fields makes a compelling case for adopting JSON in the application as well.

Working with JSON

JSON is the acronym for **JavaScript Object Notation**, and it is a subset of the JavaScript language. It has been around for almost two decades now, so it is well known and widely adopted by most languages, even though it is actually language-independent. You can read all about it on its website (`https://www.json.org/`), but we are going to give you a quick introduction to it now.

JSON is based on two structures: a collection of name/value pairs, and an ordered list of values. It's quite straightforward to realize that these two objects map to the `dict` and `list` data types in Python, respectively. As data types, JSON offers strings, numbers, objects, and values consisting of `true`, `false`, and `null`. Let's see a quick example to get us started:

```python
# json_examples/json_basic.py
import sys
import json

data = {
    'big_number': 2 ** 3141,
    'max_float': sys.float_info.max,
    'a_list': [2, 3, 5, 7],
}

json_data = json.dumps(data)
data_out = json.loads(json_data)
assert data == data_out  # json and back, data matches
```

We begin by importing the `sys` and `json` modules. Then we create a simple dictionary with some numbers inside and a list. We wanted to test serializing and deserializing using very big numbers, both `int` and `float`, so we put 2^{3141} and whatever is the biggest floating point number our system can handle.

We serialize with `json.dumps()`, which takes data and converts it into a JSON formatted string. That data is then fed into `json.loads()`, which does the opposite: from a JSON formatted string, it reconstructs the data into Python. On the last line, we make sure that the original data and the result of the serialization/deserialization through JSON match.

Let's see what JSON data would look like if we printed it:

```python
# json_examples/json_basic.py
import json

info = {
    'full_name': 'Sherlock Holmes',
    'address': {
        'street': '221B Baker St',
        'zip': 'NW1 6XE',
        'city': 'London',
        'country': 'UK',
    }
}

print(json.dumps(info, indent=2, sort_keys=True))
```

In this example, we create a dictionary with Sherlock Holmes' data in it. If, like us, you are a fan of Sherlock Holmes, and are in London, you will find his museum at that address (which we recommend visiting; it's small but very nice).

Notice how we call `json.dumps`, though. We have told it to indent with two spaces, and sort keys alphabetically. The result is this:

```
$ python json_basic.py
{
  "address": {
    "city": "London",
    "country": "UK",
    "street": "221B Baker St",
    "zip": "NW1 6XE"
  },
  "full_name": "Sherlock Holmes"
}
```

The similarity with Python is evident. The one difference is that if you place a comma on the last element in a dictionary, as is customary in Python, JSON will complain.

Let us show you something interesting:

```python
# json_examples/json_tuple.py
import json

data_in = {
```

```
    'a_tuple': (1, 2, 3, 4, 5),
}

json_data = json.dumps(data_in)
print(json_data)  # {"a_tuple": [1, 2, 3, 4, 5]}
data_out = json.loads(json_data)
print(data_out)  # {'a_tuple': [1, 2, 3, 4, 5]}
```

In this example, we have used a tuple instead of a list. The interesting bit is that, conceptually, a tuple is also an ordered list of items. It doesn't have the flexibility of a list, but still, it is considered the same from the perspective of JSON. Therefore, as you can see by the first `print`, in JSON a tuple is transformed into a list. Naturally then, the information that the original object was a tuple is lost, and when deserialization happens, `a_tuple` is actually translated to a Python list. It is important that you keep this in mind when dealing with data, as going through a transformation process that involves a format that only comprises a subset of the data structures you can use implies there may be information loss. In this case, we lost the information about the type (tuple versus list).

This is actually a common problem. For example, you can't serialize all Python objects to JSON, as it is not always clear how JSON should revert that object. Think about `datetime`, for example. An instance of that class is a Python object that JSON won't be able to serialize. If we transform it into a string such as `2018-03-04T12:00:30Z`, which is the ISO 8601 representation of a date with time and time zone information, what should JSON do when deserializing? Should it decide that *this is actually deserializable into a datetime object, so I'd better do it*, or should it simply consider it as a string and leave it as it is? What about data types that can be interpreted in more than one way?

The answer is that when dealing with data interchange, we often need to transform our objects into a simpler format prior to serializing them with JSON. The more we manage to simplify our data, the easier it is to represent that data in a format like JSON, which has limitations.

In some cases, though, and mostly for internal use, it is useful to be able to serialize custom objects, so, just for fun, we are going to show you how with two examples: complex numbers (because we love math) and *datetime* objects.

Custom encoding/decoding with JSON

In the JSON world, we can consider terms like encoding/decoding as synonyms for serializing/deserializing. They basically mean transforming to and back from JSON.

In the following example, we are going to learn how to encode complex numbers –
which aren't serializable to JSON by default – by writing a custom encoder:

```python
# json_examples/json_cplx.py
import json

class ComplexEncoder(json.JSONEncoder):
    def default(self, obj):

        print(f"ComplexEncoder.default: {obj=}")
        if isinstance(obj, complex):
            return {
                '_meta': '_complex',
                'num': [obj.real, obj.imag],
            }
        return super().default(obj)

data = {
    'an_int': 42,
    'a_float': 3.14159265,
    'a_complex': 3 + 4j,
}

json_data = json.dumps(data, cls=ComplexEncoder)
print(json_data)

def object_hook(obj):
    print(f"object_hook: {obj=}")
    try:
        if obj['_meta'] == '_complex':
            return complex(*obj['num'])
    except KeyError:
        return obj

data_out = json.loads(json_data, object_hook=object_hook)
print(data_out)
```

We start by defining a `ComplexEncoder` class as a subclass of the `JSONEncoder` class.
Our class overrides the `default` method. This method is called whenever the encoder
encounters an object that it cannot encode and is expected to return an encodable
representation of that object.

Our `default()` method checks whether its argument is a `complex` object, in which case it returns a dictionary with some custom meta information, and a list that contains both the real and the imaginary part of the number. That is all we need to do to avoid losing information for a complex number. If we receive anything other than an instance of `complex`, we call the `default()` method from the parent class, which just raises a `TypeError`. We then call `json.dumps()`, but this time we use the `cls` argument to specify our custom encoder. The result is printed:

```
$ python json_cplx.py
ComplexEncoder.default: obj=(3+4j)
{"an_int": 42, "a_float": 3.14159265,
 "a_complex": {"_meta": "_complex", "num": [3.0, 4.0]}}
```

Half the job is done. For the deserialization part, we could have written another class that would inherit from `JSONDecoder`, but instead we have chosen to use a different technique that is simpler and uses a small function: `object_hook`.

Within the body of `object_hook()`, we find a `try` block. The important part is the two lines within the body of the `try` block itself. The function receives an object (notice that the function is only called when `obj` is a dictionary), and if the metadata matches our convention for complex numbers, we pass the real and imaginary parts to the `complex()` function. The `try/except` block is there because our function will be called for every dictionary object that is decoded, so we need to handle the case where our `_meta` key is not present.

The decoding part of the example outputs:

```
object_hook: obj={'_meta': '_complex', 'num': [3.0, 4.0]}
object_hook: obj={'an_int': 42, 'a_float': 3.14159265, 'a_complex':
(3+4j)}
{'an_int': 42, 'a_float': 3.14159265, 'a_complex': (3+4j)}
```

You can see that `a_complex` has been correctly deserialized.

Let's now consider a slightly more complex (no pun intended) example: dealing with `datetime` objects. We are going to split the code into two blocks, first the serializing part, and then the deserializing one:

```python
# json_examples/json_datetime.py
import json
from datetime import datetime, timedelta, timezone

now = datetime.now()
now_tz = datetime.now(tz=timezone(timedelta(hours=1)))
```

```python
class DatetimeEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, datetime):
            try:
                off = obj.utcoffset().seconds
            except AttributeError:
                off = None

            return {
                '_meta': '_datetime',
                'data': obj.timetuple()[:6] + (obj.microsecond, ),
                'utcoffset': off,
            }
        return super().default(obj)

data = {
    'an_int': 42,
    'a_float': 3.14159265,
    'a_datetime': now,
    'a_datetime_tz': now_tz,
}
json_data = json.dumps(data, cls=DatetimeEncoder)
print(json_data)
```

The reason why this example is slightly more complex lies in the fact that `datetime` objects in Python can be time zone-aware or not; therefore, we need to be more careful. The flow is the same as before, only we are dealing with a different data type. We start by getting the current date and time information, and we do it both without (`now`) and with (`now_tz`) time zone awareness, just to make sure our script works. We then proceed to define a custom encoder as before, overriding the `default()` method. The important bits in that method are how we get the time zone offset (`off`) information, in seconds, and how we structure the dictionary that returns the data. This time, the metadata says it's *datetime* information. We save the first six items in the time tuple (year, month, day, hour, minute, and second), plus the microseconds in the `data` key, and the offset after that. Could you tell that the value of `'data'` is a concatenation of tuples? Good job if you could!

When we have our custom encoder, we proceed to create some data, and then we serialize. The `print` statement outputs the following (we have reformatted the output to make it more readable):

```
{
    "an_int": 42,
```

```
    "a_float": 3.14159265,
    "a_datetime": {
        "_meta": "_datetime",
        "data": [2021, 5, 17, 23, 1, 58, 75097],
        "utcoffset": null
    },
    "a_datetime_tz": {
        "_meta": "_datetime",
        "data": [2021, 5, 17, 23, 1, 58, 75112],
        "utcoffset": 3600
    }
}
```

Interestingly, we find out that `None` is translated to `null`, its JavaScript equivalent. Moreover, we can see that our data seems to have been encoded properly. Let's proceed with the second part of the script:

```python
# json_examples/json_datetime.py
def object_hook(obj):
    try:
        if obj['_meta'] == '_datetime':
            if obj['utcoffset'] is None:
                tz = None
            else:
                tz = timezone(timedelta(seconds=obj['utcoffset']))
            return datetime(*obj['data'], tzinfo=tz)
    except KeyError:
        return obj

data_out = json.loads(json_data, object_hook=object_hook)
```

Once again, we first verify that the metadata is telling us it's a `datetime`, and then we proceed to fetch the time zone information. Once we have that, we pass the 7-tuple (using * to unpack its values in the call) and the time zone information to the `datetime()` call, getting back our original object. Let's verify it by printing `data_out`:

```
{
    'a_datetime': datetime.datetime(
        2021, 5, 17, 23, 10, 2, 830913
    ),
    'a_datetime_tz': datetime.datetime(
        2021, 5, 17, 23, 10, 2, 830927,
        tzinfo=datetime.timezone(datetime.timedelta(seconds=3600))
```

```
    ),
    'a_float': 3.14159265,
    'an_int': 42
}
```

As you can see, we got everything back correctly. As an exercise, we would like to challenge you to write the same logic but for a `date` object, which should be simpler.

Before we move on to the next topic, a word of caution. Perhaps it is counter-intuitive, but working with `datetime` objects can be one of the trickiest things to do, so although we are pretty sure this code is doing what it is supposed to do, we want to stress that we only tested it lightly. So, if you intend to grab it and use it, please do test it thoroughly. Test for different time zones, test for daylight saving time being on and off, test for dates before the epoch, and so on. You might find that the code in this section needs some modifications to suit your cases.

I/O, streams, and requests

I/O stands for **input/output**, and it broadly refers to the communication between a computer and the outside world. There are several different types of I/O, and it is outside the scope of this chapter to explain all of them, but it's worth going through a couple of examples. The first one will introduce the `io.StringIO` class, which is an in-memory stream for text I/O. The second one instead will escape the locality of our computer, and demonstrate how to perform an HTTP request.

Using an in-memory stream

In-memory objects can be useful in a multitude of situations. Memory is much faster than a disk, it's always available, and for small amounts of data can be the perfect choice.

Let's see the first example:

```python
# io_examples/string_io.py
import io

stream = io.StringIO()
stream.write('Learning Python Programming.\n')
print('Become a Python ninja!', file=stream)

contents = stream.getvalue()
print(contents)
```

```
stream.close()
```

In the preceding code snippet, we import the `io` module from the standard library. This is a very interesting module that features many tools related to streams and I/O. One of them is `StringIO`, which is an in-memory buffer in which we're going to write two sentences, using two different methods, as we did with files in the first examples of this chapter. We can either call `StringIO.write()` or we can use `print`, telling it to direct the data to our stream.

By calling `getvalue()`, we can get the content of the stream. We then proceed to print it, and finally we close it. The call to `close()` causes the text buffer to be immediately discarded.

There is a more elegant way to write the previous code:

```
# io_examples/string_io.py
with io.StringIO() as stream:
    stream.write('Learning Python Programming.\n')
    print('Become a Python ninja!', file=stream)
    contents = stream.getvalue()
    print(contents)
```

Yes, it is again a context manager. Like the built-in `open()`, `io.StringIO()` works well within a context manager block. Notice the similarity with `open`: in this case too, we don't need to manually close the stream.

When running the script, the output is:

```
$ python string_io.py
Learning Python Programming.
Become a Python ninja!
```

Let's now proceed with the second example.

Making HTTP requests

In this section, we explore two examples on HTTP requests. We will use the `requests` library for these examples, which you can install with `pip`, and it is included in the requirements file for this chapter.

We're going to perform HTTP requests against the httpbin.org (`http://httpbin.org/`) API, which, interestingly, was developed by Kenneth Reitz, the creator of the `requests` library itself.

This library is among the most widely adopted all over the world:

```python
# io_examples/reqs.py
import requests

urls = {
    "get": "https://httpbin.org/get?t=learn+python+programming",
    "headers": "https://httpbin.org/headers",
    "ip": "https://httpbin.org/ip",
    "user-agent": "https://httpbin.org/user-agent",
    "UUID": "https://httpbin.org/uuid",
    "JSON": "https://httpbin.org/json",
}

def get_content(title, url):
    resp = requests.get(url)
    print(f"Response for {title}")
    print(resp.json())

for title, url in urls.items():
    get_content(title, url)
    print("-" * 40)
```

The preceding snippet should be simple to understand. We declare a dictionary of URLs against which we want to perform HTTP requests. We have encapsulated the code that performs the request into a tiny function, `get_content()`. As you can see, we perform a GET request (by using `requests.get()`), and we print the title and the JSON decoded version of the body of the response. Let us spend a few words on this last bit.

When we perform a request to a website, or to an API, we get back a response object, which is, very simply, what was returned by the server we performed the request against. The body of some responses from `httpbin.org` happens to be JSON encoded, so instead of getting the body as it is (by using `resp.text`) and manually decoding it calling `json.loads()` on it, we simply combine the two by leveraging the `json()` method on the response object. There are plenty of reasons why the `requests` package has become so widely adopted, and one of them is definitely its ease of use.

Now, when you perform a request in your application, you will want to have a much more robust approach in dealing with errors and so on, but for this chapter, a simple example will do. We will see more examples of requests in *Chapter 14*, *Introduction to API Development*.

Going back to our code, in the end, we run a `for` loop and get all the URLs. When you run it, you will see the result of each call printed on your console, which should look like this (prettified and trimmed for brevity):

```
$ python reqs.py
Response for get
{
    "args": {"t": "learn python programming"},
    "headers": {
        "Accept": "*/*",
        "Accept-Encoding": "gzip, deflate",
        "Host": "httpbin.org",
        "User-Agent": "python-requests/2.25.1",
        "X-Amzn-Trace-Id": "Root=1-60a42902-3b6093e26ae375244478",
    },
    "origin": "86.8.174.15",
    "url": "https://httpbin.org/get?t=learn+python+programming",
}
... rest of the output omitted ...
```

Notice that you might get a slightly different output in terms of version numbers and IPs, which is fine. Now, GET is only one of the HTTP verbs, albeit one of the most commonly used. Let us also look at how to use the POST verb. This is the type of request you make when you need to send data to the server. Every time you submit a form on the web, you're making a POST request. So, let's try to make one programmatically:

```python
# io_examples/reqs_post.py
import requests

url = 'https://httpbin.org/post'
data = dict(title='Learn Python Programming')

resp = requests.post(url, data=data)
print('Response for POST')
print(resp.json())
```

The preceding code is very similar to what we saw before, only this time we don't call `get()`, but `post()`, and because we want to send some data, we specify that in the call. The `requests` library offers much more than this. It is a project that we encourage you to check out and explore, as it's quite likely you will be using it too.

Running the previous script (and applying some prettifying magic to the output) yields the following:

```
$ python reqs_post.py
Response for POST
{
    "args": {},
    "data": "",
    "files": {},
    "form": {"title": "Learn Python Programming"},
    "headers": {
        "Accept": "*/*",
        "Accept-Encoding": "gzip, deflate",
        "Content-Length": "30",
        "Content-Type": "application/x-www-form-urlencoded",
        "Host": "httpbin.org",
        "User-Agent": "python-requests/2.25.1",
        "X-Amzn-Trace-Id": "Root=1-60a43131-5032cdbc14db751fe775",
    },
    "json": None,
    "origin": "86.8.174.15",
    "url": "https://httpbin.org/post",
}
```

Notice how the headers are now different, and we find the data we sent in the form key/value pair of the response body.

We hope these short examples are enough to get you started, especially with requests. The web changes every day, so it's worth learning the basics and then brushing up every now and then.

Persisting data on disk

In this last section of this chapter, we'll look at how to persist data on disk in three different formats. To persist data means that the data is written to non-volatile storage, like a hard drive, for example, and it is not deleted when the process that wrote it ends its life cycle. We will explore pickle and shelve, as well as a short example that will involve accessing a database using **SQLAlchemy**, perhaps the most widely adopted ORM library in the Python ecosystem.

Serializing data with pickle

The `pickle` module, from the Python standard library, offers tools to convert Python objects into byte streams, and vice versa. Even though there is a partial overlap in the API that `pickle` and `json` expose, the two are quite different. As we have seen previously in this chapter, JSON is a text format that is human readable, language independent, and supports only a restricted subset of Python data types. The `pickle` module, on the other hand, is not human readable, translates to bytes, is Python-specific, and, thanks to the wonderful Python introspection capabilities, supports a large number of data types.

Besides the above-mentioned differences between `pickle` and `json`, there are also some important security concerns that you need to be aware of if you are considering using `pickle`. *Unpickling* erroneous or malicious data from an untrusted source can be dangerous, so if we decide to adopt it in our application, we need to be extra careful.

> If you do use `pickle`, you should consider using a cryptographic signature to ensure that your pickled data has not been tampered with. We will see how to generate cryptographic signatures in Python in *Chapter 9*, *Cryptography and Tokens*.

That said, let's see it in action by means of a simple example:

```python
# persistence/pickler.py
import pickle
from dataclasses import dataclass

@dataclass
class Person:
    first_name: str
    last_name: str
    id: int

    def greet(self):
        print(f'Hi, I am {self.first_name} {self.last_name}'
              f' and my ID is {self.id}')

people = [
```

```
    Person('Obi-Wan', 'Kenobi', 123),
    Person('Anakin', 'Skywalker', 456),
]

# save data in binary format to a file
with open('data.pickle', 'wb') as stream:
    pickle.dump(people, stream)

# load data from a file
with open('data.pickle', 'rb') as stream:
    peeps = pickle.load(stream)

for person in peeps:
    person.greet()
```

In this example, we create a `Person` class using the `dataclass` decorator, which we saw in *Chapter 6*, *OOP, Decorators, and Iterators*. The only reason we wrote this example with a dataclass is to show you how effortlessly `pickle` deals with it, with no need for us to do anything we wouldn't do for a simpler data type.

The class has three attributes: `first_name`, `last_name`, and `id`. It also exposes a `greet()` method, which simply prints a hello message with the data.

We create a list of instances and save it to a file. In order to do so, we use `pickle.dump()`, to which we feed the content to be *pickled*, and the stream to which we want to write. Immediately after that, we read from that same file, using `pickle.load()` to convert the entire content of the stream back into Python objects. To make sure that the objects have been converted correctly, we call the `greet()` method on both of them. The result is the following:

```
$ python pickler.py
Hi, I am Obi-Wan Kenobi and my ID is 123
Hi, I am Anakin Skywalker and my ID is 456
```

The `pickle` module also allows you to convert to (and from) byte objects, by means of the `dumps()` and `loads()` functions (note the `s` at the end of both names). In day-to-day applications, `pickle` is usually used when we need to persist Python data that is not supposed to be exchanged with another application. One example we stumbled upon, a few years ago, was the session manager in a `flask` plugin, which pickles the session object before storing it in a Redis database. In practice, though, you are unlikely to have to deal with this library very often.

Another tool that is possibly used even less, but that proves to be very useful when you are short on resources, is `shelve`.

Saving data with shelve

A `shelf` is a persistent dictionary-like object. The beauty of it is that the values you save into a `shelf` can be any objects you can `pickle`, so you're not restricted like you would be if you were using a database. Albeit interesting and useful, the `shelve` module is used quite rarely in practice. Just for completeness, let's see a quick example of how it works:

```python
# persistence/shelf.py
import shelve

class Person:
    def __init__(self, name, id):
        self.name = name
        self.id = id

with shelve.open('shelf1.shelve') as db:
    db['obi1'] = Person('Obi-Wan', 123)
    db['ani'] = Person('Anakin', 456)
    db['a_list'] = [2, 3, 5]
    db['delete_me'] = 'we will have to delete this one...'
    print(list(db.keys()))  # 'ani', 'delete_me', 'a_list', 'obi1']

    del db['delete_me']  # gone!
    print(list(db.keys()))  # ['ani', 'a_list', 'obi1']
    print('delete_me' in db)  # False
    print('ani' in db)  # True

    a_list = db['a_list']
    a_list.append(7)
    db['a_list'] = a_list
    print(db['a_list'])  # [2, 3, 5, 7]
```

Apart from the wiring and the boilerplate around it, this example resembles an exercise with dictionaries. We create a simple `Person` class and then we open a `shelve` file within a context manager. As you can see, we use the dictionary syntax to store four objects: two `Person` instances, a list, and a string. If we print the `keys`, we get a list containing the four keys we used. Immediately after printing it, we delete the (aptly named) `delete_me` key/value pair from the shelf. Printing the `keys` again shows the deletion has succeeded. We then test a couple of keys for membership and, finally, we append number 7 to `a_list`. Notice how we have to extract the list from the shelf, modify it, and save it again.

If this behavior is undesired, there is something we can do:

```python
# persistence/shelf.py
with shelve.open('shelf2.shelve', writeback=True) as db:
    db['a_list'] = [11, 13, 17]
    db['a_list'].append(19)  # in-place append!
    print(db['a_list'])  # [11, 13, 17, 19]
```

By opening the shelf with `writeback=True`, we enable the `writeback` feature, which allows us to simply append to `a_list` as if it actually was a value within a regular dictionary. The reason why this feature is not active by default is that it comes with a price that you pay in terms of memory consumption and slower closing of the shelf.

Now that we have paid homage to the standard library modules related to data persistence, let's take a look at one of the most widely adopted ORMs in the Python ecosystem: SQLAlchemy.

Saving data to a database

For this example, we are going to work with an in-memory database, which will make things simpler for us. In the source code of the book, we have left a couple of comments to show you how to generate a SQLite file, so we hope you'll explore that option as well.

> You can find a free database browser for SQLite at `https://dbeaver.io`. DBeaver is a free multi-platform database tool for developers, database administrators, analysts, and all people who need to work with databases. It supports all popular databases: MySQL, PostgreSQL, SQLite, Oracle, DB2, SQL Server, Sybase, MS Access, Teradata, Firebird, Apache Hive, Phoenix, Presto, and so on.

Before we dive into the code, allow us to briefly introduce the concept of a relational database.

A relational database is a database that allows you to save data following the **relational model**, invented in 1969 by Edgar F. Codd. In this model, data is stored in one or more tables. Each table has rows (also known as **records**, or **tuples**), each of which represents an entry in the table. Tables also have columns (also known as **attributes**), each of which represents an attribute of the records. Each record is identified through a unique key, more commonly known as the **primary key**, which is the union of one or more columns in the table. To give you an example: imagine a table called `Users`, with columns `id`, `username`, `password`, `name`, and `surname`.

Such a table would be perfect to contain users of our system; each row would represent a different user. For example, a row with the values 3, `fab`, `my_wonderful_pwd`, `Fabrizio`, and `Romano` would represent Fabrizio's user in the system.

The reason why the model is called *relational* is because you can establish relations between tables. For example, if you added a table called `PhoneNumbers` to our fictitious database, you could insert phone numbers into it, and then, through a relation, establish which phone number belongs to which user.

In order to query a relational database, we need a special language. The main standard is called **SQL**, which stands for **Structured Query Language**. It is born out of something called **relational algebra**, which is a family of algebras used to model data stored according to the relational model and perform queries on it. The most common operations you can perform usually involve filtering on the rows or columns, joining tables, aggregating the results according to some criteria, and so on. To give you an example in English, a query on our imaginary database could be: *Fetch all users (username, name, surname) whose username starts with "m", who have at most one phone number*. In this query, we are asking for a subset of the columns in the `User` table. We are filtering on users by taking only those whose username starts with the letter *m*, and even further, only those who have at most one phone number.

> Back in the days when Fabrizio was a student in Padova, he spent a whole semester learning both the relational algebra semantics and standard SQL (among other things). If it wasn't for a major bicycle accident he had the day of the exam, he would say that this was one of the most fun exams he ever had to take!

Now, each database comes with its own *flavor* of SQL. They all respect the standard to some extent, but none fully do, and they are all different from one another in some respects. This poses an issue in modern software development. If our application contains SQL code, it is quite likely that if we decided to use a different database engine, or maybe a different version of the same engine, we would find our SQL code needs amending.

This can be quite painful, especially since SQL queries can become very complicated quite quickly. In order to alleviate this pain a little, computer scientists have created code that maps objects of a programming language to tables of a relational database. Unsurprisingly, the name of such a tool is **Object-Relational Mapping** (**ORM**).

In modern application development, you would normally start interacting with a database by using an ORM, and should you find yourself in a situation where you can't perform a query you need to perform through the ORM, you would then resort to using SQL directly. This is a good compromise between having no SQL at all, and using no ORM, which ultimately means specializing the code that interacts with the database, with the aforementioned disadvantages.

In this section, we would like to show an example that leverages SQLAlchemy, one of the most popular third-party Python ORMs. You will have to `pip install` it into the virtual environment for this chapter. We are going to define two models (`Person` and `Address`), each of which maps to a table, and then we're going to populate the database and perform a few queries on it.

Let's start with the model declarations:

```python
# persistence/alchemy_models.py
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import (
    Column, Integer, String, ForeignKey, create_engine)
from sqlalchemy.orm import relationship
```

At the beginning, we import some functions and types. The first thing we need to do then is to create an engine. This engine tells SQLAlchemy about the type of database we have chosen for our example, and how to connect to it:

```python
# persistence/alchemy_models.py
engine = create_engine('sqlite:///:memory:')
Base = declarative_base()

class Person(Base):
    __tablename__ = 'person'

    id = Column(Integer, primary_key=True)
    name = Column(String)
    age = Column(Integer)

    addresses = relationship(
        'Address',
        back_populates='person',
        order_by='Address.email',
        cascade='all, delete-orphan'
    )

    def __repr__(self):
        return f'{self.name}(id={self.id})'

class Address(Base):
    __tablename__ = 'address'

    id = Column(Integer, primary_key=True)
```

```python
    email = Column(String)
    person_id = Column(ForeignKey('person.id'))
    person = relationship('Person', back_populates='addresses')

    def __str__(self):
        return self.email
    __repr__ = __str__

Base.metadata.create_all(engine)
```

Each model then inherits from the `Base` table, which in this example simply consists of the default, returned by `declarative_base()`. We define `Person`, which maps to a table called `person`, and exposes the attributes `id`, `name`, and `age`. We also declare a relationship with the `Address` model, by stating that accessing the `addresses` attribute will fetch all the entries in the `address` table that are related to the particular `Person` instance we're dealing with. The `cascade` option affects how creation and deletion work, but it is a more advanced concept, so we suggest you ignore it for now and maybe investigate more later on.

The last thing we declare is the `__repr__()` method, which provides us with the official string representation of an object. This is supposed to be a representation that can be used to completely reconstruct the object, but in this example, we simply use it to provide something in output. Python redirects `repr(obj)` to a call to `obj.__repr__()`.

We also declare the `Address` model, which will contain email addresses, and a reference to the person they belong to. You can see the `person_id` and `person` attributes are both about setting a relation between the `Address` and `Person` instances. Note also how we declare the `__str__()` method on `Address`, and then assign an alias to it, called `__repr__()`. This means that calling either `repr()` or `str()` on `Address` objects will ultimately result in calling the `__str__()` method. This is quite a common technique in Python, used to avoid duplicating the same code, so we took the opportunity to show it to you here.

On the last line, we tell the engine to create tables in the database according to our models.

> The `create_engine()` function supports a parameter called `echo`, which can be set to `True`, `False`, or the string `"debug"`, to enable different levels of logging of all statements and the `repr()` of their parameters. Please refer to the official SQLAlchemy documentation for further information.

A deeper understanding of this code would require more space than we can afford, so we encourage you to read up on **database management systems** (**DBMS**), SQL, relational algebra, and SQLAlchemy.

Now that we have our models, let's use them to persist some data!

Take a look at the following example:

```python
# persistence/alchemy.py
from alchemy_models import Person, Address, engine
from sqlalchemy.orm import sessionmaker

Session = sessionmaker(bind=engine)
session = Session()
```

First we create `session`, which is the object we use to manage the database. Next, we proceed by creating two people:

```python
anakin = Person(name='Anakin Skywalker', age=32)
obi1 = Person(name='Obi-Wan Kenobi', age=40)
```

We then add email addresses to both of them, using two different techniques. One assigns them to a list, and the other one simply appends them:

```python
obi1.addresses = [
    Address(email='obi1@example.com'),
    Address(email='wanwan@example.com'),
]

anakin.addresses.append(Address(email='ani@example.com'))
anakin.addresses.append(Address(email='evil.dart@example.com'))
anakin.addresses.append(Address(email='vader@example.com'))
```

We haven't touched the database yet. It's only when we use the `session` object that something actually happens in it:

```python
session.add(anakin)
session.add(obi1)
session.commit()
```

Adding the two `Person` instances is enough to also add their addresses (this is thanks to the cascading effect). Calling `commit()` is what actually tells SQLAlchemy to commit the transaction and save the data in the database. A **transaction** is an operation that provides something like a sandbox, but in a database context.

As long as the transaction hasn't been committed, we can roll back any modification we have done to the database, and by doing so, revert to the state we were in before starting the transaction itself. SQLAlchemy offers more complex and granular ways to deal with transactions, which you can study in its official documentation, as it is quite an advanced topic. We now query for all the people whose name starts with `Obi` by using `like()`, which hooks to the `LIKE` operator in SQL:

```python
obi1 = session.query(Person).filter(
    Person.name.like('Obi%')
).first()
print(obi1, obi1.addresses)
```

We take the first result of that query (we know we only have Obi-Wan anyway), and print it. We then fetch `anakin` by using an exact match on his name, just to show you a different way of filtering:

```python
anakin = session.query(Person).filter(
    Person.name=='Anakin Skywalker'
).first()

print(anakin, anakin.addresses)
```

We then capture Anakin's ID, and delete the `anakin` object from the global frame (this does not delete the entry from the database):

```python
anakin_id = anakin.id
del anakin
```

The reason we do this is because we want to show you how to fetch an object by its ID. Before we do that, we write the `display_info()` function, which we will use to display the full content of the database (fetched starting from the addresses, in order to demonstrate how to fetch objects by using a relation attribute in SQLAlchemy):

```python
def display_info():
    # get all addresses first
    addresses = session.query(Address).all()

    # display results
    for address in addresses:
        print(f'{address.person.name} <{address.email}>')

    # display how many objects we have in total
    print('people: {}, addresses: {}'.format(
```

```
        session.query(Person).count(),
        session.query(Address).count())
    )
```

The `display_info()` function prints all the addresses, along with the respective person's name, and, at the end, produces a final piece of information regarding the number of objects in the database. We call the function, then we fetch and delete `anakin`. Finally, we display the info again, to verify that he has actually disappeared from the database:

```
display_info()

anakin = session.query(Person).get(anakin_id)
session.delete(anakin)
session.commit()

display_info()
```

The output of all these snippets run together is the following (for your convenience, we have separated the output into four blocks, to reflect the four blocks of code that actually produce that output):

```
$ python alchemy.py
Obi-Wan Kenobi(id=2) [obi1@example.com, wanwan@example.com]

Anakin Skywalker(id=1) [
    ani@example.com, evil.dart@example.com, vader@example.com
]

Anakin Skywalker <ani@example.com>
Anakin Skywalker <evil.dart@example.com>
Anakin Skywalker <vader@example.com>
Obi-Wan Kenobi <obi1@example.com>
Obi-Wan Kenobi <wanwan@example.com>
people: 2, addresses: 5

Obi-Wan Kenobi <obi1@example.com>
Obi-Wan Kenobi <wanwan@example.com>
people: 1, addresses: 2
```

As you can see from the last two blocks, deleting `anakin` has deleted one `Person` object and the three addresses associated with it. Again, this is due to the fact that cascading took place when we deleted `anakin`.

This concludes our brief introduction to data persistence. It is a vast and, at times, complex domain that we encourage you to explore, learning as much theory as possible. Lack of knowledge or proper understanding, when it comes to database systems, can really bite.

Summary

In this chapter, we have explored working with files and directories. We have learned how to open files for reading and writing and how to do that more elegantly by using context managers. We also explored directories: how to list their content, both recursively and not. We also learned about paths, which are the gateway to accessing both files and directories.

We then briefly saw how to create a ZIP archive and extract its content. The source code of the book also contains an example with a different compression format: `tar.gz`.

We talked about data interchange formats, and have explored JSON in some depth. We had some fun writing custom encoders and decoders for specific Python data types.

Then we explored I/O, both with in-memory streams and HTTP requests.

And finally, we saw how to persist data using `pickle`, `shelve`, and the SQLAlchemy ORM library.

You should now have a pretty good idea of how to deal with files and data persistence, and we hope you will take the time to explore these topics in much more depth by yourself.

The next chapter will look at cryptography and tokens.

9
Cryptography and Tokens

"Three may keep a secret, if two of them are dead."

– Benjamin Franklin, Poor Richard's Almanack

In this short chapter, we are going to give you a brief overview of the cryptographic services offered by the Python standard library. We are also going to touch upon JSON Web Tokens, an interesting standard for representing claims securely between two parties.

In particular, we are going to explore the following:

- Hashlib
- HMAC
- Secrets
- JSON Web Tokens with PyJWT, which seems to be the most popular Python library for dealing with JWTs

Let's start by taking a moment to talk about cryptography and why it is so important.

The need for cryptography

It is estimated that, in 2021, over 4 billion people worldwide use the internet. Every year, more people are using online banking services, shopping online, or just talking to friends and family on social media. All these people expect that their money will be safe, their transactions secure, and their conversations private.

Therefore, if you are an application developer, you have to take security very, very seriously. It doesn't matter how small or apparently insignificant your application is: security should always be a concern for you.

Security in information technology is achieved by employing several different means, but by far the most important one is cryptography. Almost everything you do with your computer or phone should include a layer where cryptography takes place. For example, cryptography is used to secure online payments, to transfer messages over a network in a way that even if someone intercepts them, they won't be able to read them, and to encrypt your files when you back them up in the cloud.

The purpose of this chapter is not to teach you all the intricacies of cryptography — there are entire books dedicated to the subject. Instead, we will show you how you can use the tools that Python offers you to create digests, tokens, and in general, to be on the safe(r) side when you need to implement something cryptography-related. As you read this chapter, it's worth bearing in mind that there is much more to cryptography than just encrypting and decrypting data; in fact, you won't find any examples of encryption or decryption in the entire chapter!

Useful guidelines

Always remember the following rules:

- *Rule number one*: Do not attempt to create your own hash or encryption functions. Simply don't. Use tools and functions that are there already. It is incredibly tough to come up with a good, solid, robust algorithm to do hashing or encryption, so it's best to leave it to professional cryptographers.
- *Rule number two*: Follow rule number one.

Those are the only two rules you need. Apart from them, it is very useful to understand cryptography, so try and learn as much as you can about this subject. There is plenty of information on the web, but for your convenience, we'll put some useful references at the end of this chapter.

Now, let's dig into the first of the standard library modules we want to show you: `hashlib`.

Hashlib

This module provides access to a variety of cryptographic hash algorithms. These are mathematical functions that take a message of any size and produce a fixed size result, which is referred to as a **hash** or **digest**. Cryptographic hashes have many uses, from verifying data integrity to securely storing and verifying passwords.

Ideally, cryptographic hash algorithms should be:

- **Deterministic**: The same message should always produce the same hash.

- **Irreversible**: It should not be feasible to determine the original message from the hash.

- **Collision resistant**: It should be hard to find two different messages that produce the same hash.

These properties are crucial for the secure application of hashes. For example, it is considered imperative that passwords are only stored in hashed form. The irreversibility property ensures that even if a data breach occurs and an attacker gets hold of your password database, it would not be feasible for them to obtain the original passwords. Having the passwords stored only as hashes means that the only way to verify a user's password when they log in is to compute the hash of the password they provided and compare it against the stored hash. Of course, this will not work if the hash algorithm is not deterministic. Collision resistance is important when hashes are used for data integrity verification. If we are using a hash to check that a piece of data was not tampered with, an attacker who could find a hash collision could modify the data without changing the hash, tricking us into thinking the data was not changed.

The exact set of algorithms that are actually available through `hashlib` vary depending on the underlying libraries used on your platform. Some algorithms, however, are guaranteed to be present on all systems. Let's see how to find out what is available (note that your results might be different from ours):

```
# hlib.py
>>> import hashlib
>>> hashlib.algorithms_available
{'mdc2', 'sha224', 'whirlpool', 'sha1', 'sha3_512', 'sha512_256',
 'sha256', 'md4', 'sha384', 'blake2s', 'sha3_224', 'sha3_384',
 'shake_256', 'blake2b', 'ripemd160', 'sha512', 'md5-sha1',
 'shake_128', 'sha3_256', 'sha512_224', 'md5', 'sm3'}
>>> hashlib.algorithms_guaranteed
{'blake2s', 'md5', 'sha224', 'sha3_512', 'shake_256', 'sha3_256',
 'shake_128', 'sha256', 'sha1', 'sha512', 'blake2b', 'sha3_384',
 'sha384', 'sha3_224'}
```

By opening a Python shell, we can get the set of available algorithms for our system. If our application has to talk to third-party applications, it's always best to pick an algorithm out of the guaranteed set, though, as that means every platform actually supports them. Notice that a lot of them start with *sha*, which stands for *secure hash algorithm*.

Let's keep going in the same shell: we are going to create a hash for the byte string
`b'Hash me now!'`:

```
>>> h = hashlib.blake2b()
>>> h.update(b'Hash me')
>>> h.update(b' now!')
>>> h.hexdigest()
'56441b566db9aafcf8cdad3a4729fa4b2bfaab0ada36155ece29f52ff70e1e9d'
'7f54cacfe44bc97c7e904cf79944357d023877929430bc58eb2dae168e73cedf'
>>> h.digest()
b'VD\x1bVm\xb9\xaa\xfc\xf8\xcd\xad:G)\xfaK+\xfa\xab\n\xda6\x15^'
b'\xce)\xf5/\xf7\x0e\x1e\x9d\x7fT\xca\xcf\xe4K\xc9|~\x90L\xf7'
b'\x99D5}\x028w\x92\x940\xbcX\xeb-\xae\x16\x8es\xce\xdf'
>>> h.block_size
128
>>> h.digest_size
64
>>> h.name
'blake2b'
```

We have used the `blake2b()` cryptographic function, which is quite sophisticated
and was added in Python 3.6. After creating the hash object `h`, we update its message
in two steps. Not that we need to here, but sometimes we need to hash data that is
not available all at once, so it's good to know we can do it in steps.

Once we have added the entire message, we get the hexadecimal representation of
the digest. This will use two characters per byte (as each character represents 4 bits,
which is half a byte). We also get the byte representation of the digest, and then we
inspect its details: it has a block size (the internal block size of the hash algorithm in
bytes) of 128 bytes, a digest size (the size of the resulting hash in bytes) of 64 bytes,
and a name.

Let's see what we get if, instead of the `blake2b()` function, we use `sha256()`:

```
>>> hashlib.sha256(b'Hash me now!').hexdigest()
'10d561fa94a89a25ea0c7aa47708bdb353bbb062a17820292cd905a3a60d6783'
```

The resulting hash is shorter (and therefore less secure). Notice that we can construct
the hash object with the message and compute the digest in one line.

Hashing is a very interesting topic, and of course, the simple examples we've seen
so far are just the start. The `blake2b()` function allows us a great deal of flexibility
thanks to a number of parameters that can be adjusted. This means that it can be
adapted for different applications or adjusted to protect against particular types of
attacks.

Here, we will just briefly discuss one of these parameters; for the full details, please refer to the official documentation at `https://docs.python.org/3.7/library/hashlib.html`. The `person` parameter is quite interesting. It is used to *personalize* the hash, forcing it to produce different digests for the same message. This can help to improve security when the same hash function is used for different purposes within the same application:

```
>>> import hashlib
>>> h1 = hashlib.blake2b(b'Important data', digest_size=16,
...                      person=b'part-1')
>>> h2 = hashlib.blake2b(b'Important data', digest_size=16,
...                      person=b'part-2')
>>> h3 = hashlib.blake2b(b'Important data', digest_size=16)
>>> h1.hexdigest()
'c06b9af95d5aa6307e7e3fd025a15646'
>>> h2.hexdigest()
'9cb03be8f3114d0f06bddaedce2079c4'
>>> h3.hexdigest()
'7d35308ca3b042b5184728d2b1283d0d'
```

Here we've also used the `digest_size` parameter to get hashes that are only 16 bytes long.

General-purpose hash functions, like `blake2b()` or `sha256()`, are not suitable for securely storing passwords. General-purpose hash functions are quite fast to compute on modern computers, which makes it feasible for an attacker to reverse the hash by **brute force** (trying millions of possibilities per second until they find a match). Key derivation algorithms like `pbkdf2_hmac()` are designed to be slow enough to make such brute-force attacks infeasible. The `pbkdf2_hmac()` key derivation algorithm achieves this by using many repeated applications of a general-purpose hash function (the number of iterations can be specified as a parameter). As computers get more and more powerful, it is important to increase the number of iterations we do over time, otherwise the likelihood of a successful brute-force attack on our data increases as time passes.

Good password hash functions should also use **salt**. Salt is a random piece of data used to initialize the hash function; this randomizes the output of the algorithm and protects against attacks where hashes are compared to tables of known hashes. The `pbkdf2_hmac()` function supports salting via a required `salt` parameter.

Here's how you can use `pbkdf2_hmac()` to hash a password:

```
>>> import os
>>> dk = hashlib.pbkdf2_hmac('sha256', b'Password123',
```

```
...      salt=os.urandom(16), iterations=100000
... )
>>> dk.hex()
'f8715c37906df067466ce84973e6e52a955be025a59c9100d9183c4cbec27a9e'
```

Notice that we have used `os.urandom()` to provide a 16-byte random salt, as recommended by the documentation.

We encourage you to explore and experiment with this module, as sooner or later you will have to use it. Now, let's move on to the `hmac` module.

HMAC

This module implements the **HMAC** algorithm, as described by RFC 2104 (`https://tools.ietf.org/html/rfc2104.html`). HMAC (which stands for **hash-based message authentication code** or **keyed-hash message authentication code**, depending on who you ask) is a widely used mechanism for authenticating messages and verifying that they have not been tampered with.

The algorithm combines a message with a secret key and generates a hash of the combination. This hash is referred to as a **message authentication code** (**MAC**) or **signature**. The signature is stored or transmitted along with the message. At a later time, you can verify that the message has not been tampered with by re-computing the signature using the same secret key and comparing it to the previously computed signature. The secret key must be carefully protected, otherwise an attacker with access to the key would be able to modify the message and replace the signature, thereby defeating the authentication mechanism.

Let's see a small example of how to compute a message authentication code:

```python
# hmc.py
import hmac
import hashlib

def calc_digest(key, message):
    key = bytes(key, 'utf-8')
    message = bytes(message, 'utf-8')
    dig = hmac.new(key, message, hashlib.sha256)
    return dig.hexdigest()

mac = calc_digest('secret-key', 'Important Message')
```

The `hmac.new()` function takes a secret `key`, a `message`, and the hash algorithm to use and returns an `hmac` object, which has a similar interface to the hash objects from `hashlib`. The key must be a `bytes` or `bytearray` object and the message can be any `bytes`-like object. Therefore, we convert our `key` and the `message` into bytes before creating an `hmac` instance (`dig`), which we use to get a hexadecimal representation of the hash.

We'll see a bit more of how HMAC signatures can be used later in this chapter, when we talk about JWTs. Before that, however, we'll take a quick look at the `secrets` module.

Secrets

This small module was added in Python 3.6 and deals with three things: random numbers, tokens, and digest comparison. It uses the most secure random number generators provided by the underlying operating system to generate tokens and random numbers suitable for use in cryptographic applications. Let's have a quick look at what it provides.

Random numbers

We can use three functions in order to deal with random numbers:

```python
# secrs/secr_rand.py
import secrets
print(secrets.choice('Choose one of these words'.split()))
print(secrets.randbelow(10 ** 6))
print(secrets.randbits(32))
```

The first one, `choice()`, picks an element at random from a non-empty sequence. The second, `randbelow()`, generates a random integer between 0 and the argument you call it with, and the third, `randbits()`, generates an integer with the given number of random bits in it. Running that code produces the following output (which will of course be different every time it is run):

```
$ python secr_rand.py
one
504156
3172492450
```

You should use these functions instead of those from the `random` module whenever you need randomness in the context of cryptography, as these are specially designed for this task. Let's see what the module gives us for tokens.

Token generation

Again, we have three functions for generating tokens, each in a different format. Let's see the example:

```python
# secrs/secr_rand.py
print(secrets.token_bytes(16))
print(secrets.token_hex(32))
print(secrets.token_urlsafe(32))
```

The `token_bytes()` function simply returns a random byte string containing the specified number of bytes (16, in this example). The other two do the same, but `token_hex()` returns a token in hexadecimal format, and `token_urlsafe()` returns a token that only contains characters suitable for being included in a URL. Let's see the output (which is a continuation from the previous run):

```
b'\xda\x863\xeb\xbb|\x8fk\x9b\xbd\x14Q\xd4\x8d\x15}'
9f90fd042229570bf633e91e92505523811b45e1c3a72074e19bbeb2e5111bf7
bl4qz_Av7QNvPEqZtKsLuTOUsNLFmXW3O03pn50leiY
```

Let's see how we can use these tools to write our own random password generator:

```python
# secrs/secr_gen.py
import secrets
from string import digits, ascii_letters

def generate_pwd(length=8):
    chars = digits + ascii_letters
    return ''.join(secrets.choice(chars) for c in range(length))

def generate_secure_pwd(length=16, upper=3, digits=3):
    if length < upper + digits + 1:
        raise ValueError('Nice try!')
    while True:
        pwd = generate_pwd(length)
        if (any(c.islower() for c in pwd)
            and sum(c.isupper() for c in pwd) >= upper
            and sum(c.isdigit() for c in pwd) >= digits):
            return pwd
```

```
print(generate_secure_pwd())
print(generate_secure_pwd(length=3, upper=1, digits=1))
```

Our `generate_pwd()` function simply generates a random string of a given length by joining together `length` characters picked at random from a string that contains all the letters of the alphabet (lowercase and uppercase), and the 10 decimal digits.

Then, we define another function, `generate_secure_pwd()`, that simply keeps calling `generate_pwd()` until the random string we get matches some simple requirements. The password must be `length` characters long, have at least one lowercase character, `upper` uppercase characters, and `digits` digits.

If the total number of uppercase characters, lowercase characters, and digits specified by the parameters is greater than the length of the password we are generating, we can never satisfy the conditions. So, in order to avoid getting stuck in an infinite loop, we have put a check clause in the first line of the body, and we raise a `ValueError` if the requirements cannot be satisfied.

The body of the `while` loop is straightforward: first we generate the random password, and then we verify the conditions by using `any()` and `sum()`. The `any` function returns `True` if any of the items in the iterable it's called with evaluate to `True`. The use of `sum()` is actually slightly trickier here, in that it exploits polymorphism. As you may recall from *Chapter 2, Built-In Data Types*, the `bool` type is a subclass of `int`, therefore when summing on an iterable of `True` and `False` values, they will automatically be interpreted as integers (with the values 1 and 0) by the `sum()` function. This is an example of **polymorphism**, which we briefly discussed in *Chapter 6, OOP, Decorators, and Iterators*.

Running the example produces the following result:

```
$ python secr_gen.py
nsL5voJnCi7Ote3F
J5e
```

That second password is probably not very secure...

One common use of random tokens is in password reset URLs for websites. Let's see an example of how we can generate such a URL:

```python
# secrs/secr_reset.py
import secrets

def get_reset_pwd_url(token_length=16):
    token = secrets.token_urlsafe(token_length)
```

```
    return f'https://example.com/reset-pwd/{token}'

print(get_reset_pwd_url())
```

This function is so easy we will only show you the output:

```
$ python secr_reset.py
https://example.com/reset-pwd/dfVPEPl_pCkQ8YNV4er-UQ
```

Digest comparison

This is probably quite surprising, but the `secrets` module also provides a `compare_digest(a, b)` function, which is the equivalent of comparing two digests by simply doing `a == b`. So, why would we need that function? It's because it has been designed to prevent timing attacks. These kinds of attacks can infer information about where the two digests start being different, according to the time it takes for the comparison to fail. So, `compare_digest()` prevents this attack by removing the correlation between time and failures. We think this is a brilliant example of how sophisticated attacking methods can be. If you raised your eyebrows in astonishment, maybe now it's clearer why we said never to implement cryptography functions by yourself.

This brings us to the end of our tour of the cryptographic services in the Python standard library. Now, let's move on to a different type of token: JWTs.

JSON Web Tokens

A **JSON Web Token**, or **JWT**, is a JSON-based open standard for creating tokens that assert some number of **claims**. JWTs are frequently used as authentication tokens. In this context, the claims are typically statements about the identity and permissions of an authenticated user. The tokens are cryptographically signed, which makes it possible to verify that the content of the token has not been modified since it was issued. You can learn all about this technology on the website (`https://jwt.io/`).

This type of token is comprised of three sections, separated by a dot, in the format *A.B.C*. *B* is the payload, which is where we put the claims. *C* is the signature, which is used to verify the validity of the token, and *A* is a header, which identifies the token as a JWT, and indicates the algorithm used to compute the signature. *A*, *B*, and *C* are all encoded with a URL-safe Base64 encoding (which we'll refer to as Base64URL). The Base64URL encoding makes it possible to use JWTs as part of URLs (typically as query parameters); however, JWTs do also appear in many other places, including HTTP headers.

> 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 total of 64 symbols. Base64 is used, for example, to encode images attached in an email. It happens seamlessly, so the vast majority of people are completely oblivious to this fact. Base64URL is a variant of Base64 encoding where the + and / characters (which have specific meanings in the context of a URL) are replaced with - and _. The = character (which is used for padding in Base64) also has a special meaning within URLs and is omitted in Base64URL.

The way this type of token works is slightly different to what we have seen so far in this chapter. In fact, the information that the token carries is always visible. You just need to decode *A* and *B* from Base64URL to get the algorithm and the payload. The security lies in part *C*, which is an HMAC signature of the header and payload. If you try to modify either the *A* or *B* part by editing the header or the payload, encoding it back to Base64URL, and replacing it in the token, the signature won't match, and therefore the token will be invalid.

This means that we can build a payload with claims such as *logged in as admin*, or something along those lines, and as long as the token is valid, we know we can trust that that user is actually logged in as an admin.

> When dealing with JWTs, you want to make sure you have researched how to handle them safely. Things like not accepting unsigned tokens, or restricting the list of algorithms you use to encode and decode, as well as other security measures, are very important and you should take the time to investigate and learn them.
>
> For this part of the code, you will have to have the `PyJWT` and `cryptography` Python packages installed. As always, you will find them in the requirements of the source code for this chapter.

Let's start with a simple example:

```python
# jwt/tok.py
import jwt

data = {'payload': 'data', 'id': 123456789}

token = jwt.encode(data, 'secret-key')
algs = ['HS256', 'HS512']
```

```
data_out = jwt.decode(token, 'secret-key', algorithms=algs)
print(token)
print(data_out)
```

We define the `data` payload, which contains an ID and some payload data. We create a token using the `jwt.encode()` function, which takes the payload and a secret key. The secret key is used to generate the HMAC signature of the token header and payload. Next, we decode the token again, specifying the signature algorithms that we are willing to accept. The default algorithm used to calculate the token is `HS256`; in this example, we accept either `HS256` or `HS512` when decoding (if the token had been generated using a different algorithm, it would be rejected with an exception). Let's see the output:

```
$ python tok.py
b'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
eyJwYXlsb2FkIjoiZGF0YSIsImlkIjoxMjM0NTY3ODl9.WFRY-uoACMoNYX97PXXjEfXFQO
1rCyFCyiwxzOVMn40'
{'payload': 'data', 'id': 123456789}
```

As you can see, the token is a binary string of Base64URL-encoded pieces of data. We called `jwt.decode()`, providing the correct secret key. If we had supplied the wrong key, we would have gotten an error, since the signature can only be verified with the same secret that was used to generate it.

> JWTs are often used to transmit information between two parties. For example, authentication protocols that allow websites to rely on third-party identity providers to authenticate users often use JWTs. In such cases, the secret key used to sign tokens needs to be shared between the two parties. Therefore, it is often referred to as a **shared secret**.
>
> Care must be taken to protect the shared secret, since anyone with access to it can generate valid tokens.

Sometimes, you might want to be able to inspect the content of the token without verifying the signature first. You can do so by simply calling `decode()` this way:

```
# jwt/tok.py
jwt.decode(token, options={'verify_signature': False})
```

This is useful, for example, when values in the token payload are needed to recover the secret key, but that technique is quite advanced so we won't be spending time on it in this context. Instead, let's see how we can specify a different algorithm for computing the signature:

```
# jwt/tok.py
token512 = jwt.encode(data, 'secret-key', algorithm='HS512')
data_out = jwt.decode(token512, 'secret-key', algorithms=['HS512'])
print(data_out)
```

Here we have used the HS512 algorithm to generate the token and on decoding specified that we would only accept tokens generated using the HS512 algorithm. The output is our original payload dictionary.

Now, while you are free to put whatever you want in the token payload, there are some claims that have been standardized, and they enable you to have a great deal of control over the token.

Registered claims

The JWT standard defines the following official **registered claims**:

- iss: The *issuer* of the token
- sub: The *subject* information about the party this token is carrying information about
- aud: The *audience* for the token
- exp: The *expiration time*, after which the token is considered to be invalid
- nbf: The *not before (time)*, or the time before which the token is not considered to be valid yet
- iat: The time at which the token was *issued*
- jti: The token *ID*

Claims that are not defined in the standard can be categorized as public or private:

- **Public**: Claims that are publicly allocated for a particular purpose. Public claim names can be reserved by registering them with the IANA JSON Web Token Claims Registry. Alternatively, the claims should be named in a way that ensures that they do not clash with any other public or official claim names (one way of achieving this could be to prepend a registered domain name to the claim name).

- **Private**: Any other claims that do not fall under the above categories are referred to as private claims. The meaning of such claims is typically defined within the context of a particular application and they are meaningless outside that context. To avoid ambiguity and confusion, care must be taken to avoid name clashes.

To learn all about claims, please refer to the official website. Now, let's see a couple of code examples involving a subset of these claims.

Time-related claims

Let's see how we might use the claims related to time:

```python
# jwt/claims_time.py
from datetime import datetime, timedelta, timezone
from time import sleep, time
import jwt

iat = datetime.now(tz=timezone.utc)
nfb = iat + timedelta(seconds=1)
exp = iat + timedelta(seconds=3)
data = {'payload': 'data', 'nbf': nfb, 'exp': exp, 'iat': iat}

def decode(token, secret):
    print(time())
    try:
        print(jwt.decode(token, secret, algorithms=['HS256']))
    except (
        jwt.ImmatureSignatureError, jwt.ExpiredSignatureError
    ) as err:
        print(err)
        print(type(err))

secret = 'secret-key'
token = jwt.encode(data, secret)

decode(token, secret)
sleep(2)
decode(token, secret)
sleep(2)
decode(token, secret)
```

In this example, we set the issued at (`iat`) claim to the current UTC time (**UTC** stands for **Coordinated Universal Time**). We then set the not before (`nbf`) and expire time (`exp`) at 1 and 3 seconds from now, respectively. We define a `decode()` helper function that reacts to a token not being valid yet, or being expired, by trapping the appropriate exceptions, and then we call it three times, interspersed by two calls to `sleep()`.

This way, we will try to decode the token before it is valid, then when it is valid, and finally after it has expired. This function also prints a useful timestamp before attempting to decode the token. Let's see how it goes (blank lines have been added for readability):

```
$ python jwt/claims_time.py
1631043839.6459477
The token is not yet valid (nbf)
<class 'jwt.exceptions.ImmatureSignatureError'>

1631043841.6480813
{'payload': 'data', 'nbf': 1631043840, 'exp': 1631043842, 'iat':
1631043839}

1631043843.6498601
Signature has expired
<class 'jwt.exceptions.ExpiredSignatureError'>
```

As you can see, it all executed as expected. We get descriptive messages from the exceptions and get the original payload back when the token is actually valid.

Authentication-related claims

Let's see another quick example involving the issuer (`iss`) and audience (`aud`) claims. The code is conceptually very similar to the previous example, and we're going to exercise it in the same way:

```python
# jwt/claims_auth.py
import jwt

data = {'payload': 'data', 'iss': 'hein', 'aud': 'learn-python'}

secret = 'secret-key'
token = jwt.encode(data, secret)

def decode(token, secret, issuer=None, audience=None):
    try:
        print(jwt.decode(token, secret, issuer=issuer,
                         audience=audience, algorithms=["HS256"]))
    except (
        jwt.InvalidIssuerError, jwt.InvalidAudienceError
    ) as err:
        print(err)
```

```python
        print(type(err))

decode(token, secret)

# not providing the issuer won't break
decode(token, secret, audience='learn-python')

# not providing the audience will break
decode(token, secret, issuer='hein')

# both will break
decode(token, secret, issuer='wrong', audience='learn-python')
decode(token, secret, issuer='hein', audience='wrong')

decode(token, secret, issuer='hein', audience='learn-python')
```

As you can see, this time, we have specified `issuer` and `audience`. It turns out that if we don't provide the issuer when decoding the token, it won't cause the decoding to break. However, providing the wrong issuer will actually break decoding. On the other hand, both failing to provide the audience, or providing the wrong audience, will break decoding.

As in the previous example, we have written a custom `decode()` function that reacts to the appropriate exceptions. See if you can follow along with the calls and the relative output that follows (we'll help with some blank lines):

```
$ python jwt/claims_time.py
Invalid audience
<class 'jwt.exceptions.InvalidAudienceError'>

{'payload': 'data', 'iss': 'hein', 'aud': 'learn-python'}

Invalid audience
<class 'jwt.exceptions.InvalidAudienceError'>

Invalid issuer
<class 'jwt.exceptions.InvalidIssuerError'>

Invalid audience
<class 'jwt.exceptions.InvalidAudienceError'>

{'payload': 'data', 'iss': 'hein', 'aud': 'learn-python'}
```

Now, let's see one final example for a more complex use case.

Using asymmetric (public key) algorithms

Sometimes, using a shared secret is not the best option. In such cases, it is possible to use an asymmetric key pair instead of HMAC to generate the JWT signature. In this example, we are going to create a token (and decode it) using an **RSA** key pair.

Public key cryptography, or asymmetrical cryptography, is any cryptographic system that uses pairs of keys: public keys which may be disseminated widely, and private keys which are known only to the owner. If you are interested in learning more about this topic, please see the end of this chapter for recommendations. A signature can be generated using the private key, and the public key can be used to verify the signature. Thus, two parties can exchange JWTs and the signatures can be verified without any need for a shared secret.

Now, let's create an RSA key pair. We're going to use the `ssh-keygen` utility from OpenSSH (`https://www.ssh.com/ssh/keygen/`) to do this. In the folder where our scripts for this chapter are, we created a `jwt/rsa` subfolder. Within it, run the following:

```
$ ssh-keygen -t rsa -m PEM
```

Give the name `key` when asked for a filename (it will be saved in the current folder), and simply hit the *Enter* key when asked for a passphrase.

Having generated our keys, we can now change back to the `ch09` folder, and run this code:

```python
# jwt/token_rsa.py
import jwt

data = {'payload': 'data'}

def encode(data, priv_filename, algorithm='RS256'):
    with open(priv_filename, 'rb') as key:
        private_key = key.read()
    return jwt.encode(data, private_key, algorithm=algorithm)

def decode(data, pub_filename, algorithm='RS256'):
    with open(pub_filename, 'rb') as key:
        public_key = key.read()
    return jwt.decode(data, public_key, algorithms=[algorithm])
```

```
token = encode(data, 'jwt/rsa/key')
data_out = decode(token, 'jwt/rsa/key.pub')
print(data_out)
```

In this example, we defined a couple of custom functions to encode and decode tokens using private/public keys. As you can see in the encode() function, we are using the RS256 algorithm this time. Notice that when we encode, we provide the private key, which is used to generate the JWT signature. When we decode the JWT, we instead supply the public key, which is used to verify the signature.

The logic is pretty straightforward, and we would encourage you to think about at least one use case where this technique might be more suitable than using a shared key.

Useful references

Here, you can find a list of useful references if you want to dig deeper into the fascinating world of cryptography:

- Cryptography: `https://en.wikipedia.org/wiki/Cryptography`
- JSON Web Tokens: `https://jwt.io`
- RFC standard for JSON Web Tokens: `https://datatracker.ietf.org/doc/html/rfc7519`
- Hash functions: `https://en.wikipedia.org/wiki/Cryptographic_hash_function`
- HMAC: `https://en.wikipedia.org/wiki/HMAC`
- Cryptography services (Python STD library): `https://docs.python.org/3.7/library/crypto.html`
- IANA JSON Web Token Claims Registry: `https://www.iana.org/assignments/jwt/jwt.xhtml`
- PyJWT library: `https://pyjwt.readthedocs.io/`
- Cryptography library: `https://cryptography.io/`

There is way more information on the web, and plenty of books you can also study, but we'd recommend that you start with the main concepts and then gradually dive into the specifics you want to understand more thoroughly.

Summary

In this short chapter, we explored the world of cryptography in the Python standard library. We learned how to create a hash (or digest) for a message using different cryptographic functions. We also learned how to create tokens and deal with random data when it comes to the cryptography context.

We then took a small tour outside the standard library to learn about JSON Web Tokens, which are commonly used in authentication and claims-related functionalities by modern systems and applications.

The most important thing is to understand that doing things manually can be very risky when it comes to cryptography, so it's always best to leave it to the professionals and simply use the tools we have available.

The next chapter will be about testing our code so that we can be confident that it works the way it is supposed to.

10
Testing

"Just as the wise accepts gold after testing it by heating, cutting, and rubbing it,
so are my words to be accepted after examining them, but not out of respect for me."

– Buddha

We love this quote by the Buddha. Within the software world, it translates perfectly into the healthy habit of never trusting code just because someone smart wrote it or because it's been working fine for a long time. If it has not been tested, code is not to be trusted.

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 the code. But we definitely want our code to be as predictable as possible. What we don't want is to have a surprise, in other words, our code behaving in an unpredictable way. Would you be happy to know that the software that checks the sensors of the plane that is taking you on your holiday 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 broken or unreachable, that the performance is well within the acceptable range, and so on.

This chapter is all about that—making sure that your code is prepared to face the scary outside world, that it is fast enough, and that it can deal with unexpected or exceptional conditions.

In this chapter, we're going to explore the following topics:

- General testing guidelines
- Unit testing
- A brief mention of test-driven development

Let's start by understanding what testing is.

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 who 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 that exercise the internals of the code; they inspect it down to a very fine level of detail. On the other hand, black-box tests are those that consider the software under test as if 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 plug some input into one end of the box and verify the output at the other end—that's it.

> There is also an in-between category, called **gray-box** testing, which 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. To give you an idea, here are a few:

- **Frontend tests**: They 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. They may also verify that it is possible to walk a certain path through the user interface.

- **Scenario tests**: They make use of stories (or scenarios) that help the tester work through a complex problem or test a part of the system.

- **Integration tests**: They verify the behavior of the various components of your application when they are working together sending messages through interfaces.

- **Smoke tests**: 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**): What a developer does with a product owner (for example, in a SCRUM environment) to determine whether the work that was commissioned was carried out correctly.

- **Functional tests**: They verify the features or functionalities of your software.

- **Destructive tests**: They take down parts of your system, simulating a failure, to establish how well the remaining parts of the system perform. These kinds of tests are performed extensively by companies that need to provide a highly reliable service.

- **Performance tests**: They 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 the bottlenecks in the system that could bring it to its knees in a heavy-load situation, or those that prevent scalability.

- **Usability tests**, and the closely related **user experience** (**UX**) **tests**: They aim to check whether the user interface is simple and easy to understand and use. They also aim to provide input to the designers so that the UX is improved.

- **Security and penetration tests**: They aim to verify how well the system is protected against attacks and intrusions.

- **Unit tests**: They help the developer 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**: They 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 resurfacing, an existing feature being compromised, or a new issue being introduced.

Many books and articles have been written about testing, and we 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 to collect as much knowledge about testing as you can, but you will likely be able to test your software properly only when you have accumulated enough experience.

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 to change a small feature in the code, you learn to write simpler, shorter, and better-focused tests.

We could go on with this *when you... you learn...*, but we guess you get the picture. You need to get your hands dirty and build experience. Our 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 composed of three sections:

- **Preparation**: This is where you set up the scene. You prepare all the data, the objects, and 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, an HTTP request has been made, something has happened, a method has been called, and so on.

While tests usually follow this structure, in a test suite, you will typically find some other constructs that take part in the testing game:

- **Setup**: This is something quite commonly found in several different tests. It is logic that can be customized to run for every test, class, module, or even for a whole session. In this phase, developers usually set up connections to databases, maybe populate them with data that will be needed there for the test to make sense, and so on.

- **Teardown**: This is the opposite of the setup; the teardown phase takes place when the tests have been run. Like the setup, it can be customized to run for every test, class or module, or session. Typically, in this phase, we destroy any artifacts that were created for the test suite, and clean up after ourselves. This is important because we don't want to have any lingering objects around and because it helps to make sure that each test starts from a clean slate.

- **Fixtures**: These are pieces of data used in the tests. By using a specific set of fixtures, outcomes are predictable and therefore tests can perform verifications against them.

In this chapter, we will use the *pytest* Python library. It is a powerful tool that makes testing easier than it would be if we only used standard library tools. `pytest` provides plenty of helpers so that the test logic can focus more on the actual testing than the wiring and boilerplate around it. You will see, when we get to the code, that one of the characteristics of `pytest` is that fixtures, setup, and teardown often blend into one.

Testing guidelines

Like software, tests can be good or bad, with a whole range of shades in the middle. 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 confident 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 assumptions when verifying data**. This is tricky to understand at first, but it is important. Verifying that the result of a function call is `[1, 2, 3]` is not the same as saying the output is a list that contains the numbers 1, 2, and 3. In the former, we're also assuming the ordering; in the latter, we're only assuming which items are in the list. The differences sometimes are quite subtle, but they are still very important.

- **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 a function is 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 codebase when you amend your software frequently.

- **Tests should use the minimal set of fixtures needed to do the job**. This is another crucial point. Fixtures have a tendency to grow over time. They also tend to change every now and then. If you use large amounts of fixtures and ignore redundancies in your tests, refactoring will take longer. Spotting bugs will be harder. Try to use a set of fixtures that is big enough for the test to perform correctly, but not any bigger.

- **Tests 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 **test-driven development** (**TDD**), for example, you run tests very often, so speed is essential.

- **Tests should use as few resources as possible**. 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 machine is. It could be a skinny virtual machine or a CircleCI setup; your tests should run without chewing up too many resources.

> **CircleCI** is one of the largest **CI/CD** (**Continuous Integration/ Continuous Delivery**) platforms available today. It is very easy to integrate with services like GitHub, for example. You have to add some configuration (typically in the form of a file) in your source code, and CircleCI will run tests when new code is prepared to be merged in the current codebase.

Unit testing

Now that we have an idea about what testing is and why we need it, let's introduce the developer's best friend: the **unit test**.

Before we proceed with the examples, allow us to share some words of caution: we will try to give you the fundamentals about unit testing, but we don't follow any particular school of thought or methodology to the letter. Over the years, we have tried many different testing approaches, eventually coming up with our 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

Unit tests take their name from the fact that they are used to test small units of code. To explain how to write a unit test, let's take a look at a simple snippet:

```python
# data.py
def get_clean_data(source):
    data = load_data(source)
    cleaned_data = clean_data(data)
    return cleaned_data
```

The `get_clean_data()` function 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 a return value of `load_data`. And, finally, we would need to make sure that a return value of `clean_data` is what is returned by the `get_clean_data()` function as well.

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, and a file in the filesystem.

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 developer to reproduce the whole architecture on their machine. It may require the setting up of databases, APIs, services, files and folders, and so on, 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, input and output, and underlying types. For example, if you create 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 machine.

A different approach, which we favor when it's possible to do so, is to simulate entities not by using fake objects, but using special-purpose test objects instead. For example, if our 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, we would rather spawn a test database, set up the tables and data we need, and then patch the connection settings so that our tests are running real code against the test database. This is advantageous because if the underlying libraries change in a way that introduces an issue in our code, this methodology will catch this issue. A test will break. A test with mocks, on the other hand, will blissfully continue to run successfully, because the mocked interface would have no idea about the change in the underlying library. In-memory databases are excellent options for these cases.

> One of the applications that allows 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 to check against things that can change from one database to another.

Sometimes, though, it's still not possible. For example, when the software interfaces with an API, and there is no test version of that API, we would need to simulate that API using fakes. In reality, most of the time we end up having to use a hybrid approach, where we use a test version of those technologies that allow this approach, and we use fakes for everything else. Let us now talk about fakes.

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 that you can use to mock things out.

Assertions

The verification phase is done through the use of assertions. In most cases, 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; however, when using pytest, you will typically use the generic assert statement, which makes things even simpler.

Testing a CSV generator

Let's now adopt a practical approach. We will show you how to test a piece of code, and we will touch on the rest of the important concepts around unit testing within the context of this example.

We want to write an export function that does the following: it takes a list of dictionaries, each of which represents a user. It creates a CSV file, puts a header in it, and then proceeds to add all the users who are deemed valid according to some rules. The function will take three parameters: the list of user dictionaries, the name of the CSV file to create, and an indication of whether an existing file with the same name should be overwritten.

To be considered valid, and added to the output file, a user dictionary must satisfy the following requirements: each user must have at least an email, a name, and an age. There can also be a fourth field representing the role, but it's optional. The user's email address needs to be valid, the name needs to be non-empty, and the age must be an integer between 18 and 65.

This is our task, so now we are going to show you the code, and then we're going to analyze the tests we wrote for it. But, first things first, in the following code snippets, we will be using two third-party libraries: *Marshmallow* and *Pytest*. They are both in the requirements of the book's source code, so make sure you have installed them with pip.

Marshmallow is a wonderful library that provides us with the ability to serialize (or *dump*, in Marshmallow terminology) and deserialize (or *load*, in Marshmallow terminology) objects and, most importantly, gives us the ability to define a schema that we can use to validate a user dictionary. Pytest is one of the best pieces of software we have ever seen. It is used everywhere now, and has replaced other tools such as *nose*, for example. It provides us with great tools to write beautiful short tests.

But let's get to the code. We called it `api.py` just because it exposes a function that we can use to do things. We will show it to you in chunks:

```python
# api.py
import os
import csv
from copy import deepcopy

from marshmallow import Schema, fields, pre_load
from marshmallow.validate import Length, Range

class UserSchema(Schema):
    """Represent a *valid* user. """

    email = fields.Email(required=True)
    name = fields.Str(required=True, validate=Length(min=1))
    age = fields.Int(
        required=True, validate=Range(min=18, max=65)
    )
    role = fields.String()

    @pre_load()
    def strip_name(self, data, **kwargs):
        data_copy = deepcopy(data)
        try:
            data_copy['name'] = data_copy['name'].strip()
        except (AttributeError, KeyError, TypeError):
            pass
        return data_copy

schema = UserSchema()
```

Chapter 10

This first part is where we import all the modules we need (`os`, `csv`, and `deepcopy`), and some tools from `marshmallow`, and then we define the schema for the users. As you can see, we inherit from `marshmallow.Schema`, and then we set four fields. Notice we are using two string fields (`Str`), an `Email`, and an integer (`Int`). These will already provide us with some validation from `marshmallow`. Notice there is no `required=True` in the `role` field.

We need to add a couple of custom bits of code, though. We need to add validation on `age` to make sure the value is within the range we want. Marshmallow will raise `ValidationError` if it's not. It will also take care of raising an error should we pass anything but an integer.

We also add validation on `name`, because the fact that there is a `name` key in a dictionary doesn't guarantee that the value of that name is actually non-empty. We validate that the length of the field's value is at least one. Notice we don't need to add anything for the `email` field. This is because `marshmallow` will validate it for us.

After the fields declarations, we write another method, `strip_name()`, which is decorated with the `pre_load()` Marshmallow helper. This method will be run before Marshmallow deserializes (loads) the data. As you can see, we make a copy of `data` first, as in this context it is not a good idea to work directly on a mutable object, and then make sure we strip leading and trailing spaces away from `data['name']`. That key represents the name field we just declared above. We make sure we do this within a `try`/`except` block, so deserialization can run smoothly even in case of errors. The method returns the modified copy of `data`, and Marshmallow does the rest.

We then instantiate `schema`, so that we can use it to validate data. So, let's write the `export` function:

```
# api.py
def export(filename, users, overwrite=True):
    """Export a CSV file.

    Create a CSV file and fill with valid users.  If 'overwrite'
    is False and file already exists, raise IOError.
    """
    if not overwrite and os.path.isfile(filename):
        raise IOError(f"'{filename}' already exists.")

    valid_users = get_valid_users(users)
    write_csv(filename, valid_users)
```

[325]

As you see, its internals are quite straightforward. If `overwrite` is `False` and the file already exists, we raise `IOError` with a message saying the file already exists. Otherwise, if we can proceed, we simply get the list of valid users and feed it to `write_csv()`, which is responsible for actually doing the job. Let's see how all these functions are defined:

```python
# api.py
def get_valid_users(users):
    """Yield one valid user at a time from users. """
    yield from filter(is_valid, users)


def is_valid(user):
    """Return whether or not the user is valid. """
    return not schema.validate(user)
```

Turns out we coded `get_valid_users()` as a generator, as there is no need to make a potentially big list in order to put it in a file. We can validate and save them one by one. The heart of validation is simply a delegation to `schema.validate()`, which uses the `marshmallow` validation engine. This method returns a dictionary, which is empty if the data is valid according to the schema or else it will contain error information. We don't really care about collecting the error information for this task, so we simply ignore it, and our `is_valid()` function simply returns `True` if the return value from `schema.validate()` is empty, and `False` otherwise.

One last piece is missing; here it is:

```python
# api.py
def write_csv(filename, users):
    """Write a CSV given a filename and a list of users.

    The users are assumed to be valid for the given CSV structure.
    """
    fieldnames = ['email', 'name', 'age', 'role']

    with open(filename, 'w', newline='') as csvfile:
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()

        for user in users:
            writer.writerow(user)
```

Again, the logic is straightforward. We define the header in `fieldnames`, then we open `filename` for writing, and we specify `newline=''`, which is recommended in the documentation when dealing with CSV files. When the file has been created, we get a `writer` object by using the `csv.DictWriter` class. The beauty of this tool is that it is capable of mapping the user dictionaries to the field names, so we don't need to take care of the ordering.

We write the header first, and then we loop over the users and add them one by one. Notice, this function assumes it is fed a list of valid users, and it may break if that assumption is false (with the default values, it would break if any user dictionary had extra fields).

That's the whole code you have to keep in mind. We suggest you spend a moment going through it again. There is no need to memorize it, and the fact that we have used small helper functions with meaningful names will enable you to follow the testing along more easily.

Let's now get to the interesting part: testing our `export()` function. Once again, we will show you the code in chunks:

```
# tests/test_api.py
import re
from unittest.mock import patch, mock_open, call
import pytest
from ch10.api import is_valid, export, write_csv
```

Let's start from the imports: first we bring in some tools from `unittest.mock`, then `pytest`, and, finally, we fetch the three functions that we want to actually test: `is_valid()`, `export()`, and `write_csv()`. We also import the `re` module from the standard library, as it will be needed in one of the tests.

Before we can write tests, though, we need to make a few fixtures. As you will see, a **fixture** is a function that is decorated with the `pytest.fixture` decorator. They are run before each test to which they are applied. In most cases, we expect `fixture` to return something so that we can use it in a test. We have some requirements for a user dictionary, so let's write a couple of users: one with minimal requirements, and one with full requirements. Both need to be valid. Here is the code:

```
# tests/test_api.py
@pytest.fixture
def min_user():
    """Represents a valid user with minimal data. """
    return {
        'email': 'minimal@example.com',
```

```
        'name': 'Primus Minimus',
        'age': 18,
    }


@pytest.fixture
def full_user():
    """Represents a valid user with full data. """
    return {
        'email': 'full@example.com',
        'name': 'Maximus Plenus',
        'age': 65,
        'role': 'emperor',
    }
```

In this example, the only difference between the users is the presence of the role key, but it's enough to show you the point, we hope.

Notice that instead of simply declaring dictionaries at a module level, we have actually written two functions that return a dictionary, and we have decorated them with the @pytest.fixture decorator. This is because when you declare a dictionary that is supposed to be used in your tests at the module level, you need to make sure you copy it at the beginning of every test. If you don't, you may have a test that modifies it, and this will affect all tests that follow it, compromising their integrity. By using these fixtures, pytest will give us a new dictionary every test run, so we don't need to go through that copy procedure. This helps to respect the principle of independence, which says that each test should be self-contained and independent.

If a fixture returns another type instead of dict, then that is what you will get in the test.

Fixtures are also *composable*, which means they can be used in one another, which is a very powerful feature of pytest. To show you this, let's write a fixture for a list of users, in which we put the two we already have, plus one that would fail validation because it has no age. Let's take a look at the following code:

```python
# tests/test_api.py
@pytest.fixture
def users(min_user, full_user):
    """List of users, two valid and one invalid. """
    bad_user = {
        'email': 'invalid@example.com',
        'name': 'Horribilis',
    }
    return [min_user, bad_user, full_user]
```

So, now we have two users that we can use individually, and we also have a list of three users.

The first round of tests will be testing how we validate a user. We will group all the tests for this task within a class. This helps to give related tests a namespace, a place to be. As we'll see later on, it also allows us to declare class-level fixtures, which are defined just for the tests belonging to the class. One of the perks of declaring a fixture at a class level is that you can easily override one with the same name that lives outside the scope of the class. Take a look at this code:

```python
# tests/test_api.py
class TestIsValid:
    """Test how code verifies whether a user is valid or not. """

    def test_minimal(self, min_user):
        assert is_valid(min_user)

    def test_full(self, full_user):
        assert is_valid(full_user)
```

We start very simply by making sure our fixtures actually pass validation. This helps ensure that our code will correctly validate users that we know to be valid, with minimal as well as full data. Notice that we gave each test function a parameter matching the name of a fixture. This has the effect of activating the fixture for the test. When `pytest` runs the tests, it will inspect the parameters of each test and pass the return values of the corresponding fixture functions as arguments to the test.

Next, we are going to test the age. Two things to notice here: we will not repeat the class signature, so the code that follows is indented by four spaces because these are all methods within the same class. Also, we're going to use parametrization.

Parametrization is a technique that enables us to run the same test multiple times, but feeding different data to it. It is very useful as it allows us to write the test only once with no repetition, and the result will be very intelligently handled by `pytest`, which will run all those tests as if they were actually separate, thus providing us with clear error messages when they fail. Another solution would be to write one test with a `for` loop inside that runs through all the pieces of data we want to test against. The latter solution is of much lower quality though, as the framework won't be able to give you specific information as if you were running separate tests. Moreover, should any of the `for` loop iterations fail, there would be no information about what would have happened after that, as subsequent iterations would not happen. Finally, the body of the test would get more difficult to understand, due to the `for` loop extra logic. Therefore, parametrization is a far superior choice for this use case.

It spares us from having to write a bunch of almost identical tests to exhaust all possible scenarios. Let's see how we test the age:

```python
# tests/test_api.py
    @pytest.mark.parametrize('age', range(18))
    def test_invalid_age_too_young(self, age, min_user):
        min_user['age'] = age
        assert not is_valid(min_user)
```

Right, so we start by writing a test to check that validation fails when the user is too young. According to our rule, a user is too young when they are younger than 18. We check for every age between 0 and 17, by using `range()`.

If you take a look at how the parametrization works, you see that we declare the name of an object, which we then pass to the signature of the method, and then we specify which values this object will take. For each value, the test will be run once. In the case of this first test, the object's name is `age`, and the values are all those returned by `range(18)`, which means all integer numbers from 0 to 17 are included. Notice how we feed `age` to the test method, right after `self`.

We also use the `min_user` fixture in this test. In this case, we change the `age` within the `min_user` dictionary, and then we verify that the result of `is_valid(min_user)` is `False`. We do this last bit by asserting on the fact that `not False` is `True`. In `pytest`, this is how you check for something. You simply assert that something is truthy. If that is the case, the test has succeeded. Should it instead be the opposite, the test will fail.

> Note that `pytest` will re-evaluate the fixture function for each test run that uses it, so we are free to modify the fixture data within the test without affecting any other tests.

Let's proceed and add all the tests needed to make validation fail on the age:

```python
# tests/test_api.py
    @pytest.mark.parametrize('age', range(66, 100))
    def test_invalid_age_too_old(self, age, min_user):
        min_user['age'] = age
        assert not is_valid(min_user)

    @pytest.mark.parametrize('age', ['NaN', 3.1415, None])
    def test_invalid_age_wrong_type(self, age, min_user):
        min_user['age'] = age
        assert not is_valid(min_user)
```

So, another two tests. One takes care of the other end of the spectrum, from 66 years of age to 99. And the second one instead makes sure that age is invalid when it's not an integer number, so we pass some values, such as a string, a float, and `None`, just to make sure. Notice how the structure of the test is basically always the same, but, thanks to the parametrization, we feed very different input arguments to it.

Now that we have the age-failing all sorted out, let's add a test that actually checks the age is within the valid range:

```python
# tests/test_api.py
    @pytest.mark.parametrize('age', range(18, 66))
    def test_valid_age(self, age, min_user):
        min_user['age'] = age

        assert is_valid(min_user)
```

It's as easy as that. We pass the correct range, from 18 to 65, and remove the `not` in the assertion. Notice how all tests start with the `test_` prefix, so that `pytest` can discover them, and have a different name.

We can consider the age as being taken care of. Let's move on to write tests on mandatory fields:

```python
# tests/test_api.py
    @pytest.mark.parametrize('field', ['email', 'name', 'age'])
    def test_mandatory_fields(self, field, min_user):
        del min_user[field]
        assert not is_valid(min_user)

    @pytest.mark.parametrize('field', ['email', 'name', 'age'])
    def test_mandatory_fields_empty(self, field, min_user):
        min_user[field] = ''
        assert not is_valid(min_user)

    def test_name_whitespace_only(self, min_user):
        min_user['name'] = ' \n\t'
        assert not is_valid(min_user)
```

These three tests still belong to the same class. The first one tests whether a user is invalid when one of the mandatory fields is missing. Notice that at every test run, the `min_user` fixture is restored, so we only have one missing field per test run, which is the appropriate way to check for mandatory fields. We simply remove the key from the dictionary. This time the parametrization object takes the name `field`, and, by looking at the first test, you see all the mandatory fields in the parametrization decorator: `email`, `name`, and `age`.

In the second one, things are a little different. Instead of removing keys, we simply set them (one at a time) to the empty string. Finally, in the third one, we check for the name to be made of whitespace only.

The previous tests take care of mandatory fields being there and being non-empty, and of the formatting around the `name` key of a user. Good. Let's now write the last two tests for this class. We want to check that email is valid, and in the second one, the type for email, name, and the role:

```python
# tests/test_api.py
    @pytest.mark.parametrize(
        'email, outcome',
        [
            ('missing_at.com', False),
            ('@missing_start.com', False),
            ('missing_end@', False),
            ('missing_dot@example', False),
            ('good.one@example.com', True),
            ('δοκιμή@παράδειγμα.δοκιμή', True),
            ('аджай@экзампл.рус', True),
        ]
    )
    def test_email(self, email, outcome, min_user):
        min_user['email'] = email
        assert is_valid(min_user) == outcome
```

This time, the parametrization is slightly more complex. We define two objects (`email` and `outcome`) and then we pass a list of tuples, instead of a simple list, to the decorator. Each time the test is run, one of those tuples will be unpacked to fill the values of `email` and `outcome`, respectively. This allows us to write one test for both valid and invalid email addresses, instead of two separate ones. We define an email address, and we specify the outcome we expect from validation. The first four are invalid email addresses, but the last three are actually valid. We have used a couple of examples with non-ASCII characters, just to make sure we're not forgetting to include our friends from all over the world in the validation.

Notice how the validation is done, asserting that the result of the call needs to match the outcome we have set.

Let's now write a simple test to make sure validation fails when we feed the wrong type to the fields (again, the age has been taken care of separately before):

```python
# tests/test_api.py
    @pytest.mark.parametrize(
```

```
        'field, value',
        [
            ('email', None),
            ('email', 3.1415),
            ('email', {}),
            ('name', None),
            ('name', 3.1415),
            ('name', {}),
            ('role', None),
            ('role', 3.1415),
            ('role', {}),
        ]
    )
    def test_invalid_types(self, field, value, min_user):
        min_user[field] = value
        assert not is_valid(min_user)
```

As we did before, just for fun, we pass three different values, none of which is actually a string. This test could be expanded to include more values, but, honestly, we shouldn't need to write tests such as this one. We have included it here just to show you what's possible, but normally you would need to focus on making sure the code considers valid types those that have to be considered valid, and that should be enough.

Before we move to the next test class, let take a moment to talk a bit more about something we briefly touched on when testing the age.

Boundaries and granularity

While checking for the age, we wrote three tests to cover the three ranges: 0-17 (fail), 18-65 (success), and 66-99 (fail). Why did we do this? The answer lies in the fact that we are dealing with two boundaries: 18 and 65. So our testing needs to focus on the three regions those two boundaries define: before 18, within 18 and 65, and after 65. How you do it is not crucial, as long as you make sure you test the boundaries correctly. This means if someone changes the validation in the schema from `18 <= value <= 65` to `18 <= value < 65` (notice the second `<=` is now `<`), there must be a test that fails on 65.

This concept is known as a **boundary**, and it's very important that you recognize them in your code so that you can test against them.

Another important thing is to understand which zoom level we want in order to get close to the boundaries. In other words, which unit should I use to move around them?

In the case of age, we're dealing with integers, so a unit of 1 will be the perfect choice (which is why we used 16, 17, 18, 19, 20, ...). But what if you were testing for a timestamp? Well, in that case, the correct granularity will likely be different. If the code has to act differently according to your timestamp and that timestamp represents seconds, then the granularity of your tests should zoom down to seconds. If the timestamp represents years, then years should be the unit you use. We hope you get the picture. This concept is known as **granularity** and needs to be combined with that of boundaries, so that by going around the boundaries with the correct granularity, you can make sure your tests are not leaving anything to chance.

Let's now continue with our example, and test the export function.

Testing the export function

In the same test module, we defined another class that represents a test suite for the export() function. Here it is:

```python
# tests/test_api.py
class TestExport:
    """Test behavior of 'export' function. """

    @pytest.fixture
    def csv_file(self, tmp_path):
        """Yield a filename in a temporary folder.
        Due to how pytest 'tmp_path' fixture works, the file does
        not exist yet.
        """
        yield tmp_path / "out.csv"

    @pytest.fixture
    def existing_file(self, tmp_path):
        """Create a temporary file and put some content in it. """
        existing = tmp_path / 'existing.csv'
        existing.write_text('Please leave me alone...')
        yield existing
```

Let's start by analyzing the fixtures. We have defined them at the class level this time, which means they will be alive only for as long as the tests in the class are running. We don't need these fixtures outside of this class, so it doesn't make sense to declare them at a module level like we've done with the user ones.

So, we need two files. If you recall what we wrote at the beginning of this chapter, when it comes to interaction with databases, disks, networks, and so on, we should mock everything out. However, when possible, we prefer to use a different technique. In this case, we will employ temporary folders, which will be created and deleted within the fixture. We are much happier if we can avoid mocking. To create temporary folders, we employ the `tmp_path` fixture, from `pytest`, which is a `pathlib.Path` object.

Now, the first fixture, `csv_file`, provides a reference to a temporary folder. We can consider the logic up to and including the `yield` as the setup phase. The fixture itself, in terms of data, is represented by the temporary filename. The file itself is not present yet. When a test runs, the fixture is created, and at the end of the test, the rest of the fixture code (the part after `yield`, if any) is executed. That part can be considered the teardown phase. In the case of the `csv_file` fixture, it consists of exiting the body of the function, which means the temporary folder is deleted, along with all its content. You can put much more in each phase of any fixture, and with experience, you will master the art of doing setup and teardown this way. It actually comes very naturally quite quickly.

The second fixture is very similar to the first one, but we'll use it to test that we can prevent overwriting when we call `export` with `overwrite=False`. So, we create a file in the temporary folder, and we put some content into it, just to have the means to verify it hasn't been touched.

Let's now see the tests (as we did before, we indent to remind you they are defined in the same class):

```python
# tests/test_api.py
    def test_export(self, users, csv_file):
        export(csv_file, users)
        text = csv_file.read_text()

        assert (
            'email,name,age,role\n'
            'minimal@example.com,Primus Minimus,18,\n'
            'full@example.com,Maximus Plenus,65,emperor\n'
        ) == text
```

This test employs the `users` and `csv_file` fixtures, and immediately calls `export()` with them. We expect that a file has been created, and populated with the two valid users we have (remember the list contains three users, but one is invalid).

To verify that, we open the temporary file, and collect all its text into a string. We then compare the content of the file with what we expect to be in it. Notice we only put the header, and the two valid users, in the correct order.

Now we need another test to make sure that if there is a comma in one of the values, our CSV is still generated correctly. Being a **comma-separated values** (CSV) file, we need to make sure that a comma in the data doesn't break things up:

```python
# tests/test_api.py
    def test_export_quoting(self, min_user, csv_file):
        min_user['name'] = 'A name, with a comma'
        export(csv_file, [min_user])
        text = csv_file.read_text()

        assert (
            'email,name,age,role\n'
            'minimal@example.com,"A name, with a comma",18,\n'
         ) == text
```

This time, we don't need the whole users list; we just need one, as we're testing a specific thing and we have the previous test to make sure we're generating the file correctly with all the users. Remember, always try to minimize the work you do within a test.

So, we use `min_user`, and put a nice comma in its name. We then repeat the procedure, which is very similar to that of the previous test, and finally we make sure that the name is put in the CSV file surrounded by double quotes. This is enough for any good CSV parser to understand that they don't have to break on the comma inside the double quotes.

Now, we want one more test, which needs to check that when the file already exists and we don't want to override it, our code won't do that:

```python
# tests/test_api.py
    def test_does_not_overwrite(self, users, existing_file):
        with pytest.raises(IOError) as err:
            export(existing_file, users, overwrite=False)
        err.match(
            r"'{}' already exists\.".format(
                re.escape(str(existing_file))
            )
        )
        # let's also verify the file is still intact
        assert existing_file.read() == 'Please leave me alone...'
```

This is a beautiful test, because it allows us to show you how you can tell `pytest` that you expect a function call to raise an exception. We do it in the context manager given to us by `pytest.raises`, to which we feed the exception we expect from the call we make inside the body of that context manager. If the exception is not raised, the test will fail.

We like to be thorough in our tests, so we don't want to stop there. We also assert on the message, by using the convenient `err.match` helper. Notice that we don't need to use an `assert` statement when calling `err.match`. If the argument doesn't match, the call will raise an `AssertionError`, causing the test to fail. We also need to escape the string version of `existing_file` because on Windows, paths have backslashes, which would confuse the regular expression we feed to `err.match()`.

Finally, we make sure that the file still contains its original content (which is why we created the `existing_file` fixture) by opening it, and comparing all of its content to the string it should be.

Final considerations

Before we move on to the next topic, let's wrap up with some considerations.

First, we hope you have noticed that we haven't tested all the functions we wrote. Specifically, we didn't test `get_valid_users`, `validate`, and `write_csv`. The reason is that these functions are already implicitly tested by our test suite. We have tested `is_valid()` and `export()`, which is more than enough to make sure our schema is validating users correctly, and the `export()` function is dealing with filtering out invalid users correctly, respecting existing files when needed, and writing a proper CSV. The functions we haven't tested are the internals; they provide logic that participates in doing something that we have thoroughly tested anyway. Would adding extra tests for those functions be good or bad? Think about it for a moment.

The answer is actually difficult. The more we test, the less easily we can refactor that code. As it is now, we could easily decide to rename `validate()`, and we wouldn't have to change any of the tests we wrote. If you think about it, it makes sense, because as long as `validate()` provides correct validation to the `get_valid_users()` function, we don't really need to know about it.

If, instead, we had written tests for the `validate()` function, then we would have to change them had we decided to rename it (or to change its signature, for example).

So, what is the right thing to do? Tests or no tests? It will be up to you. You have to find the right balance. Our personal take on this matter is that everything needs to be thoroughly tested, either directly or indirectly. And we try to write the smallest possible test suite that guarantees that. This way, we will have a great test suite in terms of coverage, but not any bigger than necessary. We need to maintain those tests!

We hope this example made sense to you; we think it has allowed us to touch on the important topics.

If you check out the source code for the book, in the `test_api.py` module, you will find a couple of extra test classes that will show you how different testing would have been had we decided to go all the way with the mocks. Make sure you read that code and understand it well. It is quite straightforward and will offer you a good comparison with the approach we have shown you here.

Now, how about we run those tests?

```
$ pytest tests
========================= test session starts=========================
platform darwin -- Python 3.9.4, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: /Users/fab/.../ch10
collected 132 items

tests/test_api.py ........................................... [ 34%]
.................................................... [ 83%]
....................                                 [100%]

========================= 132 passed in 0.31s=========================
```

Make sure you run `$ pytest test` from within the `ch10` folder (add the `-vv` flag for a verbose output that will show you how parametrization modifies the names of your tests). `pytest` scans your files and folders, searching for modules that start or end with `test_`, like `test_*.py`, or `*_test.py`. Within those modules, it grabs `test`-prefixed functions or `test`-prefixed methods inside `Test`-prefixed classes (you can read the full specification in the `pytest` documentation). As you can see, 132 tests were run in less than half a second, and they all succeeded. We strongly suggest you check out this code and play with it. Change something in the code and see whether any test is breaking. Understand why it is breaking. Is it something important that means the test isn't good enough? Or is it something silly that shouldn't cause the test to break? All these apparently innocuous questions will help you gain deep insight into the art of testing.

We also suggest you study the `unittest` module, and the `pytest` library too. These are tools you will use all the time, so you need to be very familiar with them.

Let's now check out test-driven development!

Test-driven development

Let's talk briefly about **test-driven development** (**TDD**). It is a methodology that was rediscovered by Kent Beck, who wrote *Test-Driven Development by Example*, Addison Wesley, *2002*, which we encourage you to read if you want to learn about the fundamentals of this subject.

> TDD is a software development methodology that is based on the continuous repetition of a very short development cycle.

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 it 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. This technique is called *fake it 'til you make it*. In a second iteration of the TDD cycle, tests are enriched with different edge cases, and the cheating code then has to be rewritten with proper logic. Adding other test cases is sometimes called **triangulation**.

The last piece of the cycle is where the developer takes care of refactoring code and tests until they are in the desired state. This last phase is called **Refactor**.

The TDD mantra therefore is **Red-Green-Refactor**.

At first, it might feel weird to write tests before the code, and we must confess it took us a while to get used to it. If you stick to it, though, and force yourself to learn this slightly counterintuitive 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 we write our code before the tests, we 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 we write tests before the code, we can concentrate on the *what* part alone while we are writing them. When we write the code afterward, we will mostly have to take care of *how* the code has to implement *what* is required by the tests. This shift in focus allows our minds to concentrate on the *what* and *how* parts separately, yielding a brainpower boost that can feel quite surprising.

There are several other benefits that come from the adoption of this technique:

- **You will refactor with much more confidence**: Tests will break if you introduce bugs. Moreover, an architectural refactor will also benefit from having tests that act as guardians.

- **The code will be more readable**: This is crucial in a 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 loosely coupled and easier to test and maintain**: Writing the tests first forces you to think more deeply about code structure.

- **Writing tests first requires you to have a better understanding of the business requirements**: If your understanding of the requirements is lacking, 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 misunderstand.

- **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 ensures that the number of bugs in it will be much lower than in the other case.

On the other hand, there are some shortcomings of this technique:

- **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. We have fought battles with stubborn bosses in our career to be able to code using TDD. Sometimes it has been painful, but always well worth it, and we 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 it will therefore reflect in the code too**: This kind of problem is quite hard to spot until you do user acceptance testing, 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 discussion will bring clarification, which will help with writing correct tests.

- **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 a huge amount of work every time you touch your code.

We are quite passionate about TDD. When we interview for a job, one of the questions we ask is whether the company adopts it. We encourage you to check it out and use it. Use it until you feel something clicking in your mind. When that happens, then you will be free to use it according to your own judgment of the situation. Regardless of which order you write code and tests in, though, the most important thing is that you always test your code!

Summary

In this chapter, we explored the world of testing.

We tried to give you a fairly comprehensive overview of testing, especially unit testing, which is the kind of testing that a developer mostly does. We hope we have succeeded in conveying 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, we would say testing is the one that is the most important.

In the next chapter, we're going to explore debugging and profiling, which are techniques that go hand in hand with testing, and are crucial to learn well.

> We are aware that we gave you a lot of pointers in this chapter, with no links or directions. This was by choice. As a coder, there won't be a single day at work when you won't have to look something up on a documentation page, in a manual, on a website, and so on. We think it's vital for a coder to be able to search effectively for the information they need, so we hope you'll forgive us for this extra training. After all, it is for your benefit.

11
Debugging and Profiling

"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 code base ever written by a human, there will still be bugs in it; that is guaranteed. We spend an awful lot of time reading other people's code and, in our opinion, a good software developer is someone who keeps an eye out for potential bugs, 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 every coder needs to keep improving. Like testing, debugging is a skill that is best learned through experience. There are guidelines you can follow, but there is no magic book that will teach you everything you need to know in order to become good at this.

We feel that on this particular subject, we have learned the most from our colleagues. It amazes us to observe someone very skilled attacking a problem. We 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 a solution.

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 whether there is something you can study in-depth 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 that 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 us, are the non-deterministic 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 truly 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 split seconds, when your boss is breathing down your neck, that you have to be able to keep calm. And we 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 logical deduction, and to everything you need at that moment. So, take a deep breath, sit properly, and focus.

In this chapter, we 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.

Specifically, we're going to look at the following:

- Debugging techniques
- Troubleshooting guidelines
- Profiling

Debugging techniques

In this part, we'll introduce you to some of the techniques we use most often. This is not an exhaustive list, but it should give you some useful ideas for where to start when debugging your own Python code.

Debugging with print

The key to understanding any bug is to understand what your code is doing at the point where the bug occurs. For this reason, we'll be looking at a few different techniques for inspecting the state of a program while it is running.

Probably the easiest technique of all is to add `print()` calls at various points in your code. This allows you to easily see which parts of your code are executed, and what the values of key variables are at different points during execution. For example, if you are developing a Django website and what happens on a page is not what you would expect, you can fill the view with prints and keep an eye on the console while you reload the page.

There are several drawbacks and limitations to using `print()` for debugging. To use this technique, you need to be able to modify the source code and run it in a terminal where you can see the output of your `print()` function calls. This is not a problem in your development environment on your own machine, but it does limit the usefulness of this technique in other environments.

When you scatter calls to `print()` in your code, you can easily end up duplicating a lot of debugging code. For example, you may want to print timestamps (like we did when we were measuring how fast list comprehensions and generators were), or somehow build up a string with the information that you want to display. Another issue is that it's extremely easy to forget calls to `print()` in your code.

For these reasons, we sometimes prefer to use a custom debugging function rather than just bare calls to `print()`. Let's see how.

Debugging with a custom function

Having a custom debugging function in a snippet that you can quickly grab and paste into the code 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:

```python
# 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, we are using a keyword-only argument to be able to print a separator, which is a line of 40 dashes.

The function is very simple. We just redirect whatever is in `msg` to a call to `print()` and, if `print_separator` is `True`, we print a line separator. Running the code will show the following:

```
$ 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 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:

```python
# 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 that we used an `import` statement *inside* our `debug()` function to import the `time()` function from the `time` module. This allows us to avoid having to add that import outside of the function, and maybe forget it there.

Take a look at how we defined `timestamp`. It's a function parameter with a list as its default value. In *Chapter 4*, *Functions, the Building Blocks of Code*, we warned against using mutable defaults for parameters, because the default value is initialized when Python parses the function and the same object persists across different calls to the function. Most of the time, this is not the behavior you want. In this case, however, we are taking advantage of this feature to store a timestamp from the previous call to the function, without having to use an external global variable. We borrowed this trick from our studies on **closures**, a technique that we encourage you to read about.

After printing whatever message we had to print and importing `time()`, we inspect the content of the only item in `timestamp`. If it is `None`, we have no previous timestamp, so we set the value to the current time (#1). On the other hand, if we have a previous timestamp, we can calculate a difference (which we neatly format to three decimal digits) and finally, we put the current time in `timestamp` (#2).

Running this code outputs the following:

```
$ python custom_timestamp.py
Entering nasty piece of code...
First step done.
 Time elapsed: 0.300s
Second step done.
 Time elapsed: 0.500s
```

Using a custom debug function solves some of the problems associated with just using `print()`. It reduces duplication of debugging code and makes it easier to remove all your debugging code when you no longer need it. However, it still requires modifying the code and running in a console where you can inspect the output. Later in this chapter, we'll see how we can overcome those difficulties by adding logging to our code.

Using the Python debugger

Another very effective way of debugging Python is to use an interactive debugger. The Python standard library module `pdb` provides such a debugger; however, we usually prefer to use the third-party `pdbpp` package. `pdbpp` is a drop-in replacement for `pdb`, with a somewhat friendlier user interface and some handy additional tools, our favorite of which is the **sticky mode**, which allows you to see a whole function while you step through its instructions.

There are a few different ways to activate the debugger (the exact same methods work for both plain `pdb` and `pdbpp`). The most common approach is to add a call invoking the debugger to your code. This is known as adding a **breakpoint** to the code. When the code is run and the interpreter reaches the breakpoint, execution is suspended and you get console access to an interactive debugger session that allows you to inspect all the names in the current scope, and step through the program one line at a time. 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 `KeyError` because a key is missing in a dictionary. The dictionary 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 afterward. Let's see how we could intercept this moment, inspect the data, fix it, and get to the bottom of it, with the debugger:

```python
# pdebugger.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 `'third'` value, which is missing from the dictionary. 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 pdebugger.py
v1
v2
Traceback (most recent call last):
  File "pdebugger.py", line 11, in <module>
    do_something_with_value(d[key])
KeyError: 'third'
```

So, we see that that `key` is missing from the dictionary, but since every time we run this code we may get a different dictionary or `keys` tuple, this information doesn't really help us. Let's inject a call to `pdb` just before the `for` loop. You have two options:

```
import pdb
pdb.set_trace()
```

This is the most common way of doing it. You import `pdb` and call its `set_trace()` method. Many developers have macros in their editor to add this line with a keyboard shortcut. As of Python 3.7, though, we can simplify things even further, to this:

```
breakpoint()
```

The new `breakpoint()` built-in function calls `sys.breakpointhook()` under the hood, which is programmed by default to call `pdb.set_trace()`. However, you can reprogram `sys.breakpointhook()` to call whatever you want, and therefore `breakpoint()` will point to that too, which is very convenient.

The code for this example is in the `pdebugger_pdb.py` module. 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 us):

```
$ python pdebugger_pdb.py
[0] > pdebugger_pdb.py(17)<module>()
-> for key in keys:
(Pdb++) l
 17
 18 -> for key in keys:  # breakpoint comes in
 19 do_something_with_value(d[key])
 20

(Pdb++) keys  # inspecting the keys tuple
('first', 'second', 'third', 'fourth')
(Pdb++) d.keys()  # inspecting keys of 'd'
dict_keys(['first', 'second', 'fourth'])
(Pdb++) d['third'] = 'placeholder'  # add missing item
(Pdb++) c  # continue
v1
v2
placeholder
v4
Validation done.
```

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 stack trace, and interacting with the objects. In our case, we first inspect the `keys` tuple. We also inspect 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.

The debugger 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 to handle breaks with (*b*)reak. For a complete list of commands, please refer to the documentation (which you can find at `https://docs.python.org/3.7/library/pdb.html`) or type (*h*)elp in the debugger console.

You can see, from the output of the preceding run, that we could finally get to the end of the validation.

`pdb` (or `pdbpp`) is an invaluable tool that we use every day. So, go and have fun, set a breakpoint somewhere and try to inspect it, follow the official documentation, and try the commands in your code to see their effect and learn them well.

> Notice that in this example we have assumed you installed `pdbpp`. If that is not the case, then you might find that some commands behave a bit differently in plain `pdb`. One example is the letter *d*, which `pdb` interprets as the down command. In order to get around that, you would have to add an *!* in front of *d*, to tell `pdb` that it is meant to be interpreted literally, and not as a command.

Inspecting logs

Another way of debugging a misbehaving application is to inspect its logs. A **log** is an ordered list of events that occurred or actions that were taken during the running of an application. If a log is written to a file on disk, it is known as a **log file**.

Using logs for debugging is in some ways similar to adding `print()` calls or using a custom debug function. The key difference is that we typically add logging to our code from the start, to aid future debugging, rather than adding it during debugging and then removing it again. Another difference is that logging can easily be configured to output to a file or a network location. These two aspects make logging ideal for debugging code that's running on a remote machine that you might not have direct access to.

The fact that logging is usually added to the code before a bug has occurred does pose the challenge of deciding what to log. We would typically expect to find entries in the logs corresponding to the start and completion (and potentially also intermediate steps) of any important process that takes place within the application. The values of important variables should be included in these log entries. Errors also need to be logged, so that if a problem occurs, we can inspect the logs to find out what went wrong.

Nearly every aspect of logging in Python can be configured in various different ways. This gives us a lot of power, as we can change where logs are output to, which log messages are output, and how log messages are formatted, simply by reconfiguring the logging and without changing any other code. The four main types of objects involved in logging in Python are:

- **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 severity level associated with it. The most commonly used levels are `DEBUG`, `INFO`, `WARNING`, `ERROR`, and `CRITICAL`. Loggers use these levels to determine which log messages to output. Anything below the logger's level will be ignored. This means that you have to take care to log at the appropriate level. If you log everything at the `DEBUG` level, you will need to configure your logger at (or below) the `DEBUG` level to see any of your messages. This can quickly result in your log files becoming extremely big. A similar problem occurs if you log everything at the `CRITICAL` level.

Python gives you several choices of where to log to. You can log to a file, a network location, a queue, a console, your operating system's logging facilities, and so on. Where you send your logs will typically depend very much on the context. For example, when you run your code in your development environment, you will typically log to your terminal. If your application runs on a single machine, you might log to a file or send your logs to the operating system's logging facilities. On the other hand, if your application uses a distributed architecture that spans over multiple machines (such as in the case of service-oriented or microservice 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 trying to correlate giant files from several different sources to figure out what went wrong can become truly challenging.

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.

Microservice architectures are an evolution of SOAs, but follow a different set of architectural patterns.

The downside of the configurability of Python's logging is that the logging machinery is somewhat complex. The good news is that you often don't need to configure very much. If you start simple, it's actually not that difficult. To prove that, we will show you a very simple example of logging a few messages to a file:

```python
# log.py
import logging

logging.basicConfig(
    filename='ch11.log',
    level=logging.DEBUG,
    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 %s is %s', position, mylist[position]
        )
    except IndexError:
        logging.exception('Faulty position: %s', position)

logging.info('Done processing 'mylist'.')
```

Let's go through it line by line. First, we import the `logging` module, then we set up a basic configuration. We specify a filename, configure the logger to output any log messages with level DEBUG or higher, and set the message format. We'll log the date and time information, the level, and the message.

With the configuration in place, we can start logging. We start by logging an info message that tells us we're about to process our list. Inside the loop, we will log the value at each position (we use the debug() function to log at the DEBUG level). We use debug() here so that we can filter out these logs in the future (by setting the minimum level to logging.INFO or more), because we might have to handle very big lists and we don't want to always log all the values.

If we get IndexError (and we do, since we are looping over range(4)), we call logging.exception(), which logs at the ERROR level, but it also outputs the exception traceback.

At the end of the code, we log another info message to say that we are done. After running this code, we will have a new ch11.log file with the following content:

```
# ch11.log
[07/19/2021 10:32:28 PM] INFO: Starting to process 'mylist'...
[07/19/2021 10:32:28 PM] DEBUG: Value at position 0 is 1
[07/19/2021 10:32:28 PM] DEBUG: Value at position 1 is 2
[07/19/2021 10:32:28 PM] DEBUG: Value at position 2 is 3
[07/19/2021 10:32:28 PM] ERROR: Faulty position: 3
Traceback (most recent call last):
  File "log.py", line 16, in <module>
    'Value at position %s is %s', position, mylist[position]
IndexError: list index out of range
[07/19/2021 10:32:28 PM] INFO: Done processing 'mylist'.
```

This is exactly what we need to be able to debug an application that is running on a remote machine, rather than our own development environment. 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 information in the *Python HOWTOs* section of the official Python documentation: *Logging HOWTO* and *Logging Cookbook*.

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

We'll end this section on debugging by briefly mentioning a couple of other techniques that you may find useful.

Reading tracebacks

Bugs often manifest as unhandled exceptions. The ability to interpret an exception traceback is therefore a crucial skill for successful debugging. Make sure that you have read and understood the section on tracebacks in *Chapter 7*, *Exceptions and Context Managers*. If you're trying to understand why an exception happened, it is often useful to inspect the state of your program (using the techniques we discussed above) at the lines mentioned in the traceback.

Assertions

Bugs are often the result of incorrect assumptions in our code. Assertions can be very useful for validating those assumptions. If our assumptions are valid, the assertions pass and all proceeds regularly. If they are not, we get an exception telling us which of our assumptions are incorrect. Sometimes, instead of inspecting with a debugger or `print()` statements, it's quicker to drop a couple of assertions in the code just to exclude possibilities. Let's see an example:

```python
# assertions.py
mylist = [1, 2, 3]  #  pretend this comes from an external source
assert 4 == len(mylist)  # this will break
for position in range(4):
    print(mylist[position])
```

In this example, we pretend that `mylist` comes from some external source that we don't control (maybe user input). The `for` loop assumes that `mylist` has four elements and we have added an assertion to validate that assumption. When we run the code, the result is this:

```
$ python assertions.py
Traceback (most recent call last):
  File "assertions.py", line 4, in <module>
    assert 4 == len(mylist)  # this will break
AssertionError
```

This tells us exactly where the problem is.

> Running a program with the `-O` flag active will cause Python to ignore all assertions. This is something to keep in mind if our code depends on assertions to work.

Assertions also allow for a longer format that includes a second expression, such as:

```
assert expression1, expression2
```

Typically, `expression2` is a string that is fed to the `AssertionError` exception raised by the statement. For example, if we changed the assertion in the last example to the following:

```
assert 4 == len(mylist), f"Mylist has {len(mylist)} elements"
```

The result would be:

```
$ python assertions.py
Traceback (most recent call last):
  File "assertions.py", line 19, in <module>
    assert 4 == len(mylist), f"Mylist has {len(mylist)} elements"
AssertionError: Mylist has 3 elements
```

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`, `tracemalloc`, and of course `pdb`. Just head to the standard library section in the documentation and you'll find all this information very easily.

Let's now explore some troubleshooting guidelines.

Troubleshooting guidelines

In this short section, we'd like to give you a few tips that come from our troubleshooting experience.

Where to inspect

Our first suggestion concerns where to place your debugging breakpoints. Regardless of whether you are using `print()`, a custom function, `pdb`, or logging, you still have to choose where to place the calls that provide you with the information. Some places are definitely better than others, and there are ways to handle the debugging progression that are better than others.

We normally avoid placing a breakpoint inside an `if` clause. If the branch containing the breakpoint is not executed, we lose the chance to get the information we wanted. Sometimes it can be difficult to reproduce a bug, or it may take a while for your code to reach 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 step through all 100 lines, checking your data at every step. In the worst-case scenario, 99 lines (and many cups of coffee) later, you spot the bug. So, consider using a different approach.

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 backward or forward, by half the jump you did last time.

In our worst-case scenario, your debugging would go from 1, 2, 3, ..., 99, in a linear fashion, to a series of jumps such as 50, 75, 87, 93, 96, ..., 99 which is much 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

In *Chapter 10*, *Testing*, we briefly introduced you to **test-driven development** (**TDD**). One TDD practice that you really should adopt, even if you don't subscribe to TDD as a whole, is writing tests that reproduce a bug before you start changing your code to fix the bug. There are a number of reasons for this. If you have a bug and all tests are passing, it means something is wrong or missing in your test code base. Adding these tests will help you ensure that you really do fix the bug: the tests should only pass if the bug is gone. Finally, having these tests will protect you from inadvertently reintroducing the same bug again when you make further changes to your code.

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, or an external component being unresponsive. In these cases, it's important to have an idea of what the big picture was when the problem occurred 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 times, and 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.

Now let's move on to see how we can profile Python code.

Profiling Python

Profiling means having the application run while keeping track of several different parameters, such as the number of times a function is called and the amount of time spent inside it.

Profiling is closely related to debugging. Although the tools and processes used are quite different, both activities involve probing and analyzing your code to understand where the root of a problem lies and then making changes to fix it. The difference is that instead of incorrect output or crashing, the problem we are trying to solve is poor performance.

Sometimes profiling will point to where the performance bottleneck is, at which point you will need to use the debugging techniques we discussed earlier in this chapter to understand why a particular piece of code does not perform as well as it should. For example, faulty logic in a database query might result in loading thousands of rows from a table instead of just hundreds. Profiling might show you that a particular function is called many more times than expected, at which point you would need to use your debugging skills to work out why that is and fix the problem.

There are a few different ways to profile a Python application. If you take a look at the profiling section in the standard library official documentation, you will see that there are two different implementations of the same profiling interface, `profile` and `cProfile`:

- `cProfile` is written in C and adds comparatively little overhead, which makes it suitable for profiling long-running programs
- `profile` is implemented in pure Python and, as a result, adds significant overhead to profiled programs

This interface does **deterministic 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 program's call stack at regular intervals, 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 as much overhead as one would think, so we'll show you a simple example using `cProfile` from the command line.

> There are situations where even the relatively low overhead of `cProfile` is not acceptable, for example, if you need to profile code on a live production web server because you cannot reproduce the performance problem in your development environment. For such cases, you really do need a statistical profiler. If you are interested in statistical profiling for Python, we suggest you look at `py-spy` (`https://github.com/benfred/py-spy`).

We're going to calculate Pythagorean triples (we know, you've missed them...) using the following code:

```python
# 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 whether they belong to a right triangle. We use `calc_hypotenuse()` to get hypotenuse for a and b, and then, with `is_int()`, we check whether it is an integer, which means (a, b, hypotenuse) is a Pythagorean triple. When we profile this script, we get information in a tabular form.

The columns are `ncalls` (the number of calls to the function), `tottime` (the total time spent in each function), `percall` (the average time spent in each function per call), `cumtime` (the cumulative time spent in a function plus all functions it calls), `percall` (the average cumulative time spent per call), and `filename:lineno(function)`. We'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. Here is the result we got:

```
$ python -m cProfile profiling/triples.py
1502538 function calls in 0.489 seconds
Ordered by: standard name

ncalls   tottime   cumtime   filename:lineno(function)
500500    0.282     0.282    triples.py:13(calc_hypotenuse)
500500    0.065     0.086    triples.py:17(is_int)
     1    0.000     0.489    triples.py:3(<module>)
     1    0.121     0.489    triples.py:3(calc_triples)
     1    0.000     0.489    {built-in method builtins.exec}
  1034    0.000     0.000    {method 'append' of 'list' objects}
     1    0.000     0.000    {method 'disable' of '_lsprof.Profile...
500500    0.021     0.021    {method 'is_integer' of 'float' objects}
```

Even with this limited amount of data, we can still infer some useful information about this code. First, we can see that the time complexity of the algorithm we have chosen grows with the square of the input size. The number of calls to `calc_hypotenuse()` is exactly *mx (mx + 1)/2*. We run the script with `mx = 1000` and we got exactly 500,500 calls. Three main things happen inside that loop: we call `calc_hypotenuse()`, we call `is_int()`, and, if the condition is met, we append it to the `triples` list.

Taking a look at the cumulative times in the profiling report, we notice that the algorithm has spent 0.282 seconds inside `calc_hypotenuse()`, which is a lot more than the 0.086 seconds spent inside `is_int()`. Given that they were called the same number of times, let's see whether we can boost `calc_hypotenuse()` a little.

As it turns out, we can. As we mentioned earlier in this 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:

```
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 0.282 is now down to 0.084. Not bad! This means now we're spending only about 29% as much time as before inside `calc_hypotenuse()`.

Let's see whether we can improve `is_int()` as well, by changing it like this:

```python
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. When we run the profiling against it, we see that the time taken inside the `is_int()` function (the `cumtime`) has gone down to 0.068 seconds. Interestingly, the total time spent in `is_int()` (excluding the time spent in the `n.is_integer()` method), has increased slightly, but by less time than we used to spend in `n.is_integer()`. You will find the three versions in the source code for the book.

This example was trivial, of course, but enough to show you how you could profile an application. Having the number of calls that are made to a function helps us better understand 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: the results of profiling will quite likely differ depending on what system you're running on. 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 it.

When to profile

Profiling is super cool, but we need to know when it is appropriate to do it, and what to do with the results we get.

Donald Knuth once said, "*premature optimization is the root of all evil*," and, although we wouldn't have put it quite so strongly, we do agree with him. After all, who are we to disagree with the man who gave us *The Art of Computer Programming*, *TeX*, and some of the coolest algorithms we ever studied at university?

So, first and foremost: *correctness*. You want your code to deliver the correct results, 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.

Second, take care of coding *best practices*. Remember the following: 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.

Third, *refactor like a beast!* The Boy Scouts rule says:

> *"Always leave the campground cleaner than you found it."*

Apply this rule to your code.

And, finally, when all of this has been taken care of, then and only then take care of optimizing and 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 trouble boosting up some part of your code using sheer Python, nothing prevents you from rolling up your sleeves, buying 200 liters of coffee, and rewriting the slow piece of code in C—guaranteed to be fun!

Measuring execution time

Before we finish this chapter, let's briefly touch on the topic of measuring the execution time of code. Sometimes it is helpful to measure the performance of small pieces of code to compare their performance. For example, if you have different ways of implementing some operation and you really need the fastest version, you may want to compare their performance without profiling your entire application.

We've already seen some examples of measuring and comparing execution times earlier in this book, for example, in *Chapter 5*, *Comprehensions and Generators*, when we compared the performance of `for` loops, list comprehensions, and the `map()` function. At this point, we would like to introduce you to a better approach, using the `timeit` module. This module uses techniques such as timing many repeated executions of the code to improve measurement accuracy.

The `timeit` module can be a bit tricky to use. We recommend that you read about it in the official Python documentation and experiment with the examples there until you understand how to use it. Here we will just give a brief demonstration of using the command-line interface to time our two different versions of `calc_hypotenuse()` from the previous example:

```
$ python -m timeit -s 'a=2; b=3' '(a**2 + b**2) ** .5'
500000 loops, best of 5: 633 nsec per loop
```

Here we are running the `timeit` module, initializing variables `a = 2` and `b = 3`, before timing the execution of `(a**2 + b**2) ** .5`. In the output, we can see that `timeit` ran 5 repetitions timing 500,000 loop iterations executing our calculation. Out of those 5 repetitions, the best average execution time over 500,000 iterations was 633 nanoseconds. Let's see how the alternative calculation, `(a*a + b*b) ** .5`, performs:

```
$ python -m timeit -s 'a=2; b=3' '(a*a + b*b) ** .5'
2000000 loops, best of 5: 126 nsec per loop
```

This time, we get 2,000,000 loop iterations with an average of 126 nanoseconds per loop. This confirms again that the second version is significantly faster. The reason we get more loop iterations in this case is that `timeit` automatically chooses the number of iterations to ensure the total running time is at least 0.2 seconds. This helps to improve accuracy by reducing the relative impact of measurement overhead.

> For further information about measuring Python performance, make sure you check out pyperf (`https://github.com/psf/pyperf`) and pyperformance (`https://github.com/python/pyperformance`).

Summary

In this short chapter, we looked at different techniques and suggestions for debugging, troubleshooting, and profiling 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 explored techniques to inspect our code using custom functions, logging, debuggers, traceback information, profiling, and assertions. We saw simple examples of most of them and we also talked about some guidelines that will help when it comes to facing the fire.

Just remember always to *stay calm and focused*, and debugging will be much easier. 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. So, when facing a difficult bug, if you have the opportunity, make sure you go for a short walk, or take a power nap—relax. Often, the solution presents itself after a good break.

In the next chapter, we are going to explore GUIs and scripts, taking an interesting detour from the more common web application scenario.

12
GUIs and Scripting

"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 are going to write a simple scraper that finds and saves images from a web page. We'll focus on three parts:

- A simple HTTP web server in Python
- A script that scrapes a given URL
- A GUI application that scrapes a given URL

> 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 `ch12` in the root of your project for this book. Within that folder, we'll create two Python modules (`scrape.py` and `guiscrape.py`) and a folder (`simple_server`). Within `simple_server`, we'll write our HTML page: `index.html`. Images will be stored in `simple_server/img`.

The structure in `ch12` 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 macOS, you can 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 purpose of this file is just to give you an example, but for such a straightforward scenario, you can simply type its content into a console and you'll be fine.

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 our 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 these owls?</p>
  </body>
</html>
```

It's a simple page, so let's just note that we have five images, three of which are PNGs and two of which 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).

Python gives you a rudimentary HTTP server that can serve static pages. You can start it with the following command (make sure you run it from the `simple_server` folder):

```
$ python -m http.server 8000
Serving HTTP on :: port 8000 (http://[::]:8000/) ...
::1 - - [04/Sep/2021 10:40:07] "GET / HTTP/1.1" 200 -
...
```

The last line is the log you get when you access `http://localhost:8000`, where the owl page will be served. Alternatively, you can put that command in a file called `serve.sh`, and just run it (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 the one depicted in *Figure 12.1*:
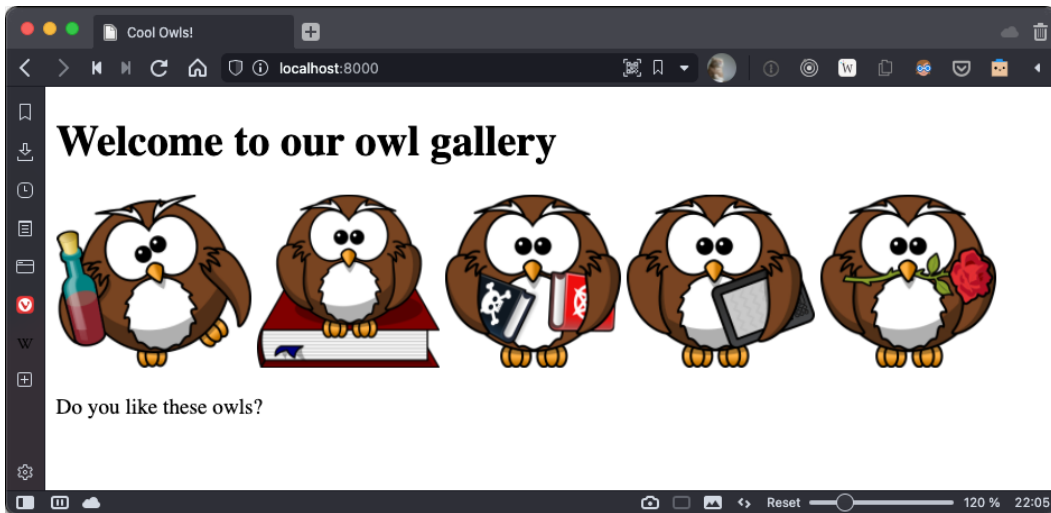
Figure 12.1: The owl page

If you wish to compare the source HTML with that in the page, you can right-click on the page and your browser should offer you an option like **View Page Source** (or something similar).

Feel free to use any other set of images, as long as you use at least one PNG and one JPG, and in the `src` tag, you use relative paths, not absolute ones. We got these lovely owls from `https://openclipart.org/`.

First approach: scripting

Now, let's start writing the script. We'll go through the source in three steps: imports, argument parsing, and business logic.

The imports

Here's how the script starts:

```python
# scrape.py
import argparse
import base64
import json
from pathlib import Path
from bs4 import BeautifulSoup
import requests
```

Going through the imports from the top, you can see that we'll need to parse the arguments, which we'll feed to the script itself (using `argparse`). We will need the `base64` library to save the images within a JSON file (so we will also need `json`), and we'll need to open files for writing (using `pathlib`). Finally, we'll need `BeautifulSoup` for scraping the web page easily, and `requests` to fetch its content. We assume you're familiar with `requests` as we used it in *Chapter 8*, *Files and Data Persistence*.

> We will explore the HTTP protocol and the requests mechanism in *Chapter 14*, *Introduction to API Development*, 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 displays it to you).

Of all these imports, only the last two don't belong to the Python standard library, so make sure you have them installed. You can check if that is the case by running the following command:

```
$ pip freeze | egrep -i "soup4|requests"
beautifulsoup4==4.9.3
requests==2.26.0
```

> The above command won't work on Windows. If that is your
> operating system, you can use the `findstr` command, instead
> of `egrep`. Alternatively, simply type `$ pip freeze`, and skim
> through to get the desired versions.

The above dependencies are listed in the `requirements.txt` file for this chapter, of
course. At this point, the only thing that might be a bit confusing is the `base64/json`
couple we are using, so allow us 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 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 single JSON file. Within the JSON, we'll put a
dictionary with keys as the image 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.

The `base64` library is quite useful. For example, every time you send an email with
an image attached to it, the image gets Base64-encoded before the email is sent. On
the recipient end, images are automatically decoded into their original binary format
so that the email client can display them.

> We used Base64 in *Chapter 9, Cryptography and Tokens*. If you
> skipped it, this would be a good time to check it out.

Parsing arguments

Now that the technicalities are out of the way, let's see the next section of our script,
argument parsing, which should be at the end of the `scrape.py` module:

```python
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 '__main__' string is the name of the scope in which top-level code executes. A module's __name__ attribute is set to '__main__' when read from standard input, a script, or from an interactive prompt.

Therefore, if you put the execution logic under that if, it will be run only when you run the script directly, as its __name__ will be '__main__' in that case. On the other hand, should you import objects from this module, then its name will be set to something else, so the logic under the if won't run.

The first thing we do is define our parser. There are several useful libraries out there nowadays to write parsers for arguments, like Docopt (`https://github.com/docopt/docopt`) and Click (`https://click.palletsprojects.com/`), but in this case, we prefer to use the standard library module, `argparse`, which is simple enough and quite powerful.

We want to feed our script three different pieces of data: the types of images we want to save, the format in which we want to save them, and the URL of the page to be scraped.

The types can be PNGs, JPGs, or both (the default option), 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()`. 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 some 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' definitions every time we change something. This translates to time saved.

Here are a few different ways to call the `scrape.py` script, which demonstrate that `type` and `format` are optional, and how you can use the short and long syntaxes to employ 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 uses 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):

```python
def scrape(url, format_, type_):
    try:
        page = requests.get(url)
    except requests.RequestException as err:
        print(str(err))
    else:
        soup = BeautifulSoup(page.content, 'html.parser')
        images = fetch_images(soup, url)
        images = filter_images(images, type_)
        save(images, format_)
```

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 `RequestException` (`err`) and print it. `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? Check out the following:

```python
def fetch_images(soup, base_url):
    images = []
    for img in soup.findAll('img'):
        src = img.get('src')
        img_url = f'{base_url}/{src}'
        name = img_url.split('/')[-1]
        images.append(dict(name=name, url=img_url))
    return images
```

The Fix spacing `fetch_images()` function takes a `BeautifulSoup` object and a base URL. All it does is loop through all of the images found on the page and fill 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. We split the img_url (for example, `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. You can read about more efficient ways of doing this in *Chapter 11, Debugging and Profiling*.

For example, if we add `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'}, ...]
```

We 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, something interesting to consider is: what would happen if the images on the page were specified with an absolute path instead of a relative one? The answer is that the script would fail to download them because this logic expects relative paths.

We hope you find the body of the `filter_images()` function below interesting. We wanted to show you how to check through multiple extensions using a mapping technique:

```python
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):
    extension = Path(filename.lower()).suffix
    return extension in extension_list
```

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()`, we have made the list comprehension simpler and more readable.

All that `matches_extension()` does is get the extension from the image name and check whether it is within the list of allowed ones.

In case you're wondering why we have collected all the images in the `images` list and then filtered out some of them, instead of checking whether we wanted to save them before adding them to the list, there are three reasons for that. The first reason is that we will need `fetch_images()` in the GUI application as it is now. The second reason is that combining fetching and filtering would produce a longer and more complicated function, and we are trying to keep the complexity level down. The third reason is that this could be a nice exercise for you to do.

Let's keep going through the code and inspect the `save()` function:

```python
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))
```

You can see that, when `images` isn't empty, this acts as a dispatcher. We either call `save_images()` or `save_json()`, depending on what information is stored in the `format_` variable.

We are almost done. Let's jump to `save_images()`. We loop over the `images` list and for each dictionary we find in it, we perform a GET request on the image URL and save its content in a file, which we name as the image itself.

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. Here's a small preview of that:

```
# images.json (truncated)
{
  "owl-alcohol.png": "iVBORw0KGgoAAAANSUhEUgAAASwAAAEICA...
  "owl-book.png": "iVBORw0KGgoAAAANSUhEUgAAASwAAAEbCAYAA...
  "owl-books.png": "iVBORw0KGgoAAAANSUhEUgAAASwAAAElCAYA...
  "owl-ebook.jpg": "/9j/4AAQSkZJRgABAQEAMQAxAAD/2wBDAAEB...
  "owl-rose.jpeg": "/9j/4AAQSkZJRgABAQEANAA0AAD/2wBDAAEB...
}
```

And that's it! Now, before proceeding to the next section, make sure you play with this script and understand how it works. Try to 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 for writing GUI applications in Python. Some of the most famous ones are **Tkinter**, **wxPython**, **Kivy**, and **PyQt**. They all offer a wide range of tools and widgets that you can use to compose a GUI application.

The one we are going to use in 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, macOS 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, we're good to go. However, if it doesn't work, please search for `tkinter` in the Python official documentation (`https://docs.python.org/3.9/library/tkinter.html`). 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 filter by image type, 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:
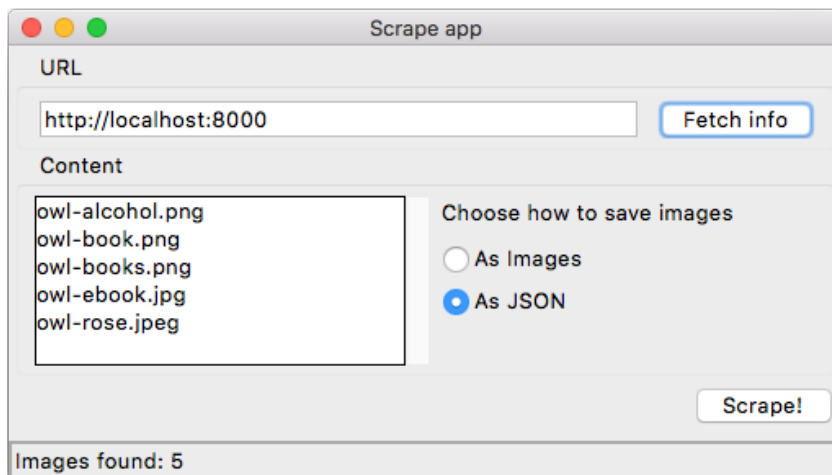
Figure 12.2: Main window of the Scrape app

Gorgeous, isn't it? As you can see, it's a very simple interface (*Figure 12.2* shows how it should look on a Mac). There is a frame (that is, a container) for the **URL** field and the **Fetch info** button, another frame for the **Listbox** (**Content**) to hold the image names and the radio buttons to control the way we save them, and there is a **Scrape!** button at the bottom right. There is also a status bar at the bottom, 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 up 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:
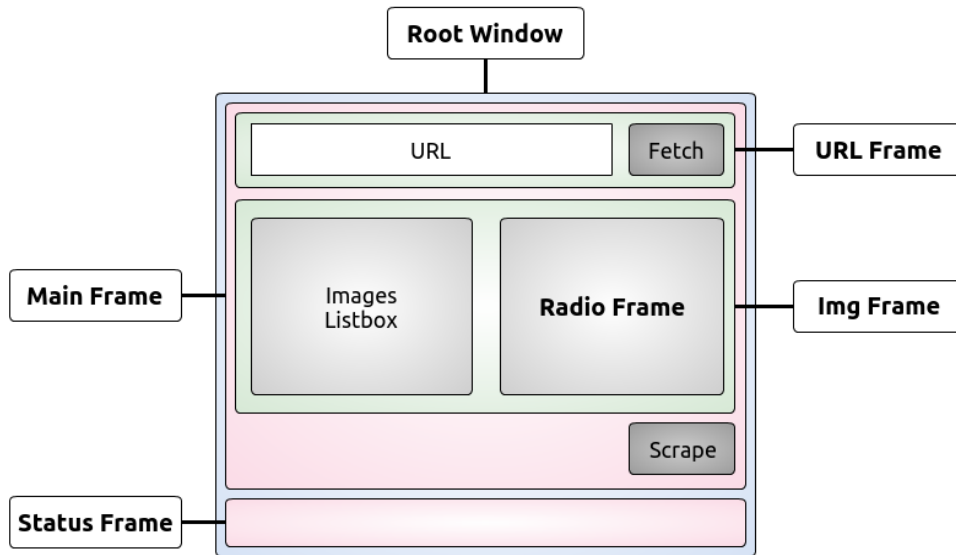
Figure 12.3: Scrape app diagram

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 text). The **Main Frame** is subsequently divided into three rows. 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, we have the 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**, which simply divides up the space into rows and columns, as in a matrix.

Now, all the code we are going to write comes from the `guiscrape.py` module, so we 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

Imports are like in the script version, except we've lost `argparse`, which is no longer needed, and we have added two lines:

```
# guiscrape.py
from tkinter import *
from tkinter import ttk, filedialog, messagebox
...
```

The first line is quite common practice when using `tkinter`, although in general it is bad practice to import using the * syntax. Doing so may result in name collisions, which result in shadowing issues and, if the imported module is very big, importing everything can be expensive.

On the second line, we import `ttk`, `filedialog`, and `messagebox` explicitly, following the conventional approach used with this library. `ttk` is Tkinter's 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 great.

The rest of the imports (omitted) are what we need in order to carry out the task you know well by now. Note that there is nothing extra we need to install with `pip` in this second part; if you have installed the requirements for the chapter, you already have everything you need.

The layout logic

We are going to present the layout logic chunk by chunk so that we 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. We will first show you, as in the script before, 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 we 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. It's possibly not the most eye-pleasing solution, but it gets the job done.

```
_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 the layout breathe a little. Little or no padding at all produces a *sardine effect*, where it feels like widgets are packed too tightly.

The second line is 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 place the **URL Frame**. This time, the parent object is `_mainframe`, as you will recall from the draft in *Figure 12.3*. This is not just a simple `Frame`; 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 *Figure 12.2* if it helps). We position this frame at (0, 0), and say that it should expand to the left (`W`) and to the right (`E`). We don't need it to expand in either of the other two directions. Finally, we use `rowconfigure()` and `columnconfigure()` to make sure it behaves correctly when resizing. 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 the Tkinter environment is called `Entry`. We instantiate it as usual, setting `_url_frame` as its parent and giving it a width in pixels. 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. This means we will not modify the text in the `_url_entry` instance directly, but rather 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 using `padx`, which adds padding on the *x*-axis (horizontal). On the other hand, `pady` takes care of the vertical direction.

By now, it should be clear that every time we 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 to 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 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 `_url_entry`, we need to drive the contents of `Listbox` by tying it to an `_images` variable. We set up `Listbox` so that `_img_frame` is its parent, and `_images` is the variable it is tied to. We also pass some dimensions.

The interesting bit comes from the `_scrollbar` instance. We pass `orient=VERTICAL` to set its orientation.

In order to tie its position to the vertical scroll of `Listbox`, when we instantiate it, we set its command to `_img_listbox.yview`. This is the first half of the contract we are creating between `Listbox` and `Scrollbar`. The other half is provided by the `_img_listbox.configure()` method, which sets `yscrollcommand=_scrollbar.set`.

By setting up this reciprocal bond, when we scroll on `Listbox`, `Scrollbar` will move accordingly and, vice versa, when we operate `Scrollbar`, `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 `Listbox` is occupying (0, 0) on `_img_frame`, `Scrollbar` takes up (0, 1), and therefore `_radio_frame` will go to (0, 2). Let's populate it:

```
_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, `Radiobutton` is also driven by a bond to an external variable, which we called `_save_method`. Each `Radiobutton` instance sets up 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.

```python
_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 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 two pixels. It needs to stick to the left, right, and bottom parts of the `_root` window, 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 with this technique by now.

Finally, on the last line, we run the application by calling the `mainloop()` method on the `Tk` instance:

```python
_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 on the first attempt, but it's easy to find helpful 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

Let's start with the code to fetch the page and images:

```python
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 err:
        sb(str(err))
    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 = f'{base_url}/{src}'
        name = img_url.split('/')[-1]
        images.append(dict(name=name, url=img_url))
    return images
```

First of all, let's discuss 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, one simple technique is to have a single dictionary that holds all the objects we need to pass back and forth.

In this simple example, we'll just populate the `config` dictionary with the images we fetch from the page, but we wanted to show you the technique so that you have at least one example.

This technique comes from our experience with JavaScript. When you code a web page, you often import several different libraries. If each of these cluttered the global namespace with all sorts of variables, there might be issues in making everything work, because of name clashes and variable overriding.

In our case, we find that using one `config` variable is a good solution to this issue.

The `fetch_url()` function is quite similar to what we did in the script. First, 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 instance of `Entry`. The text field you see on the GUI is the `Entry` object, 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 that 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 attempt 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 the action we take if things go wrong. Within the GUI application, we call `sb(str(err))`. We will see the code for the `sb()` helper function shortly. Basically, it sets the text in the status bar for us. Once you know that `sb` stands for "status bar," it makes sense, right? However, we would argue this is not a good name. We had to explain its behavior to you, which means it isn't self-explanatory. We left this as an example of how easy it can be to write poor-quality code that only makes total sense once our head is wrapped around it, hence making it difficult to spot.

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 we won't repeat it 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` as `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 would still update the status bar. 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:

```python
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 = Path(dirname).joinpath(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 either `save_images()` or `save_json()`, according to which value is held by `_save_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; there are some additional steps that must be taken before we dispatch to either `save_images()` or `save_json()`.

If we want to save the images to image format, 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 `dirname` variable. This opens up a 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 a Mac:
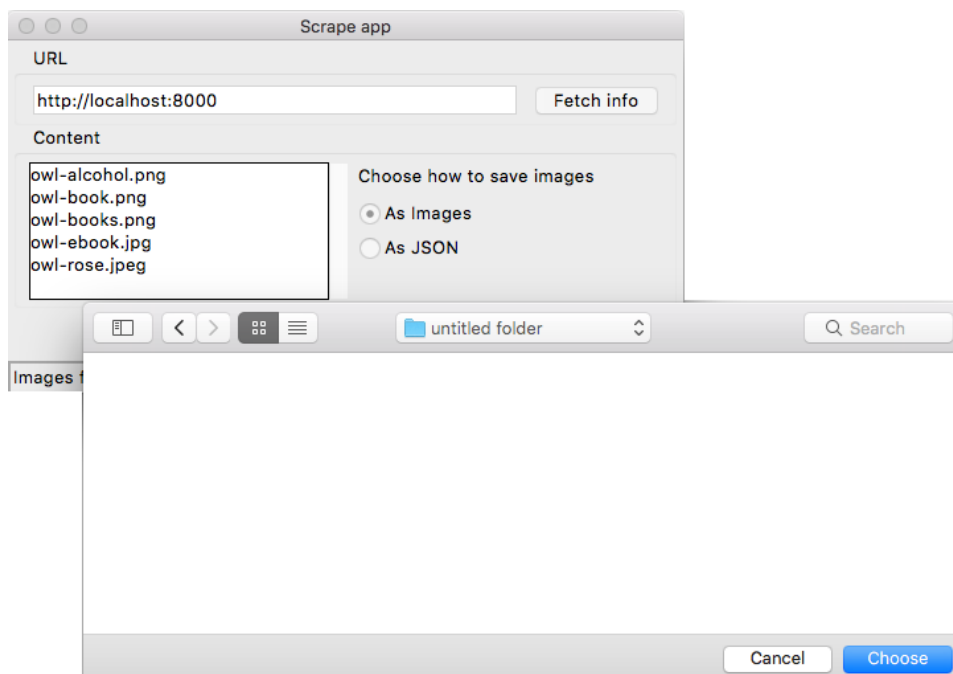
Figure 12.4: Save dialog for image format

If we cancel the operation, `dirname` is set to `None`.

Before we finish 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 joining the directory (which means the complete path) to the image name, by means of the `Path.joinpath()` method.

At the end of `save_images()`, if we saved at least one image, we alert the user that we're done.

Let's now go back to the other logic branch in `save`. This branch is executed when the user selects the **As JSON** radio button before clicking the **Scrape!** button. In this case, we want to save the data to a JSON file and 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 as a suggestion to the user, who can choose 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 two-tuples *(description, extension)*, that runs the logic of the dialog.

Here's how this dialog should look on macOS:
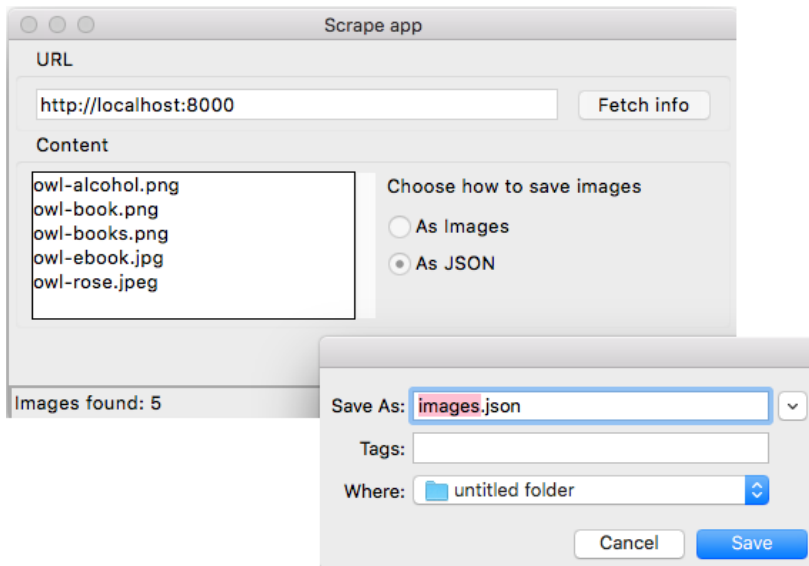
Figure 12.5: Save dialog for JSON format

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 script version of the app. We create a JSON object from a Python dictionary (`data`) that we have populated with key/value pairs made by the image's 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 filename and at least one image to save. This ensures that if the user presses the **Cancel** button, nothing happens.

Alerting the user

Finally, let's see the alerting logic. It's extremely simple:

```python
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 the `_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 a Mac:
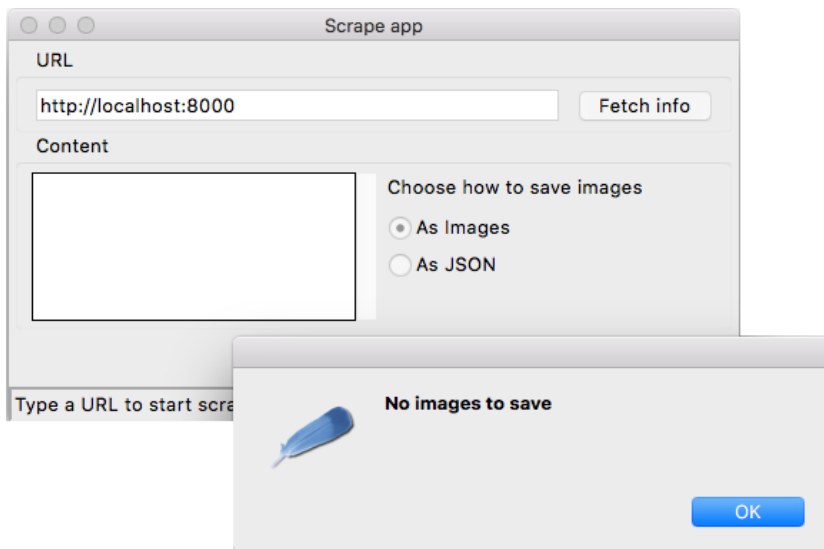
Figure 12.6: Messagebox alert example

The `messagebox` object can also be used to warn the user (`messagebox.showwarning`) or to signal an error (`messagebox.showerror`). It can also be used to provide dialogs that ask the user whether they are sure they want to proceed or if they really want to delete a certain file, and so on.

If we inspect `messagebox` by simply printing out what `dir(messagebox)` returns, we find methods such as `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`, and `YESNOCANCEL`. You can use these constants by comparing them to the user's choice so that you know the desired action to execute when the dialog closes.

Now that we have explored the code for this application, we can give it a spin with the following command:

```
$ python guiscrape.py
```

How can we improve the application?

Now that you're accustomed to the fundamentals of designing a GUI application, we would like to offer some suggestions on how to make ours better.

We can start with the code quality. Do you think this code is good enough, or would you improve it? If so, how? We would test it, and make sure it's robust and caters to all the various scenarios that a user might create by clicking around on the application. We would also make sure the behavior is what we would expect when the website we're scraping is down or unreachable for any reason.

Another thing that we could improve is the names we chose. We 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 to 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 widget set disappears gradually if you shrink. This behavior isn't acceptable, therefore one quick solution could be to make the root window fixed (that is, unable to be resized).

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 setup 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), 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:

```python
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. We 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 Listbox, so that the user can take a peek at the images before deciding to save them.

Another suggestion would be to add a menu. Maybe even a simple menu with *File* and *?* to provide the usual *Help* or *About* sections. Adding menus is not that complicated; you can add text, keyboard shortcuts, images, and so on.

In terms of business logic, it would be worth experimenting with different ways to store the data that currently is stored in the config dict. One alternative would be to use a dedicated object. You will find that being familiar with different ways to do this enables you to choose the best one for the situation at hand.

Where do we go from here?

If you are interested in digging deeper into the world of GUIs, then we would like to offer the following suggestions.

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 the center of a Cartesian plane. You can programmatically command the turtle to move forward and backward, rotate, and so on; by combining 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, Kivy, and PyQt

After you have explored the vastness of the `tkinter` realm, we would suggest you explore other GUI libraries: wxPython (`https://www.wxpython.org/`), PyQt (`https://www.riverbankcomputing.com/software/pyqt/`), and Kivy (`https://kivy.org/`). You may find that one of these works better for you, or makes it easier for you to code the application you need.

We believe that programmers can reify their ideas much better when they are conscious of what tools they have available. If your toolset is too narrow, your ideas may seem impossible or extremely hard to transform into reality, 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, our suggestion is to grow your knowledge by exploring breadth first.

Investigate several things, and then go deeper with the one or the few that looked the 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 look, 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 to us feels important, is the law or **principle of least astonishment**. It states that if in your design a necessary feature has a high astonishment 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 an application window are in the top-right corner, it's quite hard to work on a Mac, where those buttons are in 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.

Threading considerations

This topic is outside the scope of this book, but we do want to mention it.

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 (or even a different process) 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.

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. We hope you enjoyed reading it and following along as much as we enjoyed writing it.

We saw many different concepts, such as working with files and performing HTTP requests, and we talked about guidelines for usability and design.

We 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, we have pointed out several different ways in which you could improve the application, and we have challenged you with a few exercises and questions. We hope you have taken the time to play with those ideas. You 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.

13
Data Science in Brief

"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 depending on context, understanding, tools, and so on. In order to do proper data science, you need to, at the very least, know mathematics and statistics. 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.

We won't be able to talk about everything here. Therefore, in order to render this chapter meaningful, we're going to work on a project together instead.

Around 2012/2013, Fabrizio was working for a top-tier social media company in London. He stayed there for two years and was privileged to work with several very brilliant people. The company was the first in the world to have access to the Twitter Ads API, and they were partners with Facebook as well. That means a lot of data.

Their 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 Fabrizio was a part of tried to help by introducing them to Python and to the tools Python gives us to deal with data. It was a very interesting journey that led him to mentor several people in the company, eventually taking him to Manila where he gave a two-week intensive training course in Python and data science to the analysts over there.

The project we're going to do in this chapter is a lightweight version of the final example Fabrizio presented to his students in Manila. We 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.

Specifically, we are going to explore the following:

- The Jupyter Notebook and JupyterLab
- `pandas` and `numpy`: the main libraries for data science in Python
- A few concepts around Pandas's `DataFrame` class
- Creating and manipulating a dataset

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 the niceties he was used to when working with tools such as Mathematica and Maple. The result of these efforts took the name **IPython**.

In a nutshell, that small script began as an enhanced version of the Python shell and, through the efforts of other coders and eventually with 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 such as 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 became so popular and widely used that, over time, all sorts of goodies were 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 led to a huge project that ended up being 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. More recently, the Jupyter project has created **JupyterLab**, a web-based **IDE** incorporating Jupyter notebooks, interactive consoles, a code editor, and more.

This chapter's project will all be coded and run in a Jupyter Notebook, so let us briefly explain what a Notebook is. A Notebook environment is a web page that exposes a simple menu and 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 one 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 with it using a very fast messaging system.

Apart from all the graphical advantages, the beauty of having such an environment lies in the ability to run 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 one 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 the step-by-step inspection of results. 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 `ipython.org` and `jupyter.org`.

We have created a very simple example Notebook with a `fibonacci()` function that gives you a list of all the Fibonacci numbers smaller than a given `N`. It looks like this:
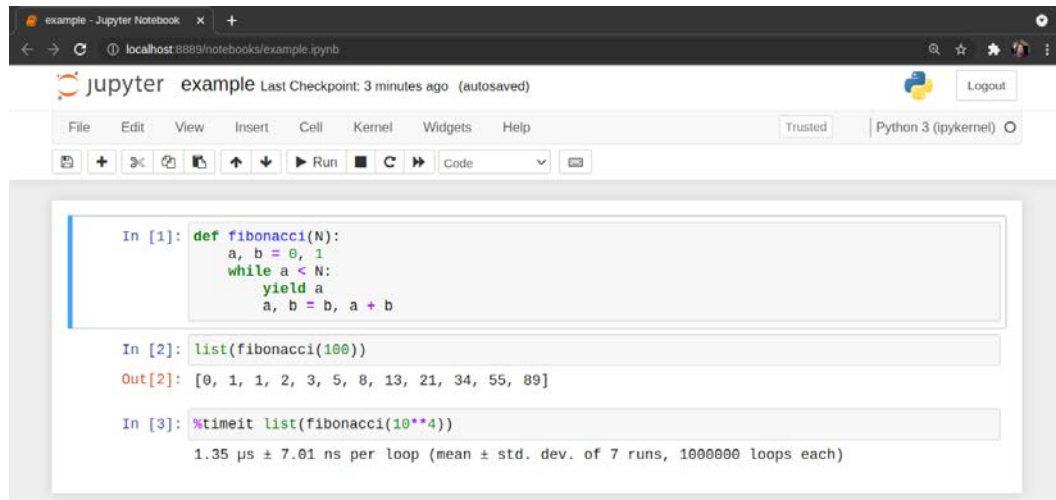
Figure 13.1: A Jupyter Notebook

Every cell has an **In []** label. If there's nothing between the brackets, it means that the 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. An ***** means that the cell is currently being executed.

You can see in the picture that in the first cell we have defined the `fibonacci()` function and executed it. This has the effect of placing the `fibonacci` name in the global scope associated with the Notebook, and therefore the `fibonacci()` function is now available to the other cells as well. In fact, in the second cell, we can run `list(fibonacci(100))` and see the results in **Out [2]**. In the third cell, we have shown you one of the several magic functions you can find in a Notebook: `%timeit` runs the code several times and provides you with a nice benchmark for it (this is implemented using the `timeit` module, which we briefly introduced in *Chapter 11, Debugging and Profiling*).

You can execute a cell as many times as you want, and change the order in which you run them. Cells are very versatile: 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.

Another useful feature is that whatever you place in the last line of a cell will be automatically printed for you. This is very handy because you're not forced to write `print(...)` explicitly.

Using Anaconda

As usual, you can install the libraries required for this chapter using the `requirements.txt` file in the source code for the chapter. Sometimes, however, installing data science libraries can be extremely painful. If you are struggling to install the libraries for this chapter in your virtual environment, an alternative choice you have is to install Anaconda. Anaconda is a free and open-source distribution of the Python and R programming languages for data science and machine learning-related applications that aims to simplify package management and deployment. You can download it from the `anaconda.org` website. Once you have installed it, you can use the Anaconda interface to create a virtual environment and install the packages listed in the `requirements.in` file, which you can also find in the source code for the chapter.

Starting a Notebook

Once you have all the required libraries installed, you can start a Notebook with the following command:

```
$ jupyter notebook
```

If you installed the requirements via Anaconda, you can also launch the Notebook from the Anaconda interface. In either case, you will have an open page in your web browser at this address (the port might be different): `http://localhost:8888/`.

You can also launch JupyterLab from Anaconda, or with the command:

```
$ jupyter lab
```

It will also open as a new page in your web browser.

Explore both interfaces. Create a new Notebook, or open the `example.ipynb` Notebook we showed you above. See which interface you prefer and get comfortable with it before proceeding with the rest of the chapter. We have included a saved JupyterLab workspace containing the Notebooks used in the rest of this chapter in the source code for the chapter (the file is called `ch13.jupyterlab-workspace`). You can use that to follow along in JupyterLab or stick to the classic Notebook interface if you prefer.

To help you follow along, we will tag each code example in this chapter with the Notebook cell number it belongs to.

> If you familiarize yourself with the keyboard shortcuts (look in the classic Notebook **Help** menu or the **Advanced Settings Editor** in JupyterLab), 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 much faster when you work in a Notebook.

Let's now move on and talk about the most interesting part of this chapter: data.

Dealing with data

Typically, when you deal with data, this is the path you go through: you fetch it; you clean and manipulate it; and then you analyze it and present results as values, spreadsheets, graphs, and so on. We 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 that it comes in a format that 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, which is a `DataFrame` from the `pandas` library.
3. We're going to manipulate the data in a `DataFrame`.
4. We're going to save a `DataFrame` to a file in different formats.
5. We're going to analyze the data and get some results out of it.

Setting up the Notebook

First things first, let's produce the data. We start from the `ch13-dataprep` Notebook. Cell #1 takes care of the imports:

```
#1
import json
import random
from datetime import date, timedelta
import faker
```

The only modules we haven't already encountered are `random` and `faker`. `random` is a standard library module for generating pseudo-random numbers. `faker` is a third-party module for generating fake data. It's very useful in tests, when you prepare your fixtures, to get all sorts of things such as names, email addresses, phone numbers, and credit card details.

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 we're using this term in a generic way. The user object may be a string, a dictionary, 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 that bad either...). Firstly, we instantiate the `Faker` that we'll use to create the data:

```
#2
fake = faker.Faker()
```

Then we need usernames. We want 1,000 unique usernames, so we loop until the `usernames` set has 1,000 elements. A `set` doesn't allow duplicate elements; therefore, uniqueness is guaranteed:

```
#3
usernames = set()
usernames_no = 1000

# populate the set with 1000 unique usernames
while len(usernames) < usernames_no:
    usernames.add(fake.user_name())
```

Next, we create a list of `users`. Each `username` will be augmented to a full-blown `user` dictionary, with other details such as `name`, `gender`, and `email`. Each `user` dictionary 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.

```
#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]
```

The get_random_name_and_gender() function is quite interesting. We use the random.
random() function to get a uniformly distributed random number between 0 and 1.
We compare the random number to 0.6, to decide whether to generate a male or a
female name. This has the effect of making 60% of our users female.

Note also the last line in the cell. Each cell automatically prints what's on the last line;
therefore, the output of #4 is a list with the first three users:

```
['{"username": "susan42", "name": "Emily Smith", "gender": ...}',
 '{"username": "sarahcarpenter", "name": "Michael Kane", ...}',
 '{"username": "kevin37", "name": "Nathaniel Miller", ...}']
```

> We hope you're following along with your own Notebook. If you
> are, 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. Also note that we've
> had to trim most of the output in this chapter to fit onto the page,
> so you will see a lot more output in your Notebook than we've
> reproduced here.

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 we have chosen is a simple example of that technique—there is a code that tells us the campaign type, then the start and end dates, then the target `age` and `gender`, and finally the currency. All values are separated by an underscore. The code to generate these campaign names can be found in cell #5:

```
#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.randrange(20, 46, 5)
    diff = random.randrange(5, 26, 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 = get_start_end_dates()
    age = get_age()
    gender = get_gender()
```

```
        currency = get_currency()
        return separator.join(
            (type_, start, end, age, gender, currency))
```

In the `get_type()` function, we use `random.choice()` to get one value randomly out of a collection. `get_start_end_dates()` is a bit more interesting. We compute two random integers: the `duration` of the campaign in days (between one day and two years) and an `offset` (a number of days between –365 and 365). We subtract `offset` (as a `timedelta`) from today's date to get the start date and add the `duration` to get the end date. Finally, we return string representations of both dates.

The `get_age()` function generates a random target age range, where both endpoints are multiples of five. We use the `random.randrange()` function, which returns a random number from a range defined by `start`, `stop`, and `step` parameters (these parameters have the same meaning as for the `range` object that we first encountered in *Chapter 3*, *Conditionals and Iteration*). We generate random numbers `age` (a multiple of 5 between 20 and 46) and `diff` (a multiple of 5 between 5 and 26). We add `diff` to `age` to get the upper limit of our age range and return a string representation of the age range.

The rest of the functions are just some 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.

In #6, we write a function that creates a complete campaign object:

```
#6
# campaign data:
# name, budget, spent, clicks, impressions
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
    }
```

We 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. To avoid having all our data look similar, we 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 number of clicks that were performed on the campaign. Normally, the number of `impressions` is greater than the number of `clicks`, because an advertisement is often viewed without being clicked on.

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 dictionary 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 for inspecting your steps.

You can vary the granularity to suit your needs. The first item in `rough_data` looks like this:

```
{'user': '{"username": "susan42", "name": "Emily Smith", ...}',
 'campaigns': [{'cmp_name': 'GRZ_20210131_20210411_30-40_F_GBP',
    'cmp_bgt': 253951,
    'cmp_spent': 17953,
    'cmp_clicks': 52573,
    'cmp_impr': 500001},
   ...
   {'cmp_name': 'BYU_20220216_20220407_20-25_F_EUR',
    'cmp_bgt': 393134,
    'cmp_spent': 158930,
    'cmp_clicks': 46631,
    'cmp_impr': 500000}]}
```

Now we can start working on the data. The first thing we need to do in order to be able to work with this `data` is to denormalize it. **Denormalization** is a process of restructuring data into a single table. This generally involves joining together data from multiple tables or flattening out nested data structures. It usually introduces some duplication of data; however, it simplifies data analysis by eliminating the need to deal with nested structures or to look related data up across multiple tables. In our case, this means transforming `data` into a list whose items are campaign dictionaries, augmented with their relative `user` dictionary. Users will be duplicated in each campaign they are associated with:

```
#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 item in `data` now looks like this:

```
{'cmp_name': 'GRZ_20210131_20210411_30-40_F_GBP',
 'cmp_bgt': 253951,
 'cmp_spent': 17953,
 'cmp_clicks': 52573,
 'cmp_impr': 500001,
 'user': '{"username": "susan42", "name": "Emily Smith", ...}'}
```

Now, we would like to help you and offer a deterministic second part of the chapter, so we're going to save the data we generated here so that we (and you, too) will be able to load it from the next Notebook, and we should then have the same results:

```
#10
with open('data.json', 'w') as stream:
    stream.write(json.dumps(data))
```

You should find the `data.json` file in the source code for the book. Now we are done with `ch13-dataprep`, so we can close it and open up the `ch13` notebook.

Creating the DataFrame

Now that we have prepared our data, we can start analyzing it. First, we have another round of imports:

```
#1
import json
import arrow
import numpy as np
import pandas as pd
from pandas import DataFrame
```

We've already seen the `json` module in *Chapter 8*, *Files and Data Persistence*. We also briefly introduced `arrow` in *Chapter 2*, *Built-In Data Types*. It is a very useful third-party library that makes working with dates and times a lot easier. `numpy` is the NumPy library, the fundamental package for scientific computing with Python. NumPy stands for Numeric Python, and it is one of the most widely used libraries in the data science environment. We'll say a bit more about it later on in this chapter. `pandas` is the very core upon which the whole project is based. **pandas** stands for **Python Data Analysis Library**. Among many other things, it provides the `DataFrame`, a matrix-like data structure with advanced processing capabilities. It's customary to `import pandas as pd` and also import `DataFrame` separately.

After the imports, we load our data into a `DataFrame` using the `pandas.read_json()` function:

```
#2
df = pd.read_json("data.json")
df.head()
```

We inspect the first five rows using the `head()` method of `DataFrame`. You should see something like this:

	cmp_name	cmp_bgt	cmp_spent	cmp_clicks	cmp_impr	user
0	GRZ_20210131_20210411_30-40_F_GBP	253951	17953	52573	500001	{"username": "susan42", "name": "Emily Smith",...
1	BYU_20210109_20221204_30-35_M_GBP	150314	125884	24575	499999	{"username": "susan42", "name": "Emily Smith",...
2	GRZ_20211124_20220921_20-35_B_EUR	791397	480963	39668	499999	{"username": "susan42", "name": "Emily Smith",...
3	GRZ_20210727_20220211_35-45_B_EUR	910204	339997	16698	500000	{"username": "susan42", "name": "Emily Smith",...
4	BYU_20220216_20220407_20-25_F_EUR	393134	158930	46631	500000	{"username": "susan42", "name": "Emily Smith",...

Figure 13.2: The first few rows of the DataFrame

Jupyter automatically renders the output of the `df.head()` call as HTML. To get a plain text representation, you can wrap `df.head()` in a `print` call.

The `DataFrame` structure is very powerful. It allows us to perform various operations on its contents. You can filter by rows or columns, aggregate data, and so on. You can operate on entire rows or columns without suffering the time penalty you would have to pay if you were working on data with pure Python. This is possible because, under the hood, `pandas` is harnessing the power of the NumPy library, which itself draws its incredible speed from the low-level implementation of its core.

Using `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.

Let's see two ways to quickly get a bird's eye view of the data:

```
#3
df.count()
```

The `count()` method returns a count of all the non-empty cells in each column. This is useful to help you understand how sparse your data is. In our case, we have no missing values, so the output is:

```
cmp_name       5140
cmp_bgt        5140
cmp_spent      5140
cmp_clicks     5140
cmp_impr       5140
user           5140
dtype: int64
```

We have 5,140 rows. Given that we have 1,000 users and the number of campaigns per user is a random number between 2 and 8, that is exactly in line with what we would expect.

> The `dtype: int64` line at the end of the output indicates that the values returned by `df.count()` are NumPy `int64` objects. Here, `dtype` stands for "data type" and `int64` means 64-bit integers. NumPy is largely implemented in C and, instead of using Python's built-in numeric types, it uses its own types, which are closely related to C language data types. This allows it to perform numerical operations much faster than pure Python.

The `describe` method is useful to quickly obtain a statistical summary of our data:

```
#4
df.describe()
```

As you can see in the output below, it gives us several measures, such as `count`, `mean`, `std` (standard deviation), `min`, and `max`, and shows how data is distributed in the various quartiles. Thanks to this method, we already have a rough idea of how our data is structured:

```
            cmp_bgt       cmp_spent     cmp_clicks      cmp_impr
count   5140.000000    5140.000000    5140.000000    5140.000000
mean  496331.855058  249542.778210   40414.236576  499999.523346
std   289001.241891  219168.636408    1704.136480       2.010877
min     1017.000000     117.000000     355.000000  499991.000000
25%   250725.500000   70162.000000   22865.250000  499998.000000
50%   495957.000000  188704.000000   37103.000000  500000.000000
75%   741076.500000  381478.750000   55836.000000  500001.000000
max   999860.000000  984005.000000   98912.000000  500007.000000
```

Let's see the three campaigns with the highest budgets:

```
#5
df.sort_values(by=['cmp_bgt'], ascending=False).head(3)
```

This gives the following output:

```
                              cmp_name cmp_bgt cmp_clicks cmp_impr
5047 GRZ_20210217_20220406_35-45_B_GBP  999860      78932   499999
922  AKX_20211111_20230908_40-50_M_GBP  999859      73078   499996
2113 BYU_20220330_20220401_35-45_B_USD  999696      42961   499998
```

A call to `tail()` shows us the campaigns with the lowest budgets:

```
#6
df.sort_values(by=['cmp_bgt'], ascending=False).tail(3)
```

Unpacking the campaign name

Now it's time to increase the complexity. 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 its own dedicated column. In order to do this, we'll use the `apply()` method of the `Series` object.

The `pandas.core.series.Series` class is a powerful wrapper around an array (think of it as a list with augmented capabilities). We can extract a `Series` object from `DataFrame` by accessing it in the same way we do with a key in a dictionary, and we can call `apply()` on that `Series` object, which will call a given function on each item in the `Series` and return a new `Series` with the results. We compose the result into a new `DataFrame`, and then join that `DataFrame` with `df`:

```
#7
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 = arrow.get(start, 'YYYYMMDD').date()
    end = arrow.get(end, 'YYYYMMDD').date()
    return type_, start, end, age, gender, currency

campaign_data = df['cmp_name'].apply(unpack_campaign_name)
campaign_cols = [
    'Type', 'Start', 'End', 'Target Age', 'Target 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` into parts. We use `arrow.get()` to get a proper `date` object out of those strings, and then we return the objects. A quick peek at the first three rows reveals:

```
   Type       Start         End Target Age Target Gender  Currency
0  GRZ   2021-01-31  2021-04-11      30-40             F       GBP
1  BYU   2021-01-09  2022-12-04      30-35             M       GBP
2  GRZ   2021-11-24  2022-09-21      20-35             B       EUR
```

That looks better! One important thing to remember: even if the dates are printed as strings, they are just the representation of the real `date` objects that are stored in `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:

```
#8
df = df.join(campaign_df)
```

And after `join()`, we take a peek, hoping to see matching data:

```
#9
df[['cmp_name'] + campaign_cols].head(3)
```

The output is as follows:

```
                           cmp_name  Type       Start         End
0  GRZ_20210131_20210411_30-40_F_GBP   GRZ  2021-01-31  2021-04-11
1  BYU_20210109_20221204_30-35_M_GBP   BYU  2021-01-09  2022-12-04
2  GRZ_20211124_20220921_20-35_B_EUR   GRZ  2021-11-24  2022-09-21
```

As you can see, `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 the `head()` method.

Unpacking the user data

We now do the 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. We create a brand new `DataFrame`, `user_df`, with this data:

```
#10
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? Next, we join `user_df` back with `df` (like we did with `campaign_df`), and take a peek at the result:

```
#11
df = df.join(user_df)


#12
df[['user'] + user_cols].head(2)
```

The output shows us that everything went well. We're not done yet though. If you call `df.columns` in a cell, you'll see that we still have ugly names for our columns. Let's change that:

```
#13
new_column_names = {
    'cmp_bgt': 'Budget',
    'cmp_spent': 'Spent',
    'cmp_clicks': 'Clicks',
    'cmp_impr': 'Impressions',
}
df.rename(columns=new_column_names, inplace=True)
```

The `rename()` method can be used to change the column (or row) labels. We've given it a dictionary mapping old column names to our preferred names. Any column that's not mentioned in the dictionary will remain unchanged. Now, with the exception of `'cmp_name'` and `'user'`, we only have nice names.

Our next step will be to add some extra columns. For each campaign, we have the number of clicks and impressions, and we have the amounts 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 straightforward, 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. Let's write a function that calculates all three ratios and adds them to the `DataFrame`:

```
#14
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)
```

Notice 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 though it looks like we're only doing three divisions, there are actually *5140 * 3* divisions, because they are performed for each row. `pandas` does a lot of work for us, and also does a very good job of hiding the complexity of it.

The function `calculate_extra_columns()` takes a `DataFrame` (`df`), and works directly on it. This mode of operation is called **in-place**. This is similar to how the `list.sort()` method sorts a list. You could also say that this function is not pure, which means it has side effects, as it modifies the mutable object it is passed as an argument.

We can take a look at the results by filtering on the relevant columns and calling `head()`:

```
#15
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   17953   52573       500001  0.105146   0.341487  0.035906
1  125884   24575       499999  0.049150   5.122442  0.251769
2  480963   39668       499999  0.079336  12.124710  0.961928
```

Now, we want to verify the accuracy of the results manually for the first row:

```
#16
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)
```

This yields the following output:

```
CTR: 0.10514578970842059 0.10514578970842059
CPC: 0.3414870751146026 0.3414870751146026
CPI: 0.03590592818814362 0.03590592818814362
```

The values match, confirming that our computations are correct. Of course, we wouldn't normally need to do this, but we 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 (this is similar to looking up a key in a dictionary). You can then access each row in the column 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. The duration is important to have, since it allows us to relate data such as the amount spent or number of impressions to the duration of the campaign (we may expect longer running campaigns to cost more and have more impressions). The day of the week can also be useful; for example, some campaigns may be tied to events that happen on particular days of the week (such as sports events that take place on Saturdays or Sundays).

```
#17
def get_day_of_the_week(day):
    return day.strftime("%A")
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)
```

`get_day_of_the_week()` is very simple. It takes a `date` object and formats it as a string, which only contains the name of the corresponding day of the week. `get_duration()` is more interesting. First, notice it takes an entire row, not just a single value. In its body, we perform a subtraction between a campaign's 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 to get the duration in days.

Now, we can introduce the fun part, the application of those two functions. First, we apply `get_day_of_the_week()` to the `Start` column (as a `Series` object); this is similar to what we did with `'user'` and `'cmp_name'`. Next, we apply `get_duration()` 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:

```
#18
df[['Start', 'End', 'Duration', 'Day of Week']].head(3)
```

This gives the following output:

```
        Start         End  Duration Day of Week
0  2021-01-31  2021-04-11        70      Sunday
1  2021-01-09  2022-12-04       694    Saturday
2  2021-11-24  2022-09-21       301   Wednesday
```

So, we now know that between the 9th of January, 2021, and the 4th of December, 2022, there are 694 days, and that the 31st of January, 2021, is a Sunday.

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. We also want to reorder the columns in our `DataFrame` so that they are 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:

```
#19
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]
```

We 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, how about taking a snapshot of `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 autocompletion pop up, so you can see all the possible options.

We're going to save our `DataFrame` in three different formats, just for fun. First, CSV:

```
#20
df.to_csv('df.csv')
```

Then JSON:

```
#21
df.to_json('df.json')
```

And finally, in an Excel spreadsheet:

```
#22
df.to_excel('df.xlsx')
```

> The `to_excel()` method requires the `openpyxl` package to be installed. It is included in the `requirements.txt` file for this chapter, so if you used that to install the requirements, you should have it in your virtual environment.

So, it's extremely easy to save a `DataFrame` in many different formats, and the good news is that the reverse is also true: it's very easy to load a spreadsheet into a `DataFrame` (just use the pandas `read_csv()` or `read_excel()` functions). The programmers behind `pandas` went a long way to make our tasks easier, which is 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, we're not going to go deep into analyzing our data, especially because the data is completely random. However, this example should still be enough to get you started with graphs and other features.

Something we have learned in our lives, and this may come as a surprise to you, is that *looks also count*, so it's very important that when you present your results, you do your best to *make them pretty*.

`pandas` uses the Matplotlib plotting library to draw graphs. We won't be using it directly, except to configure the plot style. You can learn more about this versatile plotting library at `https://matplotlib.org/`.

First, we'll tell the Notebook to render Matplotlib graphs in the cell output frame, rather than in a separate window. We do it with the following:

```
#23
%matplotlib inline
```

Then, we proceed to set up some styling for our plots:

```
#24
import matplotlib.pyplot as plt
plt.style.use(['classic', 'ggplot'])
plt.rc('font', family='serif'})
```

We use the `matplotlib.pyplot` interface to set the plot style. We've chosen to use a combination of the `classic` and `ggplot` style sheets. Style sheets are applied from left to right, so here `ggplot` will override the `classic` style for any style items that are defined in both. We also set the font family used in the plots to `serif`.

Now that our `DataFrame` is complete, let's run `df.describe()` (#25) again. The results should look like this:

	Duration	Budget	Clicks	Impressions	Spent	CTR	CPC	CPI	Age
count	5140.000000	5140.000000	5140.000000	5140.000000	5140.000000	5140.000000	5140.000000	5140.000000	5140.000000
mean	365.923930	496331.855058	40414.236576	499999.523346	249542.778210	0.080829	9.816749	0.499086	55.503891
std	213.233798	289001.241891	21704.136480	2.010877	219168.636408	0.043408	17.649877	0.438338	20.803059
min	1.000000	1017.000000	355.000000	499991.000000	117.000000	0.000710	0.003580	0.000234	18.000000
25%	180.000000	250725.500000	22865.250000	499998.000000	70162.000000	0.045730	1.778724	0.140325	38.000000
50%	369.000000	495957.000000	37103.000000	500000.000000	188704.000000	0.074206	4.977531	0.377409	56.000000
75%	553.000000	741076.500000	55836.000000	500001.000000	381478.750000	0.111673	11.620850	0.762962	73.000000
max	730.000000	999860.000000	98912.000000	500007.000000	984005.000000	0.197824	517.287324	1.968014	90.000000

Figure 13.3: Some statistics for our cleaned-up data

This kind of quick result is perfect for satisfying 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 plot the four pieces of information we have on each campaign: `'Budget'`, `'Spent'`, `'Clicks'`, and `'Impressions'`:

```
#26
df[['Budget', 'Spent', 'Clicks', 'Impressions']].hist(
    bins=16, figsize=(16, 6));
```

We extract those four columns (this will give us another `DataFrame` made with only those columns) and call the `hist()` method to get a histogram plot. We give some arguments to specify the number of bins and figure sizes, and everything else is done automatically.

Since the histogram plot instruction is the last statement in the cell, the Notebook will print its result before drawing the graph.

To suppress this behavior and have only the graph drawn with no printing, we add a semicolon at the end. Here are the graphs:
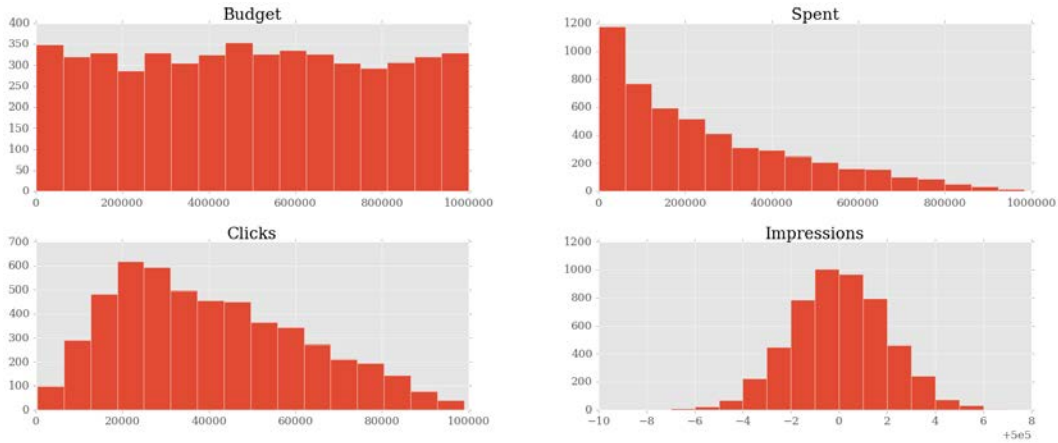
Figure 13.4: Histogram plots of the campaign data

They are beautiful, aren't they? Did you notice the serif font? How about the meaning of those figures? If you go back and take a look at the way we generate the data, you will see that all these graphs make perfect sense:

- **Budget** is selected randomly from an interval, so we expect a uniform distribution. Looking at the graph, that is exactly what we see: it's practically a constant line.

- **Spent** is also uniformly distributed, but its upper limit is the budget, which is not constant. This means we should expect an approximately logarithmic curve that decreases from left to right. Again, that is exactly what the graph shows.

- **Clicks** was generated with a triangular distribution with a mean roughly 20% of the interval size, and you can see that the peak is right there, at about 20% to the left.

- **Impressions** was a Gaussian distribution, which assumes the famous bell shape. The mean was exactly in the middle and we had a standard deviation of 2. You can see that the graph matches those parameters.

Good! Let's plot out the measures we calculated:

```
#27
df[['CTR', 'CPC', 'CPI']].hist(
    bins=20, figsize=(16, 6))
```

Here is the plot representation:
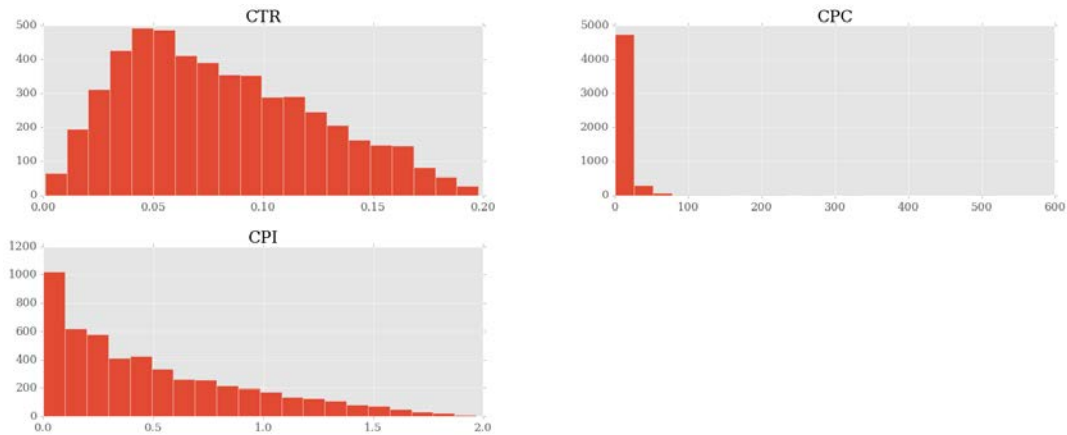
Figure 13.5: Histogram plots of computed measures

We can see that the **CPC** is highly skewed to the left, meaning that most of the **CPC** values are very low. The **CPI** has a similar shape, but is less extreme.

Suppose you want to analyze only a particular segment of the data; how would you do it? We can apply a mask to the `DataFrame` so that we get a new `DataFrame` with only the rows that satisfy the mask condition. It's like applying a global, row-wise `if` clause:

```
#28
selector = (df.Spent > 0.75 * df.Budget)
df[selector][['Budget', 'Spent', 'Clicks', 'Impressions']].hist(
    bins=15, figsize=(16, 6), color='green');
```

In this case, we prepared `selector` to filter out all the rows for which the amount 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 `selector`, we are showing you an alternative way of asking for a `DataFrame` column, by using direct property access (`object.property_name`), instead of dictionary-like access (`object['property_name']`). If `property_name` is a valid Python name, you can use both ways interchangeably.

`selector` is applied in the same way that we access a dictionary with a key. When we apply `selector` to `df`, we get back a new `DataFrame` and we select only the relevant columns from this and call `hist()` again. This time, just for fun, we want the results to be green:
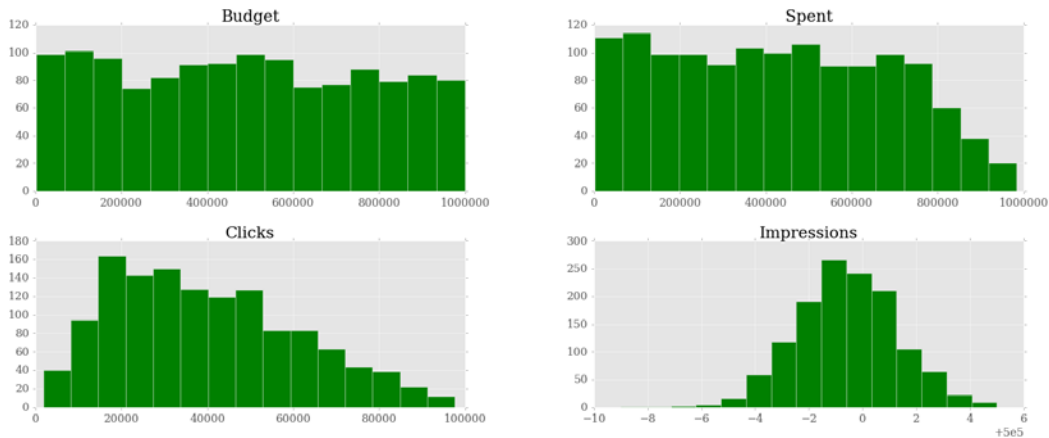
Figure 13.6: Histogram plots of campaign data where at least 75% of the budget was spent

Note that the shapes of the graphs haven't changed much, apart from the **Spent** graph, which is quite different. The reason for this is that we've asked only for the rows where the amount spent is at least 75% of the budget. This means that we're including only the rows where the amount spent is close to the budget. The budget numbers come from a uniform distribution. Therefore, it is quite obvious that the **Spent** graph is now assuming that kind of shape. If you make the boundary even tighter and ask for 85% or more, you'll see the **Spent** graph become more and more like the **Budget** one.

Let's now ask for something different. How about the measure of `'Spent'`, `'Clicks'`, and `'Impressions'` grouped by day of the week?

```
#29
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 an 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 `subplots=True` option makes `plot` draw three separate graphs:
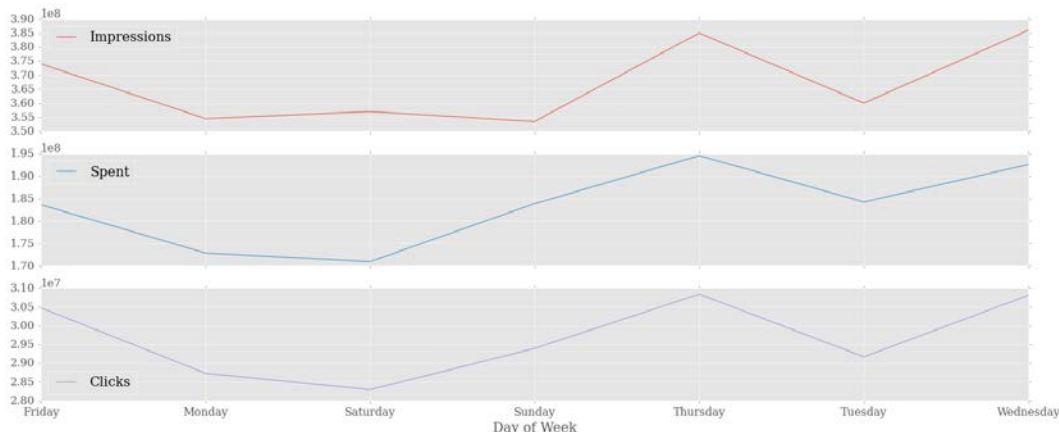
Figure 13.7: Plots of campaign data aggregated by day of the week

Interestingly enough, we can see that most of the action happens on Thursdays and Wednesdays. If this were meaningful data, this would potentially be important information to give to our clients, which is why we're 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? We'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 `'mean'` and the standard deviation (`'std'`):

```
#30
agg_config = {
    'Impressions': ['mean', 'std'],
    'Spent': ['mean', 'std'],
}
df.groupby(['Target Gender', 'Target Age']).agg(agg_config)
```

It's very easy to do. We prepare a dictionary that we'll use as a configuration. Then, we perform a grouping on the `'Target Gender'` and `'Target Age'` columns, and we pass our configuration dictionary to the `agg()` method. The output looks like this:

```
                         Impressions                    Spent
                                mean        std          mean
Target Gender Target Age
B             20-25      499999.513514  2.068970  225522.364865
              20-30      499999.233766  2.372519  248960.376623
              20-35      499999.678161  1.742053  280387.114943
...                               ...        ...           ...
M             45-50      499999.000000  1.490712  323302.200000
              45-55      499999.142857  1.956674  344844.142857
              45-60      499999.750000  1.332785  204209.750000
```

Let's do one more thing before we wrap this chapter up. We want to show you something called a **pivot table**. A pivot table is a way of grouping data, computing an aggregate value for each group, and displaying the result in table form. The pivot table is an essential tool for data analysis, so let's see a simple example:

```
#31
df.pivot_table(
    values=['Impressions', 'Clicks', 'Spent'],
    index=['Target Age'],
    columns=['Target Gender'],
    aggfunc=np.sum
)
```

We create a pivot table that shows us the correlation between `'Target Age'` and `'Impressions'`, `'Clicks'`, and `'Spent'`. These last three will be subdivided according to `'Target Gender'`. The aggregation function (`aggfunc`) used to calculate the results is the `numpy.sum()` function (`numpy.mean` would be the default, had we not specified anything). The result is a new `DataFrame` containing the pivot table:

		Clicks			Impressions			Spent	
Target Gender	B	F	M	B	F	M	B	F	M
Target Age									
20-25	2866528	2736143	3752471	36999964	35499973	42499984	16688655	15609550	22026226
20-30	3377741	3479775	3297034	38499941	39999978	39499933	19169949	17810108	15777111
20-35	2978342	2727771	2981045	43499972	34499979	38499964	24393679	18132970	17653304
20-40	3312590	3269549	3163736	41499964	40499964	37999981	19582113	22317003	19464410

Figure 13.8: A pivot table

Figure 13.8 shows a crop of the output. It's pretty clear and provides very useful information when the data is meaningful.

That's it! We'll leave you to discover more about the wonderful world of IPython, Jupyter, and data science. We 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 create slides and documents with it.

Where do we go from here?

Data science is indeed a fascinating subject. As we 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 look 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, but it's very easy to make mistakes in this field, especially when dealing with big datasets where sampling is mandatory and, therefore, the quality of your analysis 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** (`https://www.numpy.org/`): This is the main 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, the Fourier transform, random number capabilities, and much more.

- **Scikit-learn** (`https://scikit-learn.org/`): This is probably the most popular machine learning library in Python. It has simple and efficient tools for data mining and data analysis, is accessible to everybody, and is reusable in various contexts. It's built on NumPy, SciPy, and Matplotlib.

- **pandas** (`https://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 chapter.

- **IPython** (`https://ipython.org/`)/**Jupyter** (`https://jupyter.org/`): These provide a rich architecture for interactive computing.

- **Matplotlib** (`https://matplotlib.org/`): This is a Python 2-D 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, a Jupyter Notebook, web application servers, and several graphical user interface toolkits.

- **Numba** (`https://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** (`https://bokeh.pydata.org/`): This is 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** (`https://scipy.org/`) and the aforementioned **Anaconda** (`https://anaconda.org/`), that 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 we suggest that you try out ecosystems as well to see whether 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`, and `numPy`.

Of course, having to compress all this information into one single chapter means we could only touch briefly on the subjects we presented. We hope the project we've gone through together has been comprehensive enough to give you an idea of what could potentially be the workflow you follow when working in this field.

The next chapter is dedicated to API development.

14

Introduction to API Development

"The one goal of compassionate communication is to help others suffer less."

– Thich Nhat Hanh

In this chapter, we are going to learn about the concept of an **Application Programming Interface**, or **API**.

We are going to explore this important topic by working on a railway project together. As we go along, we will take the opportunity to also touch briefly upon the following topics:

- HTTP protocol, requests, and responses
- Python type hinting
- The Django web framework

There are entire books dedicated to API design, so it would be impossible for us to tell you everything you need to know about this subject within a single chapter. This consideration brought us to the decision of adopting **FastAPI** as the main technology for this project. It is a well-designed framework that has allowed us to create an API with clean, concise, and expressive code, which we think is quite apt for the purpose of this book.

> We heartily suggest you download and take a look at the source code for this chapter, as it will make it easier for you to wrap your head around the project.

We will also discuss the concept of APIs in a more abstract way, and we hope you will want to study it in depth. As a developer, in fact, you will have to deal with APIs, and likely work on some, throughout your career. Technology changes all the time, and there are plenty of choices available at the moment. It is therefore important to put some effort into learning some of the abstract concepts and ideas we need to design an API, so that we will not feel too lost when it is time to adopt another framework.

Let's now start by spending a moment on the foundation upon which all this is based.

What is the Web?

The **World Wide Web** (**WWW**), or simply the **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**, that 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 several different ways information can be exchanged over the Internet; email, instant messaging, news groups, and so on, 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 diagram in *Figure 14.1*:
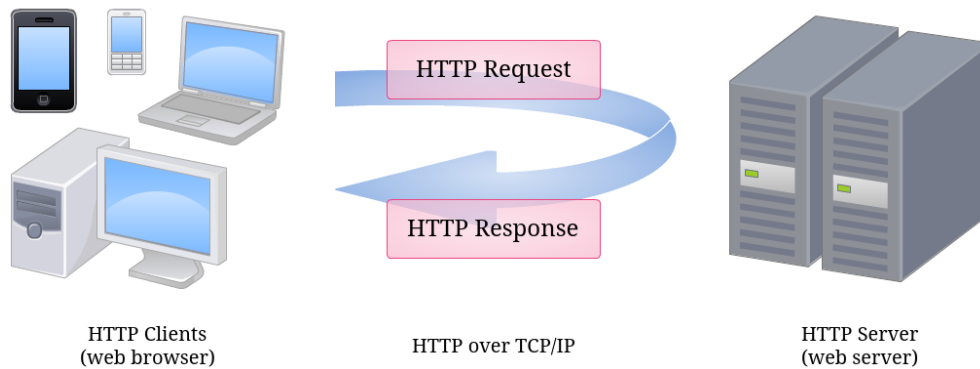
Figure 14.1: A simplified depiction of the HTTP protocol

HTTP is transmitted via the **Transmission Control Protocol/Internet Protocol**, or **TCP/IP**, which provides the tools for a reliable communication exchange over the Internet.

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 technical limitation that exists for very good reasons but, in practice, it is actually possible to browse a website with the illusion of being logged in. When a user logs in, 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.

HTTP defines a set of methods—also known as *verbs*—to indicate the desired action to be performed on a given resource. Each of them is different, but some of them share some common features. We won't be discussing all of them here, only the ones we will use in our API:

- **GET**: The GET method requests a representation of the specified resource. Requests using GET should only retrieve data.

- **POST**: The POST method is used to submit an entity to the specified resource, often causing a change in state or side effects on the server.

- **PUT**: The PUT method requests that the target resource creates or updates its state with the state defined by the representation enclosed in the request.

- **DELETE**: The DELETE method requests that the target resource deletes its state.

Other methods are HEAD, CONNECT, OPTIONS, TRACE, and PATCH. For a comprehensive explanation of all of these, please refer to this page: `https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol`.

The API we are going to write, and the two consumers for it (a console-based one, and a simple local website) all work over HTTP, which means we'll have to write code to perform and handle HTTP requests and responses. We won't keep prepending "HTTP" to the terms "request" and "response" from now on, as we trust there won't be any confusion.

Response status codes

One thing to know about HTTP responses is that they come with a status code, which expresses the outcome of the request in a concise way. Status codes consist of a number and a short description, like 404 Not Found, for example. You can check the complete list of HTTP status codes at `https://en.wikipedia.org/wiki/List_of_HTTP_status_codes`.

Status codes are classified in this way:

- **1xx informational response**: The request was received, continuing process.
- **2xx successful**: The request was successfully received, understood, and accepted.
- **3xx redirection**: Further action needs to be taken in order to complete the request.
- **4xx client error**: The request contains bad syntax or cannot be fulfilled.
- **5xx server error**: The server failed to fulfil an apparently valid request.

When consuming the API, we will receive status codes in the responses, so it's important that you at least have an idea about what they mean.

Type hinting: An overview

Before we start tackling the topic of APIs, let's take a look at Python's type hinting.

One of the reasons we chose FastAPI to create this project is that it is based on type hinting. **Type hinting** was introduced in Python 3.5, by `PEP484` (`https://www.python.org/dev/peps/pep-0484/`). This PEP builds on top of another functionality, **function annotations**, that was introduced in Python 3.0 by `PEP3107` (`https://www.python.org/dev/peps/pep-3107/`).

> PEP484 was strongly inspired by Mypy (`http://mypy-lang.org`), an optional static type checker for Python.

Python is both a **strongly typed** and a **dynamically typed** language.

Strong typing means that variables have a type, and the type matters when we perform operations on variables. Let us explain this concept with an example in an imaginary language:

```
a = 7
b = "7"
a + b == 14
concatenate(a, b) == "77"
```

In this example, we imagine a language that is weakly typed (the opposite of strongly typed). As you can see, the lines between types blur, and the language understands that when we want to sum the integer 7 with the string "7", it should interpret the latter as a number, yielding 14 as a result of the sum.

On the other hand, a concatenation function called with an integer and a string will do the opposite, and cast the integer to a string, producing the string "77" as a result.

Python does not allow this type of behavior, so if we try to sum an integer with a string, or concatenate them, it will complain that their types are not compatible and therefore it cannot carry out the operation we requested. This is good news, as weakly typed languages suffer from sneaky issues that come from the way they approach types.

Dynamic typing means that the type of a variable is determined during runtime. This results in the ability for a variable to have different types at different times, during execution. Take a look at the next snippet, this time in Python:

```
a = 7
a * 2 == 14
a = "Hello"
a * 2 == "HelloHello"
```

In this example, we can see how Python allows us to declare `a` as an integer and perform multiplication by 2 on it. Immediately after that, we re-declare `a` as a string, and the same instruction is now interpreted as a concatenation, because `a`'s type is a string. We learned in *Chapter 2*, *Built-In Data Types*, that `a` is not really changing. What happens is that the *name* `a` is pointed to a different memory location, where the new object—the string "Hello", in this case—is located.

Other languages are different, in that they adopt static typing. Let's see an example of that:

```
String greeting = "Hello";
int m = 7;
float pi = 3.141592;
```

In this example, in a statically typed pretend language, we see that the declaration of variables starts with the specification of their type. This means that they won't be able to ever change type for the whole duration of the program.

There are pros and cons to both approaches. Static typing allows the compiler to notify you about issues like trying to assign a number to a variable declared as a string, before runtime, when the software is being compiled, for example. This is something that Python cannot do because, being dynamically typed, it allows variables to change at will.

On the other hand, static typing can sometimes feel constrictive, whereas dynamic typing allows for more freedom and enables us to employ programming patterns that are not implementable in statically typed languages (or at least, not as easily).

Why type hinting?

Why is it then, that after so many years, Python was extended to support type annotations?

One benefit is that, by having some knowledge about what type a certain variable might be, modern IDEs like PyCharm, Visual Studio Code, and the like, can provide smarter code completion and error checks.

Another benefit is readability. Knowing the type of a certain variable might make it easier to understand and follow the operations the code is performing on that object.

And, of course, there are lots of tools that, thanks to type hinting, can extract information from the code and automatically create documentation for it. That's one of the ways in which FastAPI leverages type hinting, as we will see later.

In summary, this feature has proved to be quite useful, so it's not surprising that type hinting is now being adopted more and more in the Python community.

Type hinting in a nutshell

We will now explore examples of type hinting, although we will look at only what is strictly necessary in order to understand the FastAPI code that will come later.

Let's start with an example of function annotations. Take a look at the following snippet:

```python
def greet(first_name, last_name, age):
    return f"Greeting {first_name} {last_name} of age {age}"
```

The `greet()` function takes three arguments and returns a greeting string. We might want to add some annotations to it, so for example we could, in an exaggerated fashion, do this:

```python
def greet(
    first_name: "First name of the person we are greeting",
    last_name: "Last name of the person we are greeting",
    age: "The person's age"
) -> "Returns the greeting sentence":
    return f"Greeting {first_name} {last_name} of age {age}"
```

This potentially weird-looking code is actually valid Python code, functionally equivalent to the previous version without annotations. We can add any expression after the colon, and even after the parameters' declaration, we can annotate on the return value using the arrow notation.

The ability to do the above enabled the introduction of type hinting. Let's now explore an example of that:

```python
def greet(first_name: str, last_name: str, age: int = 18) -> str:
    return f"Greeting {first_name} {last_name} of age {age}"
```

This code is quite significant. It tells us that `first_name` and `last_name` parameters are supposed to be strings, while `age` is supposed to be an `int` with a default value of `18`. It also tells us that the return value is a string. We say "… parameters are supposed to be…" because nothing prevents us from calling this function with whatever type we want. Let's demonstrate it:

```python
greet(123, 456, "hello")
# returns: 'Greeting 123 456 of age hello'
```

Python has no way of enforcing parameters to be of a type that corresponds to their declaration. However, while writing that call, our IDE suggested to us that `first_name` and `last_name` were supposed to be strings, and `age` an int with default value `18`:


```python
def greet(first_name: str, last_name: str, age: int = 18) -> str:
    return f"Greeting {first_name} {last_name} of age {age}"
```

`[3]` ✓ 0.2s   Python

```
                    (first_name: str, last_name: str, age: int = 18) -> str
▷
1  greet("Fab", )
```
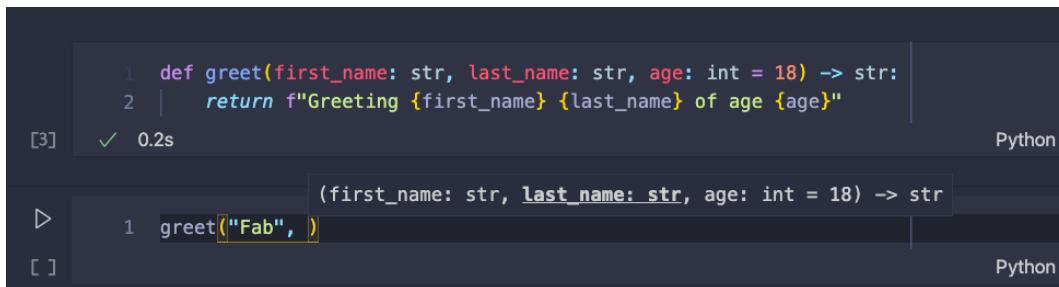
`[ ]`   Python

Figure 14.2: Visual Studio Code (Jupyter Notebook extension) leveraging type hinting

We can use all the standard Python types: *int*, *str*, *float*, *complex*, *bool*, *bytes*, and so on.

There are some Python data structures that can contain other values, such as *list*, *dict*, *set*, and *tuple*. Their internal values can have their own type too. To declare those types, and the types of their internal values, we need to use the `typing` module from the standard library.

This module, which you can find in the documentation at `https://docs.python.org/3/library/typing.html`, is a great place to start when you want to learn more about this topic. Not only does the page document the `typing` module, it also lists all the PEPs related to type hinting at the beginning, so you can gradually explore them and learn more and more about type hinting, and how and why it came to be.

Let's see another example:

```python
from typing import List
def process_words(words: List[str]):
    for word in words:
        # do something with word
```

In this short snippet, we can immediately understand that `words` is supposed to be a list of strings.

```python
from typing import Dict
def process_users(users: Dict[str, int]):
    for name, age in users.items():
        # do something with name and age
```

In this example, we can see another container class, `Dict`. By declaring `users` this way, we can expect it to be a dictionary where keys represent users' names, and values their ages.

> In Python 3.9, `Dict` and `List` have been deprecated in favor of just using `dict` or `list`, which now support the [...] syntax too.

An important class from the `typing` module is `Optional`. `Optional` is used to declare that an argument has a certain type, but that it could also be `None` (that is, it's optional):

```python
from typing import Optional
def greet_again(name: Optional[str] = None):
    if name is not None:
        print(f"Hello {name}!")
    else:
        print("Hey dude")
```

`Optional` plays an important role in FastAPI, as we'll see later, in that it allows us to define parameters that might or might not be used when interrogating an API endpoint.

One last example we want to show you concerns custom types:

```python
class Cat:
    def __init__(self, name: str):
        self.name = name


def call_cat(cat: Cat):
    return f"{cat.name}! Come here!"
```

The `call_cat()` function expects a `cat` argument, which should be an instance of `Cat`. This feature is very important in FastAPI because it allows programmers to declare schemas that represent query parameters, request bodies, and so on.

You should now be equipped to understand the main part of this chapter. We would encourage you to read up a bit more about type hinting in Python though, as it is becoming more and more common to see it in source code.

APIs: An introduction

Before we delve into the details of this chapter's specific project, let's spend a moment talking about APIs in general.

What is an API?

As we mentioned at the beginning of the chapter, API stands for Application Programming Interface.

This interface is designed to act as a connection layer between computers, or computer programs. It is therefore a type of software interface that provides a service to other pieces of software, in contrast to user interfaces, which instead connect computers to people.

An API is normally accompanied by a *specification document*, or *standard*, which describes how to build the API and how it works. A system that meets the specification is said to implement, or expose, the API. The term API can describe either the implementation or the specification.

An API is normally made of different parts, which are the tools that the programmers who write software use to interface with it. These parts are known by different names, the most common of which are *methods*, *subroutines*, or *endpoints* (we will call them endpoints in this chapter). When we use these parts, the technical term for this is *calling* them.

The API specification instructs you on how to call each endpoint, what type of requests to make, which parameters and headers to pass, which addresses to reach, and so on.

What is the purpose of an API?

There are several reasons to introduce an API into a system. One we have already mentioned is to create the means for different pieces of software to communicate.

Another important reason is to allow access to a system by hiding its internal details and implementation, and exposing to the programmers only the parts that it is safe, and necessary, to expose.

The fact that APIs hide the internals of the systems they are interfaced with provides another benefit: if the internal system changes, in terms of technology, languages, or even workflows, the API can change in the way it connects to it, but still provide a consistent interface to the other side, the one that is exposed to the public. If we put a letter into a letterbox, we don't need to know or control how the postal service will deal with it, as long as the letter arrives at the destination. So, the interface (the letterbox) is kept consistent, while the other side (the mailman, their vehicles, technology, workflows, and so on) is free to change and evolve.

Finally, APIs are able to provide necessary features, such as **authentication** and **authorization**, and data validation. Being the layer that is exposed to the world, it makes sense that they are in charge of these tasks.

> *Authentication* means the system has the ability to validate user credentials in order to unequivocally identify them. *Authorization* means the system has the ability to verify what a user has access to.

Users, systems, and data are checked and validated at the border, and if they pass the check, they can interact with the rest of the system (through the API).

This mechanism is conceptually similar to landing at an airport and having to show the police our passports. After that check is successful, we are free to interact with the system, which is the country we landed in, without having to show our passport again.

Given the above, it should not be surprising to know that basically any electronic device we own today, that is connected to the Web, is talking to a (potentially wide) range of APIs in order to perform its tasks.

API protocols

There are different types of API. They can be open to the public, or private. They can provide access to data, or services, or both. APIs can be written and designed using very different methods and standards, and they can employ different protocols.

These are the most common protocols:

- **REST** (**Representational State Transfer**) is a Web services API. REST APIs are a key part of modern Web applications such as Netflix, Uber, and Amazon, among several others. In order for an API to be considered RESTful, it has to adhere to a set of rules. These include concepts like being stateless, providing a uniform interface, and client-server independence.

- **SOAP** (**Simple Object Access Protocol**) is a well-established protocol similar to REST in that it's a type of Web API. SOAP was the first to standardize the way applications should use network connections to manage services. Its specification is very strict compared to that of REST. In general, it also requires more bandwidth.

- **RPC** (**Remote Procedural Call**) is the oldest and simplest type of API. The implementation allows programmers to execute code on the server side by remotely calling a procedure (hence, the name). These types of API are tightly coupled with the implementation of the server they allow access to, so normally they are not made for the public, and maintaining them usually proves to be quite a complex task.

There is plenty of information about API protocols online if you are interested in learning more.

API data-exchange formats

We said that an API is an interface between at least two computer systems. It would be quite unpleasant, when interfacing with another system, to have to shape the data into whatever format that system implements. Therefore, the API, which provides a communication layer between the systems, not only specifies the protocols over which the communication happens, but also which language (or languages) has to be adopted for the data that is exchanged.

The most common data-exchange formats today are **JSON**, **XML**, and **YAML**. We have already used JSON in *Chapter 8*, *Files and Data Persistence*, and we will use it as the format for the API of this chapter too. JSON is widely adopted today by many APIs, and many frameworks provide the ability to translate data from and to JSON, out of the box.

The railway API

Now that we have a working knowledge of what an API is, let's turn to something more concrete.

Before we show you the code, allow us to stress that this code is not production-ready, as that would have been too long and needlessly complex for a book's chapter. However, this code does its job and does it well, and it will allow you to learn quite a lot if you decide to study it, evolve it, and improve it. We will leave suggestions on how to do so at the end of this chapter.

We have a database with some entities that model a railway. We want to allow an external system to perform **CRUD** operations on the database, so we are going to write an API to serve as the interface to it.

> **CRUD** stands for **Create**, **Read**, **Update**, and **Delete**. These are the four basic database operations. Many HTTP services also model CRUD operations through REST or REST-like APIs.

Let's start by taking a look at the project files, so you will have an idea of where things are. You can find them in the folder for this chapter, in the source code:

```
$ tree api_code
api_code
├── api
│   ├── __init__.py
│   ├── admin.py
```

```
|       ├── config.py
|       ├── crud.py
|       ├── database.py
|       ├── deps.py
|       ├── models.py
|       ├── schemas.py
|       ├── stations.py
|       ├── tickets.py
|       ├── trains.py
|       ├── users.py
|       └── util.py
├── dummy_data.py
├── main.py
├── queries.md
└── train.db
```

Within the `api_code` folder, you can find all the files belonging to the FastAPI project. The main application module is `main.py`. We have left the `dummy_data.py` script in the code, which you can use to generate a new `train.db`, the database file. Make sure you read the `README.md` in this chapter's folder for instructions on how to do it. We have also collected a list of queries to the API for you to copy and try out, in `queries.md`.

Within the `api` package, we have the application modules. The database models are in `models.py`, and the schemas used to describe them to the API are in `schemas.py`. The other modules' purposes should be evident from their names: `users.py`, `stations.py`, `tickets.py`, `trains.py`, and `admin.py` all contain the definitions of the corresponding endpoints of the API. `util.py` contains some utility functions; `deps.py` defines the dependency providers; `config.py` holds the configuration settings' boilerplate; and, finally, `crud.py` contains the functions that perform CRUD operations on the database.

> In software engineering, **dependency injection** is a design pattern in which an object receives other objects that it depends on, called dependencies. The software responsible for constructing and injecting those dependencies is known as the *injector*, or *provider*. Hence, a dependency provider is a piece of software that creates and provides a dependency, so that other parts of the software can use it without having to take care of creating it, setting it up, and disposing of it. To learn more about this pattern, please refer to this Wikipedia page: `https://en.wikipedia.org/wiki/Dependency_injection`.

Modeling the database

When preparing the entity-relationship schema for this project, we sought to design something interesting and at the same time simple and well contained. This application considers four entities: *Stations*, *Trains*, *Tickets*, and *Users*. A Train is a journey from one station to another one. A Ticket is a connection between a Train and a User. Users can be passengers or administrators, according to what they are supposed to be able to do with the API.

In *Figure 14.3*, you can see the **entity-relationship** (**ER**) model of the database. It describes the four entities and how they relate to one another:
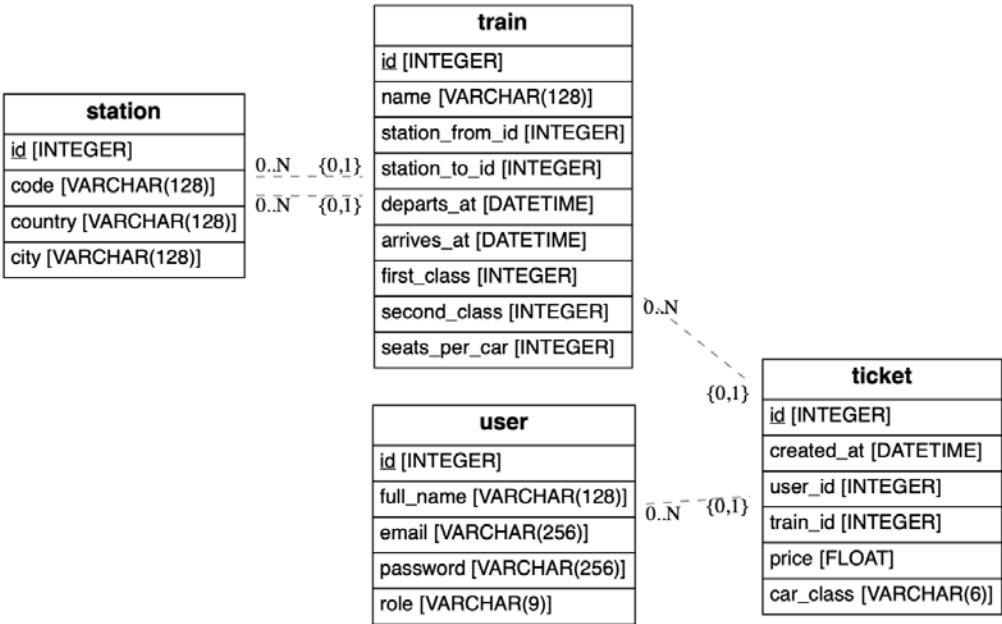
Figure 14.3: ER model of the project database

> *ERAlchemy* (`https://github.com/Alexis-benoist/eralchemy`) is a very useful tool for generating entity-relationship diagrams from databases or SQLAlchemy models. We used it to generate the diagram in *Figure 14.3*.

We have defined the database models using SQLAlchemy, and we have chosen SQLite as the DBMS, for simplicity.

> If you skipped *Chapter 8*, *Files and Data Persistence*, this would probably be a good moment to read it, as there you will learn all you need to know in order to understand the models in this chapter's project.

Let's see the `models` module:

```python
# api_code/api/models.py
import enum
import hashlib
import os
import secrets

from sqlalchemy import (
    Column, DateTime, Enum, Float, ForeignKey, Integer, Unicode
)
from sqlalchemy.orm import relationship

from .database import Base

UNICODE_LEN = 128
SALT_LEN = 64

# Enums
class Classes(str, enum.Enum):
    first = "first"
    second = "second"

class Roles(str, enum.Enum):
    admin = "admin"
    passenger = "passenger"
```

As usual, at the top of the module, we import all the necessary modules. We then define a couple of variables to indicate the default length of Unicode fields (`UNICODE_LEN`) and the length of the salt used to hash passwords (`SALT_LEN`).

> For a refresher on what a salt is, please refer to *Chapter 9*, *Cryptography and Tokens*.

We also define two enumerations: `Classes` and `Roles`, which will be used in the models' definitions.

Let's see the definition of the `Station` model:

```python
# api_code/api/models.py
class Station(Base):
    __tablename__ = "station"

    id = Column(Integer, primary_key=True)
    code = Column(
        Unicode(UNICODE_LEN), nullable=False, unique=True
    )
    country = Column(Unicode(UNICODE_LEN), nullable=False)
    city = Column(Unicode(UNICODE_LEN), nullable=False)

    departures = relationship(
        "Train",
        foreign_keys="[Train.station_from_id]",
        back_populates="station_from",
    )
    arrivals = relationship(
        "Train",
        foreign_keys="[Train.station_to_id]",
        back_populates="station_to",
    )

    def __repr__(self):
        return f"<{self.code}: id={self.id} city={self.city}>"
    __str__ = __repr__
```

The `Station` model is pretty straightforward. There are a few attributes: `id` acts as primary key, and then we have `code`, `country`, and `city`, which combined tell us all we need to know about a station. There are two relationships that link station instances to all the trains departing from, and arriving to, them. The rest of the code defines the `__repr__()` method, which provides a string representation for an instance, and whose implementation is also assigned to `__str__`, so the output will be the same whether we call `str(station_instance)` or `repr(station_instance)`. This technique is quite commonly adopted to avoid code repetition.

Notice we defined a `unique` constraint on the `code` field, so we make sure no two stations with the same code can exist in the database. Big cities like Rome, London, and Paris have more than one train station, so the fields `city` and `country` will be the same, but each station will have its own unique `code`.

Following that, we find the definition of the `Train` model:

```python
# api_code/api/models.py
class Train(Base):
    __tablename__ = "train"

    id = Column(Integer, primary_key=True)
    name = Column(Unicode(UNICODE_LEN), nullable=False)

    station_from_id = Column(
        ForeignKey("station.id"), nullable=False
    )
    station_from = relationship(
        "Station",
        foreign_keys=[station_from_id],
        back_populates="departures",
    )

    station_to_id = Column(
        ForeignKey("station.id"), nullable=False
    )
    station_to = relationship(
        "Station",
        foreign_keys=[station_to_id],
        back_populates="arrivals",
    )

    departs_at = Column(DateTime(timezone=True), nullable=False)
    arrives_at = Column(DateTime(timezone=True), nullable=False)
    first_class = Column(Integer, default=0, nullable=False)
    second_class = Column(Integer, default=0, nullable=False)
    seats_per_car = Column(Integer, default=0, nullable=False)
    tickets = relationship("Ticket", back_populates="train")

    def __repr__(self):
        return f"<{self.name}: id={self.id}>"
    __str__ = __repr__
```

In the `Train` model, we find all the attributes we need to describe a train instance, plus a handy relationship, `tickets`, that gives us access to all the tickets that have been created against a train instance. The `first_class` and `second_class` fields hold how many first- and second-class cars a train has.

We also added relationships to station instances: `station_from` and `station_to`. These allow us to fetch the station instances as objects, instead of only having access to their IDs.

Next up, the `Ticket` model:

```python
# api_code/api/models.py
class Ticket(Base):
    __tablename__ = "ticket"

    id = Column(Integer, primary_key=True)
    created_at = Column(DateTime(timezone=True), nullable=False)
    user_id = Column(ForeignKey("user.id"), nullable=False)
    user = relationship(
        "User", foreign_keys=[user_id], back_populates="tickets"
    )

    train_id = Column(ForeignKey("train.id"), nullable=False)
    train = relationship(
        "Train", foreign_keys=[train_id], back_populates="tickets"
    )

    price = Column(Float, default=0, nullable=False)
    car_class = Column(Enum(Classes), nullable=False)

    def __repr__(self):
        return (
            f"<id={self.id} user={self.user} train={self.train}>"
        )
    __str__ = __repr__
```

A `Ticket` instance has some properties too and includes two relationships, `user` and `train`, that point respectively to the user who bought the ticket, and to the train the ticket is for.

Notice how we have used the `Classes` enumeration in the definition of the `car_class` attribute. This translates to an enumeration field in the database schema definition.

Finally, the `User` model:

```python
# api_code/api/models.py
class User(Base):
    __tablename__ = "user"
```

```python
    pwd_separator = "#"

    id = Column(Integer, primary_key=True)
    full_name = Column(Unicode(UNICODE_LEN), nullable=False)
    email = Column(Unicode(256), nullable=False, unique=True)
    password = Column(Unicode(256), nullable=False)
    role = Column(Enum(Roles), nullable=False)

    tickets = relationship("Ticket", back_populates="user")

    def is_valid_password(self, password: str):
        """Tell if password matches the one stored in DB."""
        salt, stored_hash = self.password.split(
            self.pwd_separator
        )
        _, computed_hash = _hash(
            password=password, salt=bytes.fromhex(salt)
        )
        return secrets.compare_digest(stored_hash, computed_hash)

    @classmethod
    def hash_password(cls, password: str, salt: bytes = None):
        salt, hashed = _hash(password=password, salt=salt)
        return f"{salt}{cls.pwd_separator}{hashed}"

    def __repr__(self):
        return (f"<{self.full_name}: id={self.id} "
                f"role={self.role.name}>")
    __str__ = __repr__
```

The `User` model defines some properties for each user. Note how here we have another enumeration used for the user's role. A user can either be a passenger or an admin. This will allow us to present you with a simple example of how to write an endpoint that allows access only to authorized users.

There are a couple of methods on the `User` model that are used to hash and validate passwords. You might recall from *Chapter 9*, *Cryptography and Tokens*, that passwords should never be stored in a database as they are. So, in our API, when saving a password for a user, we create a hash and store it alongside the salt that was used. In the source code for the book, you will find, at the end of this module, the implementation of the `_hash()` function, which we have omitted here, for brevity.

Main setup and configuration

Let us start from the main access point of the application:

```python
# api_code/main.py
from api import admin, config, stations, tickets, trains, users
from fastapi import FastAPI

settings = config.Settings()
app = FastAPI()

app.include_router(admin.router)
app.include_router(stations.router)
app.include_router(trains.router)
app.include_router(users.router)
app.include_router(tickets.router)

@app.get("/")
def root():
    return {
        "message": f"Welcome to version {settings.api_version} of our
API"
    }
```

This is all the code in the `main.py` module. It imports the various specific endpoint modules, and includes their routers in the main app. By including a router in the main app, we enable the application to serve all the endpoints declared using that specific router. This will be clearer in a moment.

There is only one endpoint in this module, which serves a greeting message. An endpoint is a simple function, in this case, `root`, which contains the code to be executed when the endpoint is called. When and how this function will be invoked depends on the decoration setup. In this case, we have only two pieces of information: with `.get()`, we instruct the API to serve this endpoint when called with a GET request; and with `"/"`, we tell the app that this endpoint will be found at the root, which is the base URL on which the app is running. We will see this in more detail later, when we consume the API. But just to explain here briefly: if this API was served at the base URL `http://localhost:8000`, this endpoint would be called when we requested either `http://localhost:8000` or `http://localhost:8000/` (notice the difference is in the trailing slash) from a browser, for example.

Adding settings

Within the greeting message from the last snippet of code, there is a variable, `api_version`, taken from the `settings` object. All frameworks allow for a collection of settings to be used, in order to configure the app before it runs. We didn't really need to use settings in this example project—we could have just hardcoded those values in the main module—but we thought it was worth showing you how they work:

```python
# api_code/api/config.py
from pydantic import BaseSettings

class Settings(BaseSettings):
    secret_key: str
    debug: bool
    api_version: str

    class Config:
        env_file = ".env"
```

Settings are defined within a Pydantic model (`https://github.com/samuelcolvin/pydantic/`). **Pydantic** is a library that provides data validation and settings management using Python type annotations. In this case, we have three pieces of information within the settings:

- `secret_key`: Used to sign and verify JSON Web Tokens (JWTs).
- `debug`: When set to `True`, it instructs the SQLAlchemy engine to log verbosely, which is helpful when debugging queries, for example.
- `api_version`: The version of the API. We don't really make use of this information, apart from displaying it in the greeting message, but normally the version plays a very important role because the API specification changes according to which version is running.

FastAPI plucks these settings from a `.env` file, as specified by the nested `Config` class within the `Settings` model. Here is how that file looks:

```
# api_code/.env
SECRET_KEY="ec604d5610ac4668a44418711be8251f"
DEBUG=false
API_VERSION=1.0.0
```

> In order for this to work, FastAPI needs help from a library called `python-dotenv`. It is part of this chapter's requirements, so if you have installed them in your virtual environment, you're all set.

Station endpoints

Now we are going to write some FastAPI endpoints. Because this API is CRUD-oriented, there is some repetition in the code. We will therefore show you one example for each of the CRUD operations, and we will do so by using the `Station` endpoints examples. Please refer to the source code to explore the endpoints related to the other models. You will find that they all follow the same patterns and conventions, and the main difference is that they relate to different database models.

We are going to introduce concepts and technical details gradually, alongside the code examples, so the presentation should flow organically.

Reading data

Let us start our exploration with the simplest of all request types: GET. In this case, we are going to get all the stations in the database.

```python
# api_code/api/stations.py
from typing import Optional
from fastapi import (
    APIRouter, Depends, HTTPException, Response, status
)
from sqlalchemy.orm import Session
from . import crud
from .deps import get_db
from .schemas import Station, StationCreate, StationUpdate, Train

router = APIRouter(prefix="/stations")

@router.get("", response_model=list[Station], tags=["Stations"])
def get_stations(
    db: Session = Depends(get_db), code: Optional[str] = None
):
    return crud.get_stations(db=db, code=code)
```

Within the `stations.py` module, we start by importing the necessary objects from the `typing` module, and from `fastapi`. We also import `Session` from `sqlalchemy`, and a few other tools from the local codebase.

The function `get_stations()` is decorated with a `router` object, instead of `app` like it was in the main file. `APIRouter` can be thought of as a mini `FastAPI` class, in that it takes all the same options. We declare `router` and assign a prefix to it (`"/stations"`, in this case), which means all functions decorated with this router become endpoints that can be called at the address `http://localhost:8000/stations`. In this case, the empty string fed to the `.get()` method of the decorator instructs the app to serve this endpoint on the root URL for this router, which will be the concatenation of the base URL and the router prefix, as explained above.

Then we pass `response_model`, which is a list of `Station` instances, the implementation of which we will see later. Finally, `tags`, which is used to organize the documentation (we will see what they do later on).

The function itself takes some arguments, which are a database session, `db`, and an optional string, `code`, which when specified will instruct the endpoint to serve only the stations whose `code` field matches the one provided.

A few things to notice:

- Data coming with the request, such as query parameters, are specified in the endpoint declaration. If the endpoint function requires data to be sent in the body of the request, this is specified using Pydantic models (in this project, they are defined in the `schemas.py` module).

- Whatever an endpoint returns becomes the body of the response. If the `response_model` parameter is not defined, FastAPI will try to serialize the return data to JSON. However, when the response model is set, serialization goes first through the Pydantic model specified in `response_model`, and then from the Pydantic model to JSON.

- In order to use a database session in the body of an endpoint, we are using a dependency provider, which in this case is specified using the `Depends` class, to which we pass the function `get_db()`. This function yields a local database session and closes it when the endpoint call terminates.

- We use the `Optional` class, from the `typing` module, to specify all the optional parameters that may or may not be in the request.

The body of the `get_stations()` function simply returns what its homonym from the `crud` module returns. All the functions that regulate the interaction with the database live in the `crud` module.

This was a design choice that should make this code easier to reuse and test. Moreover, it simplifies reading the entry point code a lot. So, let's see the body of `get_stations()`:

```python
# api_code/api/crud.py
from datetime import datetime, timezone
from sqlalchemy import delete, update
from sqlalchemy.orm import Session, aliased
from . import models, schemas


def get_stations(db: Session, code: str = None):
    q = db.query(models.Station)
    if code is not None:
        q = q.filter(models.Station.code.ilike(code))
    return q.all()
```

Notice how similar this function's signature is to the one for the endpoint that calls it. `get_stations()` returns all instances of `Station`, optionally filtered using `code` (in case it's not `None`).

To start the API, you need to activate your virtual environment and run the following command from the `api_code` folder:

```
$ uvicorn main:app --reload
```

Uvicorn is a lightning-fast **ASGI server**, built on `uvloop` and `httptools`. It is the recommended server for FastAPI. It works seamlessly with both normal and asynchronous functions.

From the ASGI documentation page (`https://asgi.readthedocs.io/en/latest/`):

> ASGI (**Asynchronous Server Gateway Interface**) *is a spiritual successor to* **WSGI** (**Web Server Gateway Interface**), *intended to provide a standard interface between async-capable Python web servers, frameworks, and applications.*
>
> *Where WSGI provided a standard for synchronous Python apps, ASGI provides one for both asynchronous and synchronous apps, with a WSGI backwards-compatibility implementation and multiple servers and application frameworks.*

For this chapter's project, we have chosen to adopt a simple approach, therefore we haven't written any asynchronous code.

Please check out how to write asynchronous endpoints, and the scenarios in which that is the recommended approach, on the official FastAPI documentation page: `https://fastapi.tiangolo.com`.

In the command above, you don't need the `--reload` flag unless you are working on the API source code and want the server to automatically reload whenever a file is saved. It's quite a handy tool to have when developing the API.

If we called this endpoint, this is what we would see:

```
$ http http://localhost:8000/stations
HTTP/1.1 200 OK
content-length: 702
content-type: application/json
date: Thu, 19 Aug 2021 22:11:10 GMT
server: uvicorn

[
    {
        "city": "Rome",
        "code": "ROM",
        "country": "Italy",
        "id": 0
    },
    {
        "city": "Paris",
        "code": "PAR",
        "country": "France",
        "id": 1
    },
    ... some entries omitted ...
    {
        "city": "Sofia",
        "code": "SFA",
        "country": "Bulgaria",
        "id": 11
    }
]
```

Notice the command we are using to call the API: `http`. This is a command that comes with the **Httpie** utility.

> You can find Httpie at `https://httpie.io`. Httpie is a user-friendly command-line HTTP client for the API era. It comes with JSON support, syntax highlighting, persistent sessions, wget-like downloads, plugins, and more. There are other tools to perform requests, such as **curl**, but the choice is up to you, as it makes no difference which tool you use to make your requests from the command line.

The API is served by default at `http://localhost:8000`. You can add arguments to the `uvicorn` command to customize this, but there is no need for that in this case.

The first few lines of the response are information from the API engine. We learn the protocol used was HTTP1.1, and that the request succeeded (status code *200 OK*). We have info on the content length, and its type, which is JSON. Finally, a timestamp and the type of server. We are going to omit the part of this information that will repeat, from now on.

The body of the response is a list of `Station` instances, in their JSON representation, thanks to `response_model=list[Station]`, which we passed to the endpoint decoration.

If we were to search by `code`, for example using the London station one, we could use the following command:

```
$ http http://localhost:8000/stations?code=LDN
```

The above command uses the same URL but adds the `code` query parameter (after the `?`). The result is as follows:

```
$ http http://localhost:8000/stations?code=LDN
HTTP/1.1 200 OK
...
[
    {
        "city": "London",
        "code": "LDN",
        "country": "UK",
        "id": 2
    }
]
```

Notice how we got one match, which corresponds to the London station, but still, it is given back within a list, as expected by the type of `response_model` for this endpoint.

Let us now explore an endpoint dedicated to fetching a single station by ID:

```python
# api_code/api/stations.py
@router.get(
    "/{station_id}", response_model=Station, tags=["Stations"]
)
def get_station(station_id: int, db: Session = Depends(get_db)):
    db_station = crud.get_station(db=db, station_id=station_id)
    if db_station is None:
        raise HTTPException(
            status_code=404,
            detail=f"Station {station_id} not found.",
        )
    return db_station
```

For this endpoint, we configure the router to listen to a GET request, at the URL `http://localhost:8000/stations/{station_id}`, where `station_id` will be an integer. Hopefully, the way URLs are constructed is starting to make sense for you. There is the base part, `http://localhost:8000`, then the prefix for the router, `/stations`, and finally, the specific URL information which we feed to each endpoint, which in this case is `/{station_id}`.

Let's fetch the Kyiv station, with ID 3:

```
$ http http://localhost:8000/stations/3
HTTP/1.1 200 OK
...

{
    "city": "Kyiv",
    "code": "KYV",
    "country": "Ukraine",
    "id": 3
}
```

Notice how this time we got back one object by itself, instead of it being wrapped in a list like it was in the `get_stations()` endpoint. This is in accordance with the response model for this endpoint, which is set to `Station`, and it makes sense, as we are fetching a single object by ID.

The `get_station()` function takes the `station_id`, type-hinted as an integer, and the usual `db` session object. Using type hinting to specify parameters allows FastAPI to do some data validation on the type of the arguments we use when calling an endpoint.

If we were to pass a non-integer value for `station_id`, this would happen:

```
$ http http://localhost:8000/stations/kyiv
HTTP/1.1 422 Unprocessable Entity
...
{
    "detail": [
        {
            "loc": [
                "path",
                "station_id"
            ],
            "msg": "value is not a valid integer",
            "type": "type_error.integer"
        }
    ]
}
```

FastAPI responds to us with useful information: `station_id`, from the path, is not a valid integer. Notice also that the status code is 422 Unprocessable Entity, as opposed to a 200 OK, this time. In general, errors in the four hundreds (*4xx*) express client errors, as opposed to errors in the five hundreds (*5xx*), which express server errors. In this case, we are making a call using an incorrect URL (we're not using an integer), therefore it is an error on our side. Many API frameworks would return a simple status code 400 Bad Request in the same scenario, but FastAPI returns 422 Unprocessable Entity, which is oddly specific. It is easy though, in FastAPI, to customize which status would be returned upon a bad request; there are a few examples in the official documentation.

Let's see what happens when we try to fetch a station with an ID that doesn't exist:

```
$ http http://localhost:8000/stations/100
HTTP/1.1 404 Not Found
...
{
    "detail": "Station 100 not found."
}
```

This time, the URL is correct, in that `station_id` is an integer; however, there is no station with ID 100. The API returns status 404 Not Found, as the response body tells us.

If you go back to the body of this endpoint, you will notice how straightforward its logic is: provided that the arguments passed are correct—in other words, they respect the type—it tries to fetch the corresponding station from the database, by using another simple function from the `crud` module. If the station is not found, it raises an `HTTPException` with the desired status code (404) and a detail that will hopefully help the consumer understand what went wrong. If the station is found, then it is simply returned. The process of returning a JSON serialized version of objects is automatically done for us by FastAPI. The object retrieved from the database is a SQLAlchemy instance of the `Station` class (`models.Station`). That instance is fed to the Pydantic `Station` class (`schemas.Station`), which is used to produce a JSON representation that is then returned by the endpoint.

This might seem complicated, but it is actually a great example of decoupling. The workflow is already there, taken care of for us, and all we need to do is write the little puzzle pieces we need: request parameters, response models, dependencies, and so on.

Creating data

Let's now see something a bit more interesting: how to create a station. First, the endpoint:

```python
# api_code/api/stations.py
@router.post(
    "",
    response_model=Station,
    status_code=status.HTTP_201_CREATED,
    tags=["Stations"],
)
def create_station(
    station: StationCreate, db: Session = Depends(get_db)
):
    db_station = crud.get_station_by_code(
        db=db, code=station.code
    )
    if db_station:
        raise HTTPException(
            status_code=400,
            detail=f"Station {station.code} already exists.",
        )
    return crud.create_station(db=db, station=station)
```

This time, we instruct the router that we want to accept a POST request to the root URL (remember: base part, plus router prefix). We specify the response model to be `Station`, as the endpoint will be returning the newly created object, and we also specify the default status code for the response, which is 201 Created.

The `create_station()` function takes the usual db session and a `station` object. The `station` object is created for us, behind the scenes. FastAPI takes the data from the body of the request and feeds it to the Pydantic schema `StationCreate`. That schema defines all the pieces of data we need to receive, and the result is the `station` object.

The logic in the body follows this flow: it tries to get a station using the code provided; if a station is found, we cannot create one with that data. The code field is defined to be unique, therefore creating a station with the same code would result in a database error. Hence, we return status code 400 Bad Request, informing the caller that the station already exists. In the case the station is not found, we can instead proceed to create it and return it. Let's see the declaration of the Pydantic schemas first:

```python
# api_code/api/schemas.py
from pydantic import BaseModel

class StationBase(BaseModel):
    code: str
    country: str
    city: str

class Station(StationBase):
    id: int
    class Config:
        orm_mode = True

class StationCreate(StationBase):
    pass
```

Schema structure leverages inheritance. It is normal practice to have a base schema that provides functionalities common to all children. Then, each child specifies its needs separately. In this case, in the base schema, we find `code`, `country`, and `city`. When fetching stations, we also want to return the `id`, so we specify that in the `Station` class. Moreover, since this class is used to translate SQLAlchemy objects, we need to tell the model about it, and we do so within the nested `Config` class. Remember that SQLAlchemy is an **object-relational mapping** (**ORM**) technology, so we need to tell the model to turn on the ORM mode by setting `orm_mode = True`.

The `StationCreate` model doesn't need anything extra, so we simply use the `pass` instruction as a body.

Let's now see the CRUD functions for this endpoint:

```python
# api_code/api/crud.py
def get_station_by_code(db: Session, code: str):
    return (
        db.query(models.Station)
        .filter(models.Station.code.ilike(code))
        .first()
    )


def create_station(
    db: Session,
    station: schemas.StationCreate,
):
    db_station = models.Station(**station.dict())
    db.add(db_station)
    db.commit()
    db.refresh(db_station)
    return db_station
```

The `get_station_by_code()` function is fairly simple. It filters through `Station` objects with a case-insensitive match on `code` (that is why we use `ilike`; the "i" in the prefix means [case-]insensitive).

> There are other ways to perform a case-insensitive comparison, which do not involve using `ilike`. This might be the right way to go when performance is important, but for this chapter's purpose, we found the simplicity of `ilike` to be exactly what we needed.

More interesting is the `create_station()` function. It takes a `db` session and a Pydantic `StationCreate` instance. First, we get the station data in the form of a Python dictionary. We know all data must be there, otherwise the endpoint would have already failed during the initial Pydantic validation stage.

Using the data from `station.dict()`, we create an instance of the SQLAlchemy `Station` model. We add it to the database, commit the transaction, then we refresh the object and return it. We need to refresh the object because we want to return it with its `id` too, but unless we refresh it—which means re-fetching it from the database—the `id` won't be there, as it only gets assigned to the object when it is saved to the database.

Let's see this endpoint in action. Notice how we need to specify `POST` to the `http` command, which allows us to send data, in JSON format, within the body of the request. Previous requests were of the GET type, which is the default type for the `http` command. Notice also that we have split the command over two lines due to the book's line length constraints:

```
$ http POST http://localhost:8000/stations \
code=TMP country=Temporary-Country city=tmp-city
HTTP/1.1 201 Created
...
{
    "city": "tmp-city",
    "code": "TMP",
    "country": "Temporary-Country",
    "id": 12
}
```

Great! We created a station with that data. Let's now try again, but this time omitting something mandatory, like the code:

```
$ http POST http://localhost:8000/stations \
country=Another-Country city=another-city
HTTP/1.1 422 Unprocessable Entity
...
{
    "detail": [
        {
            "loc": [
                "body",
                "code"
            ],
            "msg": "field required",
            "type": "value_error.missing"
        }
    ]
}
```

Brilliant. As expected, we get a status code 422 Unprocessable Entity again, because the Pydantic `StationCreate` model validation failed, and the response body tells us why: `code` is missing in the body of the request.

Updating data

To update a station, the logic gets just a bit more complex, but not too much. Let's go through it together. First, the endpoint:

```python
# api_code/api/stations.py
@router.put("/{station_id}", tags=["Stations"])
def update_station(
    station_id: int,
    station: StationUpdate,
    db: Session = Depends(get_db),
):
    db_station = crud.get_station(db=db, station_id=station_id)
    if db_station is None:
        raise HTTPException(
            status_code=404,
            detail=f"Station {station_id} not found.",
        )
    else:
        crud.update_station(
            db=db, station=station, station_id=station_id
        )
        return Response(status_code=status.HTTP_204_NO_CONTENT)
```

The router now is instructed to listen for a PUT request, which is the type you should use to modify a web resource. The URL terminates with the `station_id`, which identifies the station we want to update. The function takes the `station_id`, a Pydantic `StationUpdate` instance, and the usual `db` session.

We start by fetching the desired station from the database. If the station is not found in the database, we simply return status code 404 Not Found, as there is nothing to update. Otherwise, we update the station and return status code 204 No Content, which is the common way to handle the response for a PUT request. We also could have returned 200 OK, but in that case, we should have returned the updated resource within the body of the response.

Let's see the code for the CRUD function responsible for updating a station:

```python
# api_code/api/crud.py
from sqlalchemy import delete, update

def update_station(
    db: Session, station: schemas.StationUpdate, station_id: int
):
```

```
    stm = (
        update(models.Station)
        .where(models.Station.id == station_id)
        .values(station.dict(exclude_unset=True))
    )
    result = db.execute(stm)
    db.commit()
    return result.rowcount
```

The `update_station()` function takes the necessary arguments to identify the station to update, and the station data that will be used to update the record in the database, plus the usual `db` session.

We build a statement using the `update()` helper, from `sqlalchemy`. We specify a `where` clause to filter the station by `id`, and we specify the new values by asking the Pydantic station object to give us a `dict` excluding anything that hasn't been passed to the call. This serves the purpose of allowing partial updates to be executed. If we omitted `exclude_unset=True` from the code, any argument that wasn't passed would end up in the dictionary, set to its default value (`None`).

Normally we would do partial updates by using a PATCH request, but these days it's pretty common to use PUT for both complete and partial updates. For simplicity and brevity, we have done so too, here.

We execute the statement and return the number of rows affected by this operation. We don't use this information in the endpoint body, but it will be a nice exercise for you to try out. We will see how to make use of that information in the endpoint that deletes a station.

The Pydantic model for a station update is this:

```
# api_code/api/schemas.py
from typing import Optional

class StationUpdate(StationBase):
    code: Optional[str] = None
    country: Optional[str] = None
    city: Optional[str] = None
```

All properties are declared as optional because we want to allow the caller to pass only what they wish to update.

Let's use this endpoint on the brand-new station we created in the previous section, with ID 12.

```
$ http PUT http://localhost:8000/stations/12 \
code=SMC country=Some-Country city=Some-city
HTTP/1.1 204 No Content
...
```

Brilliant, we got what we expected. Let's verify that the update was successful:

```
$ http http://localhost:8000/stations/12
HTTP/1.1 200 OK
...
{
    "city": "Some-city",
    "code": "SMC",
    "country": "Some-Country",
    "id": 12
}
```

It was indeed. All three properties of the object with ID 12 have been changed. Let's now try a partial update:

```
$ http PUT http://localhost:8000/stations/12 code=xxx
HTTP/1.1 204 No Content
...
```

This time we only updated the code. Let's verify again:

```
$ http http://localhost:8000/stations/12
HTTP/1.1 200 OK
...
{
    "city": "Some-city",
    "code": "xxx",
    "country": "Some-Country",
    "id": 12
}
```

Wonderful, only `code` was changed, as expected.

Deleting data

Finally, let's explore how to delete a station. As usual, endpoint first:

```python
# api_code/api/stations.py
@router.delete("/{station_id}", tags=["Stations"])
def delete_station(
    station_id: int, db: Session = Depends(get_db)
):
    row_count = crud.delete_station(db=db, station_id=station_id)
    if row_count:
        return Response(status_code=status.HTTP_204_NO_CONTENT)
    return Response(status_code=status.HTTP_404_NOT_FOUND)
```

For the deletion case, we instruct the router to listen for a DELETE request. The URL is the same one we used to get a single station, as well as to update one. The `delete_station()` function takes `station_id` and the `db` session.

Inside the body of the endpoint, we get the number of affected rows from the operation. In this case, if there was at least one, we return status code 204 No Content, which signals to the caller that the deletion was successful. If there were no rows affected, we return status code 404 Not Found. Notice how we could have written the update method exactly like this, making use of the number of affected rows, but we chose a different style so that you had a different example to learn from.

Let's see the CRUD function:

```python
# api_code/api/crud.py
from sqlalchemy import delete, update

def delete_station(db: Session, station_id: int):
    stm = delete(models.Station).where(
        models.Station.id == station_id
    )
    result = db.execute(stm)
    db.commit()
    return result.rowcount
```

This function makes use of the `delete()` helper from `sqlalchemy`. Similar to what we did for the update scenario, we create a statement that identifies a station by ID and instructs for its deletion. We execute the statement and return the number of affected rows.

Let's see this endpoint in action, on a successful scenario first:

```
$ http DELETE http://localhost:8000/stations/12
HTTP/1.1 204 No Content
...
```

We got status code 204 No Content, which tells us the deletion was successful. Let's verify it indirectly, by trying to delete the station with ID 12 one more time. This time we expect the station to be gone, so we want to see a status code 404 Not Found, in return:

```
$ http DELETE http://localhost:8000/stations/12
HTTP/1.1 404 Not Found
...
```

And indeed, this time we received status code 404 Not Found, which means a station with ID 12 wasn't found, proving that the first attempt to delete it was successful. There are a few more endpoints in the `stations.py` module, which you should check out.

The other endpoints we have written are there to create, read, update, and delete users, trains, and tickets. Apart from the fact that they act on different database and Pydantic models, they wouldn't really bring much novelty to this exposition. Therefore, let's instead look at an example of how to authenticate a user.

User authentication

Authentication, in this project, is done via a JSON Web Token. Once again, please refer to *Chapter 9, Cryptography and Tokens*, for a refresher on JWTs.

Let's start from the authentication endpoint, in the `users.py` module.

```python
# api_code/api/users.py
from .util import InvalidToken, create_token, extract_payload

@router.post("/authenticate", tags=["Auth"])
def authenticate(
    auth: Auth,
    db: Session = Depends(get_db),
    settings: Settings = Depends(get_settings),
):
    db_user = crud.get_user_by_email(db=db, email=auth.email)
    if db_user is None:
        raise HTTPException(
```

```
            status_code=status.HTTP_404_NOT_FOUND,
            detail=f"User {auth.email} not found.",
        )

    if not db_user.is_valid_password(auth.password):
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Wrong username/password.",
        )

    payload = {
        "email": auth.email,
        "role": db_user.role.value,
    }
    return create_token(payload, settings.secret_key)
```

This router has the prefix `"/users"`. To authenticate a user, we need to make a POST request to this endpoint. It takes a Pydantic `Auth` schema, the usual `db` session, and the `settings` object, which is needed to provide the secret key that was used to create the token.

If the user is not found, we simply return status code 404 Not Found. If the user is found, but the password provided doesn't correspond to the one in the database record, we can assume wrong credentials, and return status code 401 Unauthorized. Finally, if the user is found and the credentials are correct, we create a token with two simple claims: `email` and `role`. We will use the role to perform authorization functions.

The `create_token()` function is a simple convenience wrapper around `jwt.encode()` that also adds a couple of timestamps to the payload of the token. It is not worth showing that code here. Let's instead see the `Auth` model:

```
# api_code/api/schemas.py
class Auth(BaseModel):
    email: str
    password: str
```

As expected, it is very simple. We authenticate users with their email (which serves as a username) and password. That is why, in the SQLAlchemy `User` model, we have set up a uniqueness constraint on the `email` field. We need each user to have a unique username, and email is a commonly used field for this need.

Let us exercise this endpoint:

```
$ http POST http://localhost:8000/users/authenticate \
email="fabrizio.romano@example.com" password="f4bPassword"
HTTP/1.1 200 OK
...
"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9...g2cQhgyDpmyvCr75Qb_7snYI"
```

Wonderful, we got a token back (we have shortened it for brevity here)!

Now that we have a token, we can put it to use. The user we have authenticated is an admin, so we are going to show you how we could have written the endpoint to delete a station if we wanted to allow only admins to be able to do so. Let's see the code:

```python
# api_code/api/admin.py
...
from .util import is_admin

router = APIRouter(prefix="/admin")

def ensure_admin(settings: Settings, authorization: str):
    if not is_admin(
        settings=settings, authorization=authorization
    ):
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail=f"You must be an admin to access this endpoint.",
        )

@router.delete("/stations/{station_id}", tags=["Admin"])
def admin_delete_station(
    station_id: int,
    authorization: Optional[str] = Header(None),
    settings: Settings = Depends(get_settings),
    db: Session = Depends(get_db),
):
    ensure_admin(settings, authorization)
    row_count = crud.delete_station(db=db, station_id=station_id)
    if row_count:
        return Response(status_code=status.HTTP_204_NO_CONTENT)
    return Response(status_code=status.HTTP_404_NOT_FOUND)
```

In this example, you can see that the endpoint declaration and body are practically the same as their naïve counterpart, with one major difference: before attempting to delete anything, we make sure to call `ensure_admin()`. In the endpoint, we need to grab the authorization header from the request, which is responsible for bearing the token information, so that we can pass it to the `ensure_admin()` function. We do this by declaring it in the function signature, as an optional string that comes from the `Header` object. The `ensure_admin()` function, defined above, delegates to the `util.is_admin()` function, which unpacks the token, verifies its validity, and inspects the `role` field within the payload, to see if it is that of an admin. If all the checks are successful, it returns `True`, or `False` otherwise. The `ensure_admin()` function does nothing when the check is successful, but raises an `HTTPException` with status code 401 Unauthorized when the check is unsuccessful. This means that if, for any reason, the user is not authorized to make this call, the execution of the endpoint's body will immediately stop and return after its first line.

Of course, there are much more sophisticated ways to do authentication and authorization, but it would have been impractical to fit them within the chapter. This simple example, though, is good enough a start to understand how an API would need to be written in order to be secure.

Documenting the API

Documenting APIs is probably one of the most boring activities on the planet. So, we have great news! You won't have to document your FastAPI project, because the documentation is done for you by the framework. We have to thank Python's type hinting, and Pydantic, for this gift that we receive from FastAPI. Make sure your API is running, then open a browser and hit `http://localhost:8000/docs`. A page will open that should look something like this:

Figure 14.4: A partial screenshot of FastAPI self-generated documentation

In *Figure 14.4* you can see a list of endpoints. They are categorized using the `tags` argument, which we have specified in each endpoint declaration. The beauty of this page is that not only can you see each endpoint and inspect its details, but you can also exercise them, using a very friendly interface. Make sure you try it out!

Consuming an API

There are several ways to consume an API. We have seen a very common one: the console. This is a great way to experiment with an API, but it can be a bit cumbersome when the data we need to pass along with the request, including query parameters, headers, and so on, becomes more and more complex. For those situations, there are other options.

One option we often use is to talk to the API from within a Jupyter Notebook. We can use the Requests library (or anything equivalent) to call the endpoints and inspect data from within the Notebook. This solution is quite comfortable.

Another option is to use dedicated tools that provide you with a graphical user interface, like Postman, which comes packed with plenty of functionality. You can find Postman here: `https://www.postman.com`.

Browsers also now come with extensions that allow you to talk to APIs very easily.

Of course, we could also create a console or a GUI application that we can then use to exercise the API. We did something similar in *Chapter 12, GUIs and Scripting*.

Finally, interaction with an API can be done as part of a consumer application, which has its own business logic and calls the API to fetch or manipulate data. Your smartphone is an obvious example of this, as basically almost every app on it talks to an API (more likely, to several APIs).

In the rest of this chapter, we would like to show you an example of the last of these options.

Calling the API from Django

If you have already navigated the Python ecosystem, chances are you have stumbled into Django. **Django** is a web framework, written entirely in Python. It is one of the most widely adopted by the community, and for good reason. It is well written, promotes a sound approach to coding, and it is powerful.

We won't have the space here to fully explain how Django works, so we encourage you to check out its website at `https://www.djangoproject.com`, and go through the simple tutorial in the documentation. You will be amazed by the capabilities of this framework.

For the purpose of this chapter, we only need to know that Django adopts a pattern called **MTV**: **Model**, **Template**, **View**.

In short, Django provides an ORM engine, much like SQLAlchemy, that revolves around models. Models describe database tables and allow Django not only to store and handle data, but to provide a self-generated admin panel that interfaces with the database. This is just one of the many incredibly useful features of this framework. Models are the *M* in MTV.

Views are the other layer, the *V* in MTV. They can be functions or classes, and they run when the user navigates to a specific URL. Each URL corresponds to a view.

Django gives you a plethora of ready-made views, which you can extend by inheritance, to deal with generic pages, forms, lists of objects, and so on. When a view is run, it normally performs a task, which could be to retrieve objects from a database or to interpret data from a form, for example. Normally, the last step of the business logic of a view is to render a page to which it feeds the data that it has collected in its body.

That page, very often, if not always, comes from a template. A template is a file that mixes HTML code (or something equivalent), with Django template language code. This way, a view can collect some data and then pass it to the template. The template will know how to use it, to display it to the users who requested that page. Templates are the *T* in MTV.

In this short section, we are going to explore a couple of views and templates, written in Django, that are interfaced with our railway API. We are going to use the Requests library, from within Django, to access the API. Make sure you check out the source code in the `apic` folder if you want to see the Django project in its entirety.

Let's start from the view that displays stations:

```python
# apic/rails/views.py
...
from urllib.parse import urljoin
import requests
from django.conf import settings
from django.views import generic
from requests.exceptions import RequestException

class StationsView(generic.TemplateView):
    template_name = "rails/stations.html"

    def get(self, request, *args, **kwargs):
        context = self.get_context_data(**kwargs)
        api_url = urljoin(settings.BASE_API_URL, "stations")
        try:
            response = requests.get(api_url)
            response.raise_for_status()
        except RequestException as err:
            context["error"] = err
        else:
            context["stations"] = response.json()
        return self.render_to_response(context)
```

This view is a child of `TemplateView`, one of Django's generic views. In order to render a template with some data, all we need to do is to create a context dictionary and feed it to the `render_to_response()` method call. The template associated with this view is specified as a class attribute.

Notice how, after having gotten a context using the base class method `get_context_data()`, we prepare the URL for the API by using `urljoin()` from the standard library. This function takes care of all the nasty details you need to be careful of when joining URLs.

We then try to talk to the API—in this case, we ask for all stations—and if the communication is successful, we put the JSON decoded response body in `context["stations"]`, and then we render the template.

If the API responds with an error, we capture it in the `err` exception, and we put it in the context under the `error` key. This way, the view won't break and it will still be able to render the page correctly, but the context will be very different than the one for a successful scenario. We need to call `raise_for_status()` in order to make sure we get all those issues that are not related to our ability to talk to the API. If the communication with the API is successful, but for example we get back a status code 500 Internal Server Error, signaling something went wrong on the API internals, the `try/except` block won't catch it because as far as the request is concerned, everything worked fine. So, `raise_for_status()` comes to the rescue, and raises an appropriate exception when the status code is in the 4xx/5xx range.

By using `RequestException`, which is the base for all custom exceptions of the Requests library, we are sure we catch all errors that have to do with the process of communicating with the API through the Requests library. This might not be the best way to handle some scenarios, but for this simple example it is more than enough.

In order to run the Django project, make you sure activate the virtual environment for this chapter and then run this command from within the `apic` folder:

```
$ python manage.py runserver 8080
```

Notice how we set the port to `8080` instead of allowing Django to use the default, `8000`, since the API is already running on that port, which is therefore not free. Please refer to the chapter's `README.md` file to learn how to set up and run all the code for this chapter.

Visiting `http://localhost:8080/stations` on a browser yields this result:
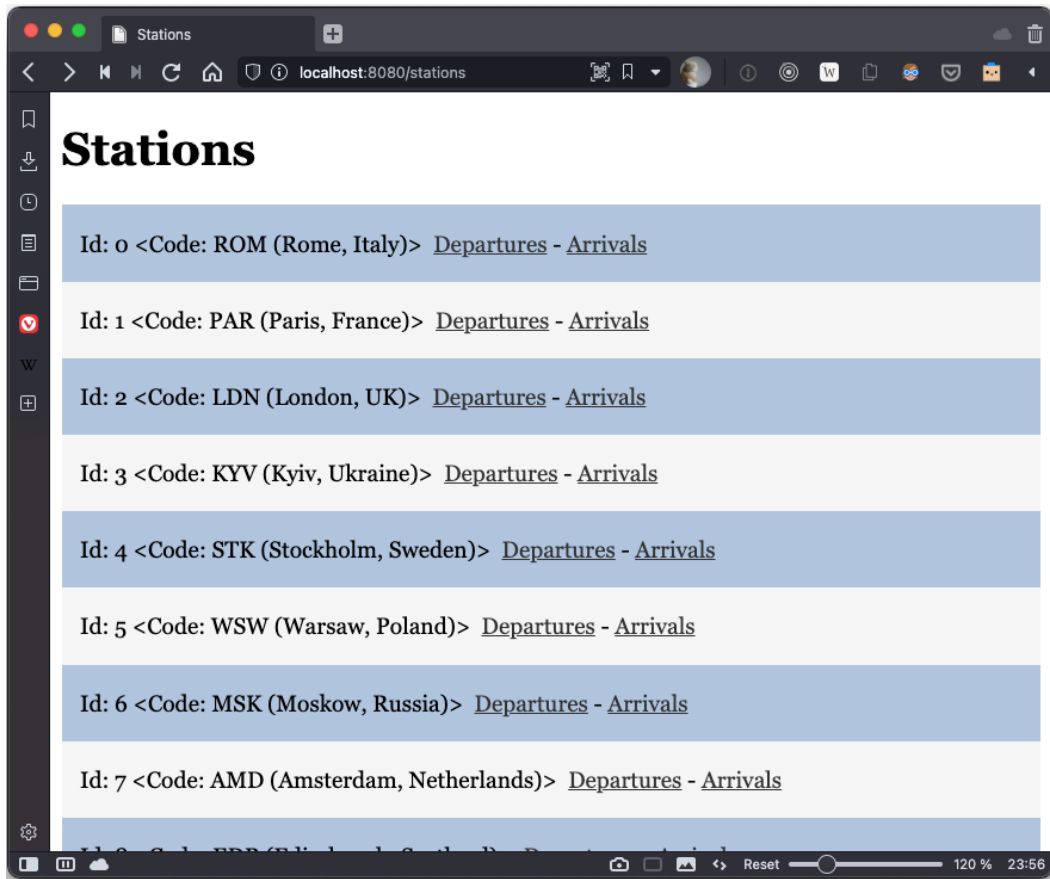
Figure 14.5: Fetching the list of stations from Django

In *Figure 14.5*, you can see the result of rendering the template assigned to the `StationsView`. Let's now see the main part of the template:

```
# apic/rails/templates/rails/stations.html
{% extends "rails/base.html" %}
{% block content %}
{% if stations %}
  <h1>Stations</h1>
{% endif %}
{% for station in stations %}
```

```
    <div class="{% cycle 'bg_color1' 'bg_color2' %}">
      <p>Id: {{ station.id }} &lt;Code: {{ station.code }}
        ({{ station.city }}, {{ station.country }})&gt; 
        <a href="{% url 'departures' station.id %}">Departures</a> -
        <a href="{% url 'arrivals' station.id %}"">Arrivals</a>
      </p>
    </div>
  {% empty %}
    {% if error %}
      <div class=" error">
        <h3>Error</h3>
        <p>There was a problem connecting to the API.</p>
        <code>{{ error }}</code>
        <p>
          (<em>The above error is shown to the user as an example.
            For security reasons these errors are normally hidden
            from the user</em>)
        </p>
      </div>
    {% else %}
      <div>
        <p>There are no stations available at this time.</p>
      </div>
    {% endif %}
  {% endfor %}
  {% endblock %}
```

Notice how Django template tags, which are surrounded by curly braces, are interspersed with the HTML code that designs the structure of the page. The template starts by declaring the fact that it is extending the base template. This is quite common, and it is done so that the base template can host all the common boilerplate, which then doesn't have to be repeated in all templates. The base template declares sections called *blocks*, which other templates can override. In this case, we override the `content` block, which represents the main body of the page.

Remember we had a list of station objects, in the context, under the key `stations`.

So if we have any stations, we display an H1 title, and then we loop over the collection. Each entry gets its own `div`. We use the `cycle` Django tag to alternate row colors. We then simply write down the skeleton of each row and use the notation `{{ variable }}` to render a variable in that part of the template. Tags that represent commands that do something, like `for`, `if`, and `block`, are instead written like this: `{% command %}`.

Each row also creates two links that take you to the departures and the arrivals for each station, respectively. We don't want to hardcode the URL for those pages in a template, so we use the Django `url` tag to calculate those for us by inspecting how the URLs are set up within the `urls` module of the application. This has the advantage that if you change the URL for a certain page, you won't have to amend the templates that link to it, as the links are calculated for you by Django.

Should the collection be empty, the body of the `empty` tag will execute. In there, if there was an error, we display its content, along with some message for the user. Otherwise, we simply state that there are no stations to display. Notice how the `empty` tag belongs to the `for` loop section and helps to keep the logic of this part concise and clean.

Clicking on the departures link for the Rome station leads to this:
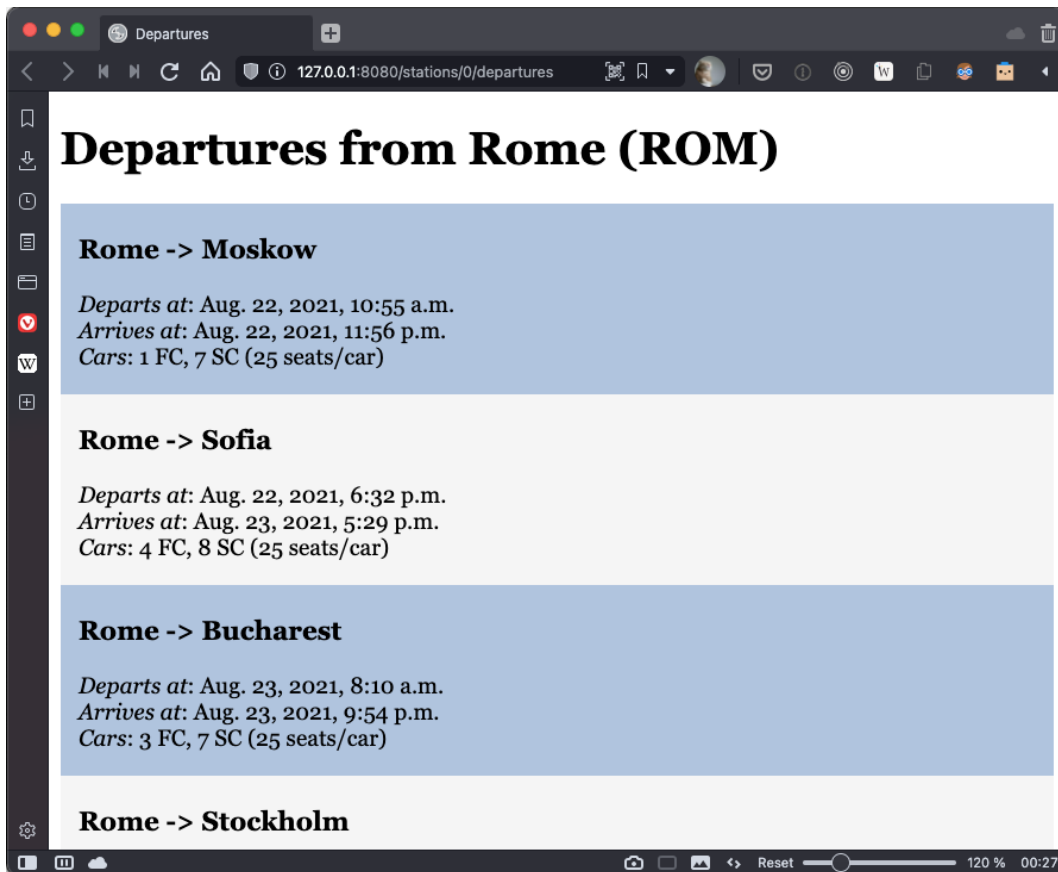
Figure 14.6: All departures from Rome

Let's see the code that renders the page depicted in *Figure 14.6*. First, the view:

```python
# apic/rails/views.py
class DeparturesView(generic.TemplateView):
    template_name = "rails/departures.html"

    def get(self, request, station_id, *args, **kwargs):
        context = self.get_context_data(**kwargs)
        api_url = urljoin(
            settings.BASE_API_URL,
            f"stations/{station_id}/departures",
        )
        try:
            response = requests.get(api_url)
            response.raise_for_status()
        except RequestException as err:
            context["error"] = err
        else:
            trains = prepare_trains(response.json(), "departs_at")
            context["departures"] = trains
        return self.render_to_response(context)

def prepare_trains(trains: list[dict], key: str):
    return list(
        map(
            parse_datetimes,
            sorted(trains, key=itemgetter(key)),
        )
    )

def parse_datetimes(train: dict):
    train["arrives_at"] = datetime.fromisoformat(
        train["arrives_at"]
    )
    train["departs_at"] = datetime.fromisoformat(
        train["departs_at"]
    )
    return train
```

The code is very similar to the one for the `StationsView`. All that changes is the value of `api_url`, which now points to a specific station's departures. We do a bit of data manipulation with the `datetime` information for departures and arrivals.

We parse that into Python `datetime` objects, so it's rendered in a nice, human-readable way in the template. The technique to talk to the API, catching any errors, is the same, and the template is also very similar, to a point that it's not worth going through its code here.

This short section was meant to show you a minimal example of how it is possible to consume an API programmatically, from another application. The Django project we wrote for this chapter only shows stations, departures and arrivals, and users. It also has a page that allows users to authenticate against the API. The idea behind this second mini-project we have added to this chapter is for you to be able to get a good grip on what's needed to talk to the API from a Django project, and then have plenty of room to expand and add functionalities yourself, both learning Django and possibly experimenting with API design at the same time.

Where do we go from here?

Hopefully, by now, you should have wrapped your head around API design. Of course, you will have to check out the source code and spend some time on it, in order to deepen your understanding of this topic. Make sure you experiment, change things, break them, and see what happens. Here are some suggestions for you if you want to learn more on the subject:

- Learn FastAPI. The website offers a great tutorial for beginners and one for advanced programmers, which we recommend. It will cover all the details and the reasons why things have to be done in a certain way, which we couldn't cover in this chapter.

- Experiment with the source code for this chapter. We have written it as simply as we could, so there is lots of room for optimization.

- Learn about WSGI—Web Server Gateway Interface—(`https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface`) and, if you are familiar with asynchronous programming, aWSGI, its asynchronous implementation (`https://pypi.org/project/awsgi/`).

- Enhance the API, adding capabilities like advanced search, filtering, a more sophisticated auth system, background tasks, pagination, sorting, and so on. You can also expand the admin section by adding other endpoints only for admin users.

- Learn about middleware in FastAPI, and about things like **CORS** (**Cross-Origin Resource Sharing**), which are important to know when we run an API in the real world.

- Amend the endpoint that books a ticket so that its logic actually checks that there are free seats on the train. Each train specifies how many first- and second-class cars there are, as well as the number of seats per car. We designed the train model this way specifically to allow you to practice with this exercise. When you are done, you should also write tests to verify your code.

- Learn Django, and when you feel confident with it, try to recreate part of the Railway API using **Django Rest Framework** (`https://www.django-rest-framework.org`), which is a Django-based app used to write APIs.

- Learn to write APIs using other technologies, such as Falcon (`https://falcon.readthedocs.io/`). See what they have in common with FastAPI, as that will improve your understanding of the concepts underlying API design, and the theory behind the frameworks.

- Learn more about **REST** (**Representational State Transfer**) APIs, as they are everywhere these days, but there isn't just one way to write them, so you will have to compare your understanding of this topic with that of your colleagues or peers.

- Learn about APIs in general. Learn about versioning, how to properly work with headers, data formats, and protocols, and so on.

- Finally—and this might be a step you will be able to take in the future— explore the async side of FastAPI. It is interesting and can be slightly more performant if applied to the right use cases.

> Remember to set `DEBUG=true` in the `.env` file, when working with the API, so that you get all the database queries logged automatically in your terminal, and you can check if the SQL code they produce actually reflects your intentions. This is quite a handy tool to have when SQLAlchemy operations become a bit more complex.

There are lots of resources on the Web, so you will be able to gather plenty of knowledge for free. API design is now a very important skill to master, so we can't stress enough how essential it is for you to dig deeper into this subject.

Summary

In this chapter, we have explored the world of APIs. We started with a brief overview of the Web and moved on to a relatively recent Python topic: type hinting. The latter is interesting enough on its own, but in this case, it is a foundational characteristic of the FastAPI framework, so it was important to make sure we covered it in this chapter.

We then discussed APIs in generic terms. We saw different ways to classify them, and the purposes and benefits of their use. We also explored protocols and data-exchange formats.

Finally, we delved into the source code, analyzing a small part of the FastAPI project which was written for this chapter, and exploring different ways in which the resulting API can be consumed.

We concluded the chapter with a series of suggestions for the next steps, which we think are quite important, given the ubiquitous nature of this topic nowadays.

The next chapter discusses packaging Python applications.

15

Packaging Python Applications

"Do you have any cheese at all?"

"No."

– Monty Python, the "Cheese Shop" sketch

In this final chapter, we're going to learn how to create an installable package for your Python project and publish it for others to use.

There are many reasons why you should publish your code. In *Chapter 1, A Gentle Introduction to Python*, we said that one of the benefits of Python is the vast ecosystem of third-party packages that you can install for free, using `pip`. Most of these were created by developers just like you and by contributing with your own projects, you will be helping to ensure that the community keeps thriving. In the long term, it will also help to improve your code, since exposing it to more users means bugs might be discovered sooner. Finally, if you are trying to get a job as a software developer, it really helps to be able to point to projects that you have worked on.

The best way to learn about packaging is to go through the process of creating a package and publishing it. That is exactly what we are going to do in this chapter. To make things more interesting, the project that we'll be working with will be a train schedule application built around the trains API from *Chapter 14, Introduction to API Development*.

In this chapter, we are going to learn:

- How to create a distribution package for your project
- How to publish your package
- About different tools for packaging

Before we dive into our train schedule project, we'll give you a brief introduction to the Python Package Index and some important terminology around Python packaging.

The Python Package Index

The **Python Package Index** (**PyPI**) is an online repository of Python packages, hosted at `https://pypi.org`. It has a web interface that can be used to browse or search for packages and view their details. It also has APIs for tools like `pip` to find and download packages to install. PyPI is open to anybody to register and distribute their projects for free. Anybody can also install any package from PyPI for free.

The repository is organized into **projects**, **releases**, and **distribution packages**. A project is a library or script or application with its associated data or resources. For example, *FastAPI*, *Requests*, *Pandas*, *SQLAlchemy*, and our train schedule application are all projects. `pip` itself is a project as well. A release is a particular version (or snapshot in time) of a project. Releases are identified by version numbers. For example, *pip 21.2.4* is a release of the *pip* project. Releases are distributed in the form of distribution packages. These are archive files, tagged with the release version, that contain the Python modules, data files, and so on, that make up the release. Distribution packages also contain metadata about the project and release, such as the name of the project, the authors, the release version, and dependencies that also need to be installed. Distribution packages are also referred to as **distributions** or just **packages**.

> In Python, the word *package* is also used to refer to an importable module that can contain other modules, usually in the form of a folder containing a `__init__.py` file. It is important not to confuse this type of importable package with a distribution package. In this chapter we'll mostly use the term package to refer to a distribution package. Where there is ambiguity, we'll use the terms *importable package* or *distribution package*.

Distribution packages can be either **source distributions** (also known as **sdists**), which require a build step before they can be installed, or **built distributions**, which only require the archive contents to be moved to the correct locations for an install. The format of source distributions is defined by PEP 517 (`https://www.python.org/dev/peps/pep-0517/`). The standard built distribution format is called a **wheel** and was originally defined in PEP 427. The current version of the wheel specification can be found at `https://packaging.python.org/specifications/binary-distribution-format/`. The wheel format replaced the (now deprecated) **egg** built distribution format.

> The Python Package Index was initially nicknamed the Cheese Shop, after the famous Monty Python sketch that we quoted from at the beginning of the chapter. Thus, the wheel distribution format is not named after the wheels of a car, but after wheels of cheese.

To help understand all of this, let's go through a quick example of what happens when we run `pip install`:

```
$ pip install --no-cache -v -v -v requests==2.26.0
```

We're telling `pip` to install release *2.26.0* of the *requests* project. By passing the `-v` command line option three times, we're telling pip to make its output as verbose as possible. We've also added the `--no-cache` command-line option to force pip to download packages from PyPI and not use any locally cached packages it might have. The output looks something like this (note that we've trimmed the output to fit on the page and omitted several lines):

```
...
1 location(s) to search for versions of requests:
* https://pypi.org/simple/requests/
```

Pip tells us that it has found information about the `requests` project at `https://pypi.org/simple/requests/`. The output continues with a list of all the available distributions of the `requests` project:

```
Found link https://.../requests-0.2.0.tar.gz..., version: 0.2.0
...
Found link https://.../requests-2.26.0-py2.py3-none-any.whl...,
version: 2.26.0
Found link https://.../requests-2.26.0.tar.gz..., version:
2.26.0
```

Now, pip collects the distributions for the release we've requested and downloads the most appropriate package. In this case, that is the wheel `requests-2.26.0-py2.py3-none-any.whl`:

```
Collecting requests==2.26.0
   ...
   Downloading requests-2.26.0-py2.py3-none-any.whl (62 kB)
```

Next, pip extracts the list of dependencies from the package and proceeds to find and download them all in the same way. Once all the required packages have been downloaded, they can be installed:

```
Installing collected packages: urllib3, idna, charset-normalizer,
certifi, requests
...
Successfully installed certifi-2021.5.30 charset-normalizer-2.0.4
idna-3.2 requests-2.26.0 urllib3-1.26.6
```

If pip had downloaded a source distribution for any of the packages (which might happen if no suitable wheel is available), it would have needed to build the package before installing it.

Now that we know the difference between a project, a release, and a package, let's meet our train schedule project so that we can start working on packaging a release and publishing it.

The train schedule project

The project we'll be working on is a simple application for showing train arrival and departure schedules. It allows you to select a station and see a list of all trains departing from or arriving at that station. All the data in the application comes from the trains API we built in *Chapter 14*, *Introduction to API Development*. The application provides both a `tkinter` GUI and a **command-line interface** (**CLI**), both of which were created using the tools and techniques we studied in *Chapter 12*, *GUIs and Scripting*.

> You will need to have the trains API from *Chapter 14* running for the train schedule application to work. We suggest that you open up a second console window and keep the API running there while you work through this chapter.

The project lives in the `train-project` sub-folder in the source code for this chapter. The main (importable) package is called `train_schedule`. We won't go through the code in detail here. Instead, we'll explain how it is structured and you can study the code in more detail yourself. It is all based on the techniques and concepts that you have learned in the book and we have left comments throughout the code to help you understand what each part does. Later on in the chapter, we will look at particular parts of the code in more detail, to explain how they relate to packaging.

Let's use the `tree` command to see how the code is organized:

```
$ tree train_schedule
train_schedule
├── api
│   ├── __init__.py
│   └── schemas.py
├── models
│   ├── __init__.py
│   ├── event.py
│   ├── stations.py
│   └── trains.py
├── views
│   ├── __init__.py
│   ├── about.py
│   ├── config.py
│   ├── dialog.py
│   ├── formatters.py
│   ├── main.py
│   ├── stations.py
│   └── trains.py
├── __init__.py
├── __main__.py
├── cli.py
├── config.py
├── gui.py
├── icon.png
├── metadata.py
└── resources.py
```

We can see that `train_schedule` is an importable Python package with three sub-packages: `api`, `models`, and `views`.

The `api` package defines our interface to the trains API. Within the `api/__init__.py` file you will find a `TrainAPIClient` class, which is responsible for all communication with the API. The `schemas.py` module defines `pydantic` schemas to represent the `Train` and `Station` objects that we receive from the API.

The GUI application is structured according to the **Model-View-Controller** (**MVC**) design pattern. The `models` package contains the *models* that are responsible for managing the station and train data (in the `models/stations.py` and `models/trains.py` modules). The `models/events.py` module implements a callback mechanism that allows other parts of the application to be notified when the data in the models are updated.

> The MVC pattern is a commonly used pattern for developing graphical user interfaces and web applications. You can read more about it at `https://en.wikipedia.org/wiki/Model-view-controller`.

The `views` package implements the *views* that are responsible for presenting the data to the user. The `views/stations.py` module defines a `StationChooser` class that displays stations in a drop-down list so that the user can select the station they're interested in. In `views/trains.py` we have a `TrainsView` class that displays information about trains in a tabular format. `views/formatters.py` defines some helper functions for formatting data. The main application window is defined in `views/main.py`, while `views/dialog.py` contains a base class for dialogs that is used in `views/about.py` and `views/config.py` to create an *About* dialog and a *Configuration* dialog.

The `gui.py` module defines a `TrainApp` class, which is the *controller* that is responsible for coordinating interaction between the models and the views.

The command-line interface of the application is defined in `cli.py`. In `config.py` we define a `pydantic` settings class for handling application configuration. To make our code more portable, we use the very handy `platformdirs` package (see `https://pypi.org/project/platformdirs/`) to store a configuration file in an appropriate location for whatever platform we are running on.

The `resource.py` and `metadata.py` modules contain some code for dealing with data files (like `icon.png`) and metadata included in the distribution package. We'll talk about these in more detail later on in the chapter.

The only module that is left to talk about is `__main__.py`. Within an importable Python package, the module name `__main__.py` gets special treatment. It allows the package to be executed as a script.

Let's see what this file looks like and what happens when we run it:

```python
# train/train_schedule/__main__.py
import sys

from .cli import main as cli_main
from .gui import main as gui_main

if __name__ == "__main__":
    if len(sys.argv) > 1:
        cli_main()
    else:
        gui_main()
```

We have some imports to get the standard library `sys` module and to load the `main` functions from our `cli` and `gui` modules, aliasing them as `cli_main` and `gui_main`. Then we have the same `if __name__ == "__main__"` check that we saw in *Chapter 12*. Finally, we resort to a simple trick to allow us to run either the CLI or GUI versions of the application, depending on whether we have command-line arguments or not. We do this by checking the length of `sys.argv`, which is a list containing the command-line arguments to our script. The length is always at least 1, since `sys.argv[0]` contains the name our script was run with. If the length is greater than 1, it means we do have arguments and we invoke `cli_main`, otherwise we run `gui_main`. Let's see it in action:

```
$ python -m train_schedule
```

> To execute a package with a `__main__.py` file as a script, we have to use the `-m` option to the Python interpreter. This tells the interpreter to import the package first before executing it. Without this option, the interpreter will attempt to run the `__main__.py` file as a standalone script, which will fail because it won't be able to import other modules from the package. You can read more about this at `https://docs.python.org/3/using/cmdline.html`.

You will need to install the dependencies listed in the `requirements/main.txt` file in the chapter's source code to be able to run the application. You can ignore the `requirements/build.txt` file for now. It lists dependencies that we will need later in the chapter to build and publish our distribution packages.

The first time we run the application, we're presented with a configuration dialog prompting us to enter the URL for the train API.

Figure 15.1 shows what this looks like on a Windows machine:
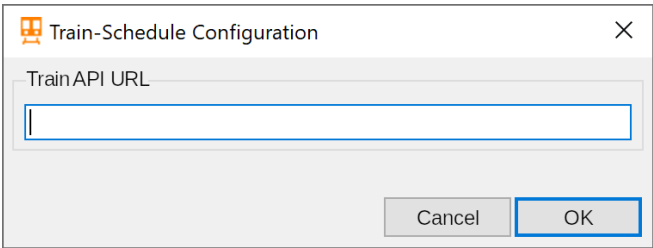
Figure 15.1: The train schedule configuration dialog

Type in `http://localhost:8000` and click **OK** to proceed. The URL will be saved to a configuration file, so next time we run the app we won't be asked to enter it again. If we want to change the URL, we can open the configuration dialog again by clicking on **Preferences...** under the **Edit** menu.

Now we can select a station from the drop-down list to see arrivals and departures. *Figure 15.2* shows what it looks like when we select the station in Rome:
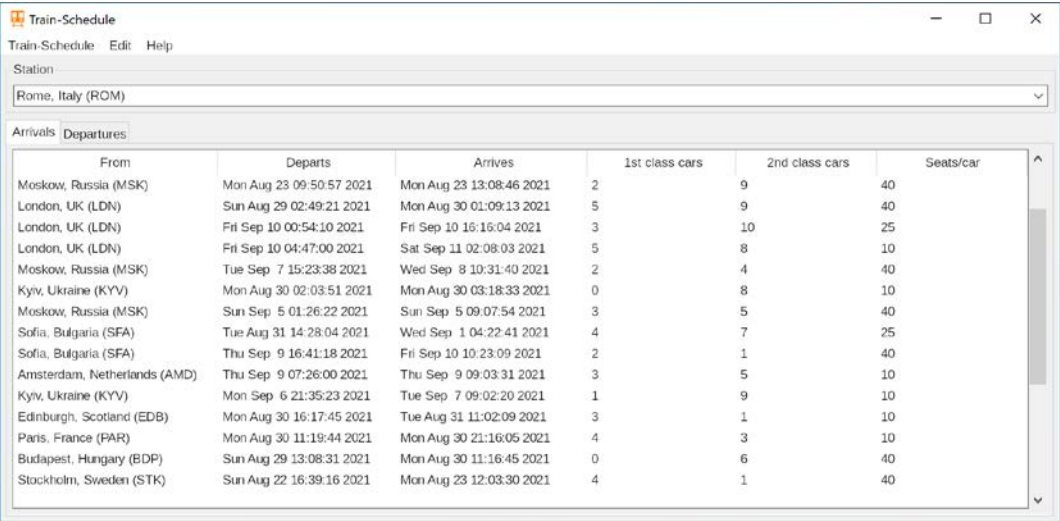
Figure 15.2: The main window of the train schedule application

Before we move on, let's quickly see the command-line interface:

```
$ python -m train_schedule stations
0: Rome, Italy (ROM)
1: Paris, France (PAR)
2: London, UK (LDN)
3: Kyiv, Ukraine (KYV)
```

```
 4: Stockholm, Sweden (STK)
 5: Warsaw, Poland (WSW)
 6: Moskow, Russia (MSK)
 7: Amsterdam, Netherlands (AMD)
 8: Edinburgh, Scotland (EDB)
 9: Budapest, Hungary (BDP)
10: Bucharest, Romania (BCR)
11: Sofia, Bulgaria (SFA)
```

As you can see, we get a printout with the `id`, `city`, `country`, and `code` of each station.

We'll leave it up to you to explore further. If there's something you don't understand, look it up in the official Python documentation or refer back to previous chapters of the book. Some of the techniques we discussed in *Chapter 11*, *Debugging and Profiling*, such as adding `print` statements (or a custom debug function), can also be useful to help you understand what's happening while the code is running.

Now that we're familiar with our project, we can start working on preparing a release and building distribution packages.

Packaging with setuptools

We will be using the `setuptools` library to package our project. `setuptools` is currently the most popular packaging tool for Python. It is an extension of the original, standard library `distutils` packaging system. `setuptools` has many features that are not available in `distutils` and is more actively maintained. Direct use of `distutils` has been actively discouraged for many years. The `distutils` module will be deprecated in Python 3.10 and removed from the standard library in Python 3.12.

In this section, we'll be looking at how to set up our project to build packages with `setuptools`.

Required files

To build a package and publish it, we need to add some files to our project. If you look at the content of the `train-project` folder in the chapter source code, you will see the following files alongside the `train_schedule` folder:

```
CHANGELOG.md LICENSE MANIFEST.in README.md pyproject.toml setup.cfg
setup.py
```

We'll discuss each of these in turn, starting with `pyproject.toml`.

pyproject.toml

This file was introduced by PEP 518 (`https://www.python.org/dev/peps/pep-0518/`) and extended by PEP 517 (`https://www.python.org/dev/peps/pep-0517/`). The aim of these PEPs was to define standards to allow projects to specify their build dependencies, and to specify what build tool should be used to build their packages. For a project using `setuptools`, this looks like:

```
# train-project/pyproject.toml
[build-system]
requires = ["setuptools>=51.0.0", "wheel"]
build-backend = "setuptools.build_meta"
```

Here we have specified that we require at least version 51.0.0 of `setuptools` and any release of the `wheel` project, which is the reference implementation of the wheel distribution format. Note that the `requires` field here does not list dependencies for running our code, only for building a distribution package. We'll talk about how to specify dependencies for running our project later.

The `build-backend` specifies the Python object that will be used to build packages. For `setuptools`, this is the `build_meta` module in the `setuptools` (importable) package.

> The `pyproject.toml` file uses the TOML configuration file format. You can learn more about TOML at `https://toml.io/en/`.

PEP 518 also allows putting configuration for other development tools in the `pyproject.toml` file. Of course, the tools in question also need to support reading their configuration from this file.

```
# train-project/pyproject.toml
[tool.black]
line-length = 66

[tool.isort]
profile = 'black'
line_length = 66
```

We have added configuration for *black*, a popular code formatting tool, and *isort*, a tool for sorting imports alphabetically, to our `pyproject.toml` file. We've configured both tools to use a line length of 66 characters to ensure our code will fit on a book page. We've also configured `isort` to maintain compatibility with `black`.

> You can learn more about `black` and `isort` on their websites at
> `https://black.readthedocs.io/en/stable/index.html` and
> `https://pycqa.github.io/isort/index.html`.

License

You should include a license that defines the terms under which your code is distributed. There are many software licenses that you can choose from. If you're not sure which to use, the website `https://choosealicense.com/` is a useful resource to help you. However, if you are at all in doubt about the legal implications of any particular license or need advice, you should consult a legal professional.

We are distributing our train schedule project under the MIT license. This is a simple license that allows anyone to use, distribute, or modify the code as long as they include our original copyright notice and license.

By convention, the license is included in a text file named `LICENSE` or `LICENSE.txt` although some projects also use other names such as `COPYING`.

README

Your project should also include a *README* file describing the project, why it exists, and even some basic usage instructions. The file can be in plain text format or use a markup syntax like **reStructuredText** or **Markdown**. The file is typically called `README` or `README.txt` if it is a plain text file, `README.rst` for reStructuredText, or `README.md` for Markdown.

Our `README.md` file contains a short paragraph describing the purpose of the project and some simple usage instructions.

Changelog

Although it is not required, it is considered good practice to include a changelog file with your project. This file summarizes the changes made in each release of your project. The changelog is useful for informing your users of new features that are available or of changes in the behavior of your software that they need to be aware of.

Our changelog file is named `CHANGELOG.md` and is written in Markdown format.

setup.cfg

The `setup.cfg` file is the configuration file for `setuptools`. It is an INI style configuration file with `key = value` entries organized into sections, each of which starts with a name in square brackets, for example `[metadata]`.

`setup.cfg` can be used to configure all your project's metadata. It can also define which (importable) packages, modules, and data files need to be included in your distribution packages. We will go through the contents of our `setup.cfg` file in the next few sections of this chapter.

Like `pyproject.toml`, `setup.cfg` can also be used to configure other tools. We have the following bit of configuration for the *flake8* tool at the bottom of our `setup.cfg` file:

```
# train-project/setup.cfg
[flake8]
max-line-length = 66
```

`flake8` is a Python code style checker that can help to point out PEP 8 violations in our code. We have configured it to warn us if any lines in our code are longer than 66 characters. As we've mentioned before, this is shorter than the 80 characters mandated by PEP 8, to make sure our code fits onto the pages of this book.

setup.py

Since `setuptools` implemented PEP 517, the `setup.py` file is no longer required. It is only needed to support legacy tools that are not PEP 517-compliant. If the file exists, it must be a Python script that calls the `setuptools.setup()` function. Our version looks like this:

```
# train-project/setup.py
import setuptools

setuptools.setup()
```

All the options that can be configured in `setup.cfg` can also be passed as keyword arguments to the `setuptools.setup()` function instead. This can be useful for projects where some package options or metadata need to be computed dynamically and cannot be statically configured in `setup.cfg`. This should generally be avoided, though, and as much as possible should be configured via `setup.cfg`.

> Older versions of `setuptools` did not support configuring package options via `setup.cfg`, so projects had to pass everything as `setup` arguments in `setup.py`. Many projects still do this, since they started using `setuptools` before configuration via `setup.cfg` was supported and they have not had a compelling reason to change.

The main reason for still including a `setup.py` file in a new project is to allow us to do an editable install of our project. Instead of building a wheel and installing that in your virtual environment, an editable install just puts a link to your project source folder in your virtual environment. This means that Python will behave as if your package has been installed from an sdist or a wheel, but any changes you make to the code will take effect without needing to rebuild and reinstall.

The main benefit of installing your project in a virtual environment during development is that your code will behave more similarly to when someone else installs it from a distribution package. For example, you won't need to run Python inside your project folder to import your code. This will make it easier for you to spot bugs that can occur if your code makes invalid assumptions about the environment it executes in. An editable install makes it easier to do this, because you won't have to reinstall every time you change something.

> The reason we need a `setup.py` file to do an editable install is that PEP 517 does not support such installs. Therefore, pip has to fall back to the legacy behavior of executing the `setup.py` script directly.

Let's try it. Navigate to the `ch15` folder in the book's source code, activate the virtual environment, and run the following command:

```
$ python -m train_schedule
```

You should get an error like this:

```
/.../ch15/.venv/bin/python: No module named train_schedule
```

Now, let's install and try again. We can do an editable install by passing the `-e` option to `pip install`:

```
$ pip install -e ./train-project
```

After pip has run successfully, we can run our application again:

```
$ python -m train_schedule
```

This time, it should work. You can verify that we really have an editable install by stopping the application, changing something in the code, running it again, and seeing what happens.

> If you look at the contents of the `train-project` folder after running `pip install -e`, you will see there is a new `train_schedule.egg-info` folder. This folder contains the metadata for our editable install. The `egg-info` suffix in the name is a leftover from the old egg distribution format.

MANIFEST.in

By default, only a limited set of files will be included in a source distribution. The `MANIFEST.in` file can be used to add any other files that may be needed to build and install your package from an sdist. This is often required if you have configured `setuptools` to read some package metadata from files included in your project. We'll show you an example of what this looks like in a moment.

You can find details of the exact list of files that are automatically included in source distributions along with details of the syntax of the `MANIFEST.in` file at `https://packaging.python.org/guides/using-manifest-in/`.

Now that we've covered all the files that need to be added to our project, let's take a closer look at the contents of `setup.cfg`.

Package metadata

The first section of our `setup.cfg` file defines our package metadata. Let's go through it a few entries at a time:

```
# train-project/setup.cfg
[metadata]
name = train-schedule
author = Heinrich Kruger, Fabrizio Romano
author_email = heinrich@example.com, fabrizio@example.com
```

The metadata section starts with the `[metadata]` heading. Our first few metadata entries are quite simple. We define the `name` of our project, identify the authors using the `author` field, and list the author email addresses in the `author_email` field.

We've used fake email addresses for this example project, but for a real project you should use your real email address.

PyPI requires that all projects must have unique names. It's a good idea to check this when you start your project, to make sure that no other project has already used the name you want. It's also advisable to make sure your project name won't easily be confused with another; this will reduce the chances of anyone accidentally installing the wrong package.

Next, we have the release version:

```
# train-project/setup.cfg
[metadata]
...
version = 1.0.0
```

The `version` field specifies the version number of your release. You can choose any versioning scheme that makes sense for your project, but it must comply with the rules defined in PEP 440 (`https://www.python.org/dev/peps/pep-0440/`). A PEP 440-compatible version consists of a sequence of numbers, separated by dots, followed by optional pre-release, post-release, or developmental release indicators. A pre-release indicator can consist of the letter `a` (for *alpha*), `b` (for *beta*), or `rc` (for *release-candidate*), followed by a number. A post-release indicator consists of the word `post` followed by a number. A developmental release indicator consists of the word `dev` followed by a number. A version number without a release indicator is referred to as a *final* release. For example:

- `1.0.0.dev1` would be the first developmental release of version 1.0.0 of our project
- `1.0.0.a1` would be our first alpha release
- `1.0.0.b1` is a beta release
- `1.0.0.rc1` is our first release candidate
- `1.0.0` is the final release of version 1.0.0
- `1.0.0.post1` is our first post-release

Developmental releases, pre-releases, final releases, and post-releases with the same main version number are ordered as in the list above.

Popular versioning schemes include **semantic versioning**, which aims to convey information about compatibility between releases through the versioning scheme, and **date-based versioning**, which typically uses the year and month of a release to indicate the version.

> Semantic versioning uses a version number consisting of three numbers, called the *major*, *minor*, and *patch* versions, separated by dots. This results in a version that looks like `major.minor.patch`. If a new release is completely compatible with its predecessor, only the patch number is incremented; usually such a release only contains small bug fixes. For a release that adds new functionality without breaking compatibility with previous releases, the minor number should be incremented. The major number should be incremented if the release is incompatible with older versions. You can read all about semantic versioning at `https://semver.org/`.

The next few entries in our `setup.cfg` are descriptions of our project:

```
# train-project/setup.cfg
[metadata]
...
description = A train app to demonstrate Python packaging
long_description = file: README.md, CHANGELOG.md
long_description_content_type = text/markdown
```

The `description` field should be a short, single sentence summary of the project, while `long_description` can contain a longer, more detailed description. We've told `setuptools` to use the concatenation of our `README.md` and `CHANGELOG.md` files as the long description. The `long_description_content_type` specifies the format of the long description; in our case we use `text/markdown` to specify that we used Markdown format.

We need our `README.md` and `CHANGELOG.md` files to be included in the source distribution, so that they can be added to the long description in the package metadata when we build or install from a source distribution. `README.md` is one of the filenames that `setuptools` automatically includes in the source distribution, but `CHANGELOG.md` is not. Therefore, we have to explicitly include it in our `MANIFEST.in` file:

```
# train-project/MANIFEST.in
include CHANGELOG.md
```

The next metadata entries in our `setup.cfg` file are some URLs for our project:

```
# train-project/setup.cfg
[metadata]
...
url = https://github.com/PacktPublishing/Learn-Python-Programming-
Third-Edition
project_urls =
    Learn Python Programming Book = https://www.packtpub.com/...
```

The `url` field contains the homepage for your project. It's quite common for projects to use this to link to their source code on a code hosting service such as GitHub or GitLab, which is what we've done here. The `project_url` field can be used to specify any number of additional URLs. The URLs can either be entered on a single line as comma-separated `key = value` pairs, or as one `key = value` pair per line as we have done. This is often used to link to online documentation, bug trackers, or other websites related to the project. We've used this field to add a link to information about our book on the publisher's website.

The project's license should also be specified in the metadata:

```
# train-project/setup.cfg
[metadata]
...
license = MIT License
license_files = LICENSE
```

The `license` field is used to name the license the project is distributed under, while the `license_files` field lists files related to the project license that should be included in the distribution. Any files listed here are automatically included in the distribution and do not need to be added to the `MANIFEST.in` file.

Our last couple of metadata entries are meant to help potential users find our project on PyPI:

```
# train-project/setup.cfg
[metadata]
...
classifiers =
    Intended Audience :: End Users/Desktop
    License :: OSI Approved :: MIT License
    Operating System :: MacOS
    Operating System :: Microsoft :: Windows
    Operating System :: POSIX :: Linux
    Programming Language :: Python :: 3
    Programming Language :: Python :: 3.8
    Programming Language :: Python :: 3.9
    Programming Language :: Python :: 3.10
keywords = trains, packaging example
```

The `classifiers` field can be used to specify a list of *trove classifiers*, which are used to categorize projects on PyPI. The PyPI website allows users to filter by trove classifier when searching for projects. Your project's classifiers must be chosen from the list of official classifiers at `https://pypi.org/classifiers/`.

We've used classifiers to indicate that our project is aimed at desktop end users, that it's released under the MIT license, that it works on macOS, Windows, and Linux, and that it's compatible with Python 3 (specifically versions 3.8, 3.9, and 3.10). Note that the classifiers are there purely to provide information to users and help them find your project on the PyPI website. They have no impact on which operating systems or Python versions your package can actually be installed on.

The keywords field can be used to provide additional keywords to help users find your project. Unlike the classifiers, there are no restrictions on what keywords you can use.

Accessing metadata in your code

It is often useful to be able to access your distribution metadata in your code. For example, many libraries expose their release version as a __version__ or VERSION attribute. This allows other packages to adapt themselves to the version of the library that is installed (for example, to work around a known bug in a particular version). Having access to the metadata in the code is one way of doing this, without needing to keep the version up to date in two places.

In our project, we use the project name, author, description version, and license information from the metadata in the *About* dialog of our application. Let's see how we did it:

```python
# train-project/train_schedule/metadata.py
from importlib.metadata import PackageNotFoundError, metadata


def get_metadata():
    try:
        meta = metadata(__package__)
    except PackageNotFoundError:
        meta = {
            "Name": __package__.replace("_", "-"),
            "Summary": "description",
            "Author": "author",
            "Version": "version",
            "License": "license",
        }
    return meta
```

We use the metadata function from the importlib.metadata standard library module to load the package metadata.

To get the metadata, we have to give the metadata the name of our (importable) package. The interpreter makes this available to us via the `__package__` global variable.

> The `importlib.metadata` module was added in Python 3.8. If you need to support older versions of Python, check out the `importlib-metadata` project: `https://pypi.org/project/importlib-metadata/`.

The `metadata` function returns a `dict`-like object with the metadata. The keys are similar to, but not quite the same as, the names of the `setup.cfg` entries we use to define the metadata. Details of all the keys and their meanings can be found in the metadata specification at `https://packaging.python.org/specifications/core-metadata/`.

The metadata can only be accessed if the package has been installed. If the package is not installed, we get a `PackageNotFoundError`. To make sure it's possible to run the code even if it is not installed, we catch the exception and supply some dummy values for metadata keys that we know we want to use.

We use the metadata in our `__init__.py` file, to set some global variables:

```python
# train-project/train_schedule/__init__.py
from .metadata import get_metadata

_metadata = get_metadata()

APP_NAME = _metadata["Name"]
APP_TITLE = APP_NAME.title()
VERSION = _metadata["Version"]
AUTHOR = _metadata["Author"]
DESCRIPTION = _metadata["Summary"]
LICENSE = _metadata["License"]

ABOUT_TEXT = f"""{APP_TITLE}

{DESCRIPTION}

Version: {VERSION}
Authors: {AUTHOR}
License: {LICENSE}
Copyright: © 2021 {AUTHOR}"""
```

If we run the application without installing it first, and we click on **About…** in the **Help** menu, we can see our dummy values in the *About* dialog. *Figure 15.3* shows what this looks like:
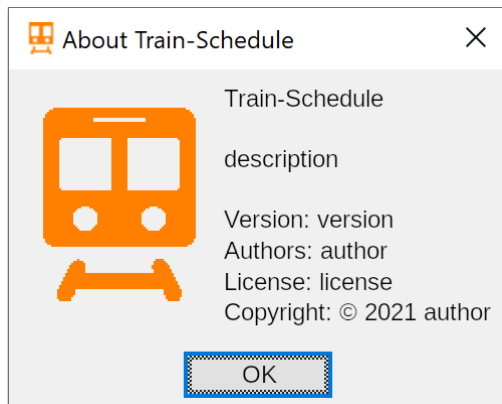
Figure 15.3: The train schedule About dialog

It's not ideal, but during development it's at least good enough for us to be able to see that the dialog works as it should and that we're displaying the correct dummy values in the correct places. It would be better to see the dialog as our users will see it when they have installed our package from PyPI. That way we can be sure that the metadata is read and displayed correctly.

If we do an editable install and run the application again, the *About* dialog looks like the image in *Figure 15.4*:
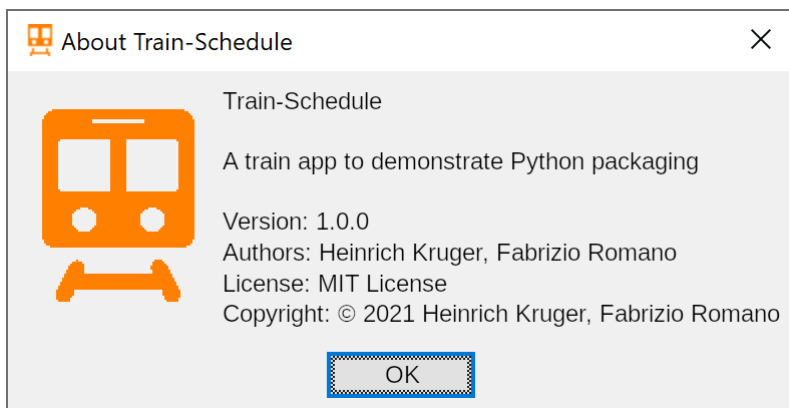
Figure 15.4: The About dialog when the app has been installed

That looks much better. This is also what our users will see when they install and run the application.

That brings us to the end of the metadata section of our `setup.cfg` file. Next up we have the `[options]` section, which is used to define the package contents and dependencies. Let's start by looking at how to specify the package contents.

Defining the package contents

The most important content we need to add to our distribution is, of course, our code. We do this by using the `packages` and `py_modules` options for `setuptools`. The `packages` option takes a list of (importable) packages that need to be added to the distribution. You have to list both the top-level packages (for example `train_schedule`) and every sub-package (like `train_schedule.api`, `train_schedule.models`, and so on). In bigger projects this can be a serious nightmare, especially if you decide to rename or reorganize your packages. Fortunately, there is a solution. You can use the `find:` directive to tell `setuptools` to automatically include any importable packages it finds in your project folder.

Let's see what this looks like:

```
# train-project/setup.cfg
[options]
packages = find:
```

That's quite easy, isn't it? If you need to, you also have the option of configuring where in your project folder `setuptools` looks for packages and specify packages that should be included or excluded from the search. For more detail, please consult the `setuptools` documentation on this topic at `https://setuptools.readthedocs.io/en/latest/userguide/package_discovery.html`.

The `py_modules` option lets you specify a list of Python modules that do not belong to importable packages, that should be included in your distribution. This should be a list of top-level Python files, without the `.py` extension. The `py_modules` option does not support a `find:` directive like the `packages` option does. This is not really a problem, because it is quite rare for projects to have more than a couple of modules that do not belong to packages (our project doesn't have any).

Sometimes, you also need to include non-code files in your distribution. For our project, we want to include the `icon.png` file, so that we can display it in the title bar of our windows and in the *About* dialog. The easiest way to do this is to use the `include_package_data` option in your `setup.cfg` file:

```
# train-project/setup.cfg
[options]
...
include_package_data = True
```

You also have to list the files you want to distribute in your `MANIFEST.in` file:

```
# train-project/MANIFEST.in
include train_schedule/icon.png
```

Note that the data files that you include *must* be placed inside an importable package.

Accessing package data files

Now that we can distribute data files in our package, we need to know how to access those files when our package is installed in a virtual environment on a user's computer. In your development environment, it is easy and therefore quite tempting to just hard-code the path to the file relative to your project folder. This won't work when your code is installed in a virtual environment and run from outside your project folder. To make matters worse, it is not even guaranteed that the data file will exist as a separate file on the filesystem. For example, Python has the ability to import packages and modules from ZIP archives.

Fortunately, Python does provide a mechanism to access data files (referred to as **resources**) from within packages. The tools to do this can be found in the `importlib.resources` module. Let's see an example from our project:

```python
# train-project/train_schedule/resources.py
from importlib import resources

def load_binary_resource(name):
    return resources.read_binary(__package__, name)
```

We have defined our own `load_binary_resource()` helper function, which uses `importlib.resources.read_binary()` to read binary data from a package resource. To do this, we have to supply the name of the package and the name of the resource, which is just the name of the data file. This function is analogous to:

```python
with open(resource, "rb") as stream:
    return stream.read()
```

except that Python is doing more work behind the scenes, since it's not guaranteed that the resource actually exists as a file on disk.

Let's see how we use our `load_binary_resource()` helper function in our `train_schedule/views/main.py` module to load the `icon.png` file (note that we've omitted some code here to only show you the relevant bits):

```python
# train-project/train_schedule/views/main.py
import tkinter as tk
from ..resources import load_binary_resource

ICON_FILENAME = "icon.png"

class MainWindow:
    def __init__(self):
        self.root = tk.Tk()
        self._set_icon()

    def _set_icon(self):
        """Set the window icon"""
        self.icon = tk.PhotoImage(
            data=load_binary_resource(ICON_FILENAME)
        )
        self.root.iconphoto(True, self.icon)
```

We import the `tkinter` module and our `load_binary_resource()` function. We use a global variable `ICON_FILENAME` to define the name of the icon file. The `_set_icon()` method of the `MainWindow` class calls `load_binary_resource()` to read the image data and uses it to create a `tkinter` `PhotoImage` object. We pass this `PhotoImage` object to the `self.root` Tk object's `iconphoto()` method to set the window icon.

As you can see, it's just as easy to load data from a package data file in our distribution as it is to read from any other file. The `importlib.resources` module has several other useful functions; we encourage you to read about them in the standard library documentation. Right now, though, it's time for us to see how to specify dependencies for our package.

Specifying dependencies

As we saw at the beginning of the chapter, a distribution package can provide a list of projects it depends on and pip will ensure that releases of those projects are installed when it installs the package. Of course, this means that we need to specify our dependencies to `setuptools` when we build a package.

We do this by using the `install_requires` option:

```
# train-project/setup.cfg
[options]
...
install_requires =
    platformdirs>=2.0
    pydantic>=1.8.2,<2.0
    requests~=2.0
```

As you can see, our project depends on the `platformdirs`, `pydantic`, and `requests` projects. We can also use **version specifiers** to indicate which releases of dependencies we require. Besides the normal Python comparison operators, version specifiers can also use ~= to indicate a *compatible release*. The compatible release specifier is a way of indicating releases that may be expected to be compatible under a semantic versioning scheme. For example, in our case `requests~=2.0` means that we require any 2.x version of the `requests` project, from 2.0 up to 3.0 (not included). A version specifier can also take a comma-separated list of version clauses that must all be satisfied. For example, `pydantic>=1.8.2,<2.0` means that we want at least version 1.8.2 of `pydantic`, but not version 2.0 or greater. Note that this is not the same as `pydantic~=1.8.2`, which would mean at least version 1.8.2, but not version 1.9 or greater. For the full details of the dependency syntax and how versions are matched, please refer to PEP 508 (`https://www.python.org/dev/peps/pep-0508/`).

You should be careful not to make your `install_requires` version specifiers too strict. Bear in mind that your package is likely to be installed alongside various other packages in the same virtual environment. This is particularly true of libraries or tools for developers. Allowing as much freedom as possible in the required versions of your dependencies means that projects that depend on yours are less likely to encounter dependency conflicts between your package and those of other projects they depend on. Making your version specifiers too restrictive also means that your users won't benefit from bug fixes or security patches in one of your dependencies unless you also publish a new release to update your version specifier.

Apart from dependencies on other projects, you can also specify which versions of Python your project requires. In our project, we use features that were added in Python 3.8, so we specify that we require at least Python 3.8:

```
# train-project/setup.cfg
[options]
...
python_requires = >=3.8
```

As with `install_requires`, it is generally best to avoid limiting the Python versions you support too much. You should only restrict the Python version if you know your code won't work on all actively supported versions of Python 3.

> You can find a list of *Active Python Releases* on the official Python download page: `https://www.python.org/downloads/`.

You should make sure that your code does actually work on all the versions of Python and the dependencies that you support in your setup configuration. One way of doing this is to create several virtual environments with different Python versions and different versions of your dependencies installed. Then you can run your test suite in all these environments. Doing this manually would be very time consuming. Fortunately, there are tools that will automate this process for you. The most well-known of these tools is called *tox*. You can find out more about it at `https://tox.readthedocs.io/en/latest/`.

You can also specify optional dependencies for your package. Pip will only install such dependencies if a user specifically requests it. This is useful if some dependencies are only required for a feature that many users are not likely to need. Users who want the extra feature can then install the optional dependency and everyone else gets to save disk space and network bandwidth. For example, the PyJWT project, which we used in *Chapter 9, Cryptography and Tokens*, depends on the cryptography project to sign JWTs using asymmetric keys. Many users of PyJWT do not use this feature, so the developers made cryptography an optional dependency.

Optional (or extra) dependencies are specified in the `[options.extras_require]` section of the `setup.cfg` file. This section can contain any number of named lists of optional dependencies. These lists are referred to as *extras*. In our project we have one extra, called `dev`:

```
# train-project/setup.cfg
[options.extras_require]
dev =
    black
    flake8
    isort
    pdbpp
```

This is a common convention for listing tools that are useful during development of a project as optional dependencies. Many projects also have an extra `test` dependency, for installing packages that are only needed to run the project's test suite.

To include optional dependencies when installing a package, you have to add the names of the extras you want in square brackets when you run `pip install`. For example, to do an editable install of our project with the `dev` dependencies included, you can run:

```
$ pip install -e ./train-project[dev]
```

We are nearly done with our `setuptools` configuration. There's only one more section to add before we can build our package and publish it.

Entry points

So far, we've been running our application by typing:

```
$ python -m train_schedule
```

This is not particularly user-friendly. It would be much better if we could run our application by just typing:

```
$ train-schedule
```

The good news is that this is possible. We can achieve this by configuring script **entry points** for our distribution. A script entry point is a function that we want to be able to execute as a command-line or GUI script. When our package is installed, pip will automatically generate scripts that import the specified functions and run them.

We configure entry points in the `[options.entry_points]` section of the `setup.cfg` file. Let's see what ours looks like:

```
# train-project/setup.cfg
[options.entry_points]
console_scripts =
    train-schedule-cli = train_schedule.cli:main
gui_scripts =
    train-schedule = train_schedule.gui:main
```

The `entry_points` configuration contains a number of *groups*, each of which contains a mapping of *names* to *object references*. The `console_scripts` group is used to define command-line, or console, scripts, while the `gui_scripts` group defines GUI scripts. We've defined a console script called `train-schedule-cli`, mapped to the `main()` function in the `train_schedule.cli` module; and a GUI script called `train-schedule`, mapped to the `main()` function in the `train_schedule.gui` module.

> The Windows operating system treats console and GUI applications differently. Console applications are launched in a console window and can print to the screen and read keyboard input through the console. GUI applications are launched without a console window. On other operating systems, there is no difference between `console_scripts` and `gui_scripts`.

When pip installs our package, it will generate scripts called `train-schedule` and `train-schedule-cli` and put them in the `bin` folder of our virtual environment (or `Scripts`, if you're using Windows).

> The `console_scripts` and `gui_scripts` entry point group names have special meanings, but you can also use other group names. Pip won't generate scripts for entry points in other groups, but they can be useful for other purposes. Specifically, many projects that support extending their functionality via plugins use particular entry point group names for plugin discovery. This is a more advanced subject that we won't discuss in detail here, but if you are interested you can read about it in the `setuptools` documentation: `https://setuptools.readthedocs.io/en/latest/userguide/entry_point.html`.

That brings us to the end of the preparation phase. The `setuptools` configuration is complete and we have everything we need to start building our distribution packages.

Building and publishing packages

We are going to be using the package builder provided by the *build* project (`https://pypi.org/project/build/`) to build our distribution package. We're also going to need the *twine* (`https://pypi.org/project/twine/`) utility to upload our packages to PyPI. You can install these tools from the `requirements/build.txt` file provided with the source code of this chapter. We recommend installing these into a new virtual environment.

> Because project names on PyPI have to be unique, you won't be able to upload the `train-schedule` project without changing the name first. You should change the `name` in the `setup.cfg` file to something unique before building distribution packages. Bear in mind that this means that the filenames of your distribution packages will also be different from ours.

Building

The `build` project provides a simple script for building packages according to the PEP 517 specification. It will take care of all the details of building distribution packages for us. When we run it, `build` will do the following:

1. Create a virtual environment.
2. Install the build requirements listed in the `pyproject.toml` file into the virtual environment.
3. Import the build backend specified in the `pyproject.toml` file and run it to build a source distribution.
4. Create another virtual environment and install the build requirements.
5. Import the build backend and use it to build a wheel from the source distribution built in *step 3*.

Let's see it in action. Enter the `train-project` folder in the chapter's source code, and run the following command:

```
$ python -m build
* Creating venv isolated environment...
* Installing packages in isolated environment...
 (setuptools>=51.0.0, wheel)
* Getting dependencies for sdist...
...
* Building sdist...
...
* Building wheel from sdist
* Creating venv isolated environment...
* Installing packages in isolated environment...
 (setuptools>=51.0.0, wheel)
* Getting dependencies for wheel...
...
* Installing packages in isolated environment... (wheel)
* Building wheel...
...
Successfully built train-schedule-1.0.0.tar.gz and
 train_schedule-1.0.0-py3-none-any.whl
```

We've removed a lot of lines from the output to make it easier to see how it follows the steps we listed above. If you look at the content of your `train-project` folder, you'll notice there is a new folder called `dist` with two files: `train-schedule-1.0.0.tar.gz` is our source distribution, and `train_schedule-1.0.0-py3-none-any.whl` is the wheel.

Before uploading your package, it's a good idea to do a couple of checks to make sure it built correctly. First, we can use `twine` to verify that the `long_description` will render correctly on the PyPI website:

```
$ twine check dist/*
Checking dist/train_schedule-1.0.0-py3-none-any.whl: PASSED
Checking dist/train-schedule-1.0.0.tar.gz: PASSED
```

If `twine` reports any problems, you should fix them and rebuild the package. In our case the checks passed, so let's install our wheel and make sure it works. In a separate virtual environment, run:

```
$ pip install dist/train_schedule-1.0.0-py3-none-any.whl
```

With the wheel installed in your virtual environment, try to run the application, preferably from outside the project directory. If you encounter any errors during installation or when running your code, check your setup configuration carefully for typos. `setuptools` will ignore any sections or options it doesn't recognize, so misspelled names can result in broken wheels. Also make sure that any data files your project relies on are correctly listed in your `MANIFEST.in` file.

Our package seems to have built successfully, so let's move on to publishing it.

Publishing

Since this is only an example project, we'll upload it to TestPyPI instead of the real PyPI. This is a separate instance of the package index that was created specifically to allow developers to test package uploads and experiment with packaging tools and processes.

Before you can upload packages, you will need to register an account. You can do this now, by going to the TestPyPI website at `https://test.pypi.org` and clicking on **Register**. Once you've completed the registration process and verified your email address, you will need to generate an API token. You can do so on the **Account Settings** page of the TestPyPI website. Make sure you copy the token and save it before closing the page. You should save your token to a file named `.pypirc` in your user home directory. The file should look like this:

```
[testpypi]
  username = __token__
  password = pypi-...
```

The `password` value should of course be replaced with your actual token.

> We strongly recommend that you enable two-factor authentication for both your TestPyPI account and especially your real PyPI account.

Now you are ready to run `twine` to upload your distribution packages:

```
$ twine upload --repository testpypi dist/*
Uploading distributions to https://test.pypi.org/legacy/
Uploading train_schedule-1.0.0-py3-none-any.whl
100%|████████████████████████| 28.0k/28.0k [00:01<00:00, 19.8kB/s]
Uploading train-schedule-1.0.0.tar.gz
100%|████████████████████████| 23.0k/23.0k [00:01<00:00, 23.6kB/s]

View at:
https://test.pypi.org/project/train-schedule/1.0.0/
```

`twine` displays handy progress bars to show how the uploads are progressing. Once the upload is complete, it prints out a URL where you can see details of your package. Open it in your browser and you should see our project description with the contents of our `README.md` and `CHANGELOG.md` files. On the left of the page, you'll see links for the project URLs, the author details, license information, keywords, and classifiers.
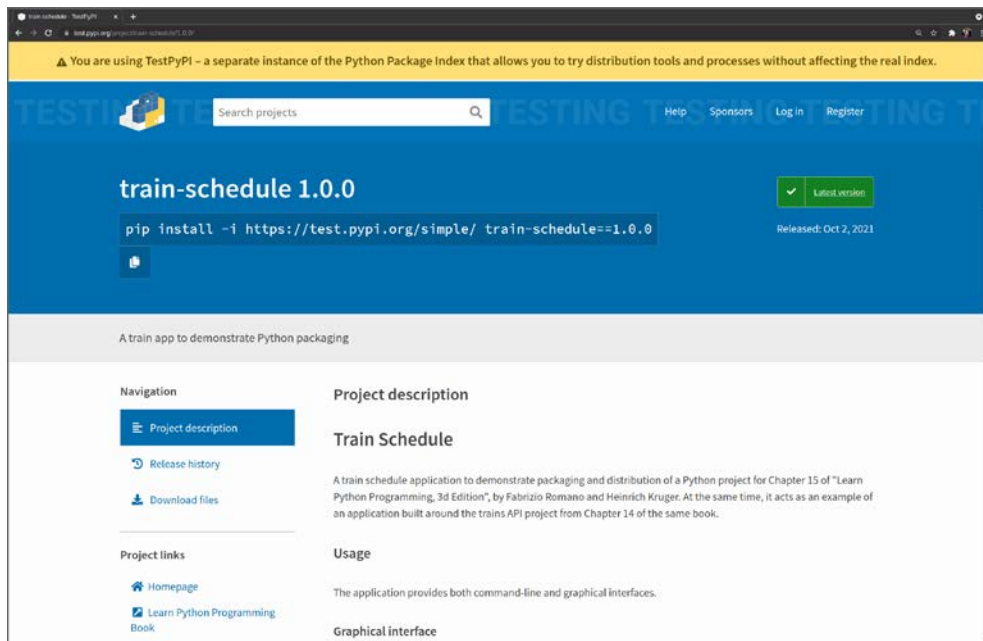
Figure 15.5: Our project page on the TestPyPI website

Figure 15.5 shows what this page looks like for our train-schedule project. You should check all the information on the page carefully and make sure it matches what you expect to see. If not, you will have to fix the metadata in your setup.cfg, rebuild, and re-upload.

> PyPI won't let you re-upload distributions with the same filenames as previously uploaded packages. To fix your metadata, you will have to increment the version of your package. For this reason, it is a good idea to use developmental release numbers until you are 100% sure everything is correct.

Now we can install our package from the TestPyPI repository. Run the following in a new virtual environment:

```
pip install --index-url https://test.pypi.org/simple/ \
    --extra-index-url https://pypi.org/simple/ train-schedule==1.0.0
```

The --index-url option tells pip to use https://test.pypi.org/simple/ as the main package index. We use --extra-index-url https://pypi.org/simple/ to tell pip to also look up packages in the standard PyPI index, so that dependencies that aren't available in the TestPyPI index can be installed. The package installs successfully, which confirms that our package was built and uploaded correctly.

If this were a real project, we would now proceed to uploading to the real PyPI. The process is the same as for TestPyPI. When you save your PyPI API key, you should add it to your existing .pypirc file under the heading [pypi], like this:

```
[pypi]
  username = __token__
  password = pypi-…
```

You also don't need to use the --repository option to upload your package to the real PyPI; you can just run the following:

```
$ twine upload dist/*
```

As you can see, it's not very difficult to package and publish a project, but there are quite a few details to take care of. The good news is that most of the work only needs to be done once, when you publish your first release. For subsequent releases, you usually just need to update the version and maybe adjust your dependencies. In the next section, we'll give you some advice that should make the process easier.

Advice for starting new projects

It can be a tedious process to do all the preparation work for packaging in one go. It's also easy to make a mistake like forgetting to list a required dependency if you try to write all your setup config just before publishing your package for the first time. It's much easier to start with very simple `pyproject.toml` and `setup.cfg` files, containing only the essential configuration and metadata. You can then add to your metadata and configuration as you work on your project. For example, every time you start using a new third-party project in your code, you can immediately add it to your `install_requires` list. It's also a good idea to start writing your README file early on and expanding it as you progress. You may even find that writing a paragraph or two describing your project helps you to think more clearly about what it is you are trying to achieve.

To help you, we have created what we think is a good initial skeleton for a new project. You can find it in the `skeleton-project` folder in this chapter's source code:

```
$ tree skeleton-project/
skeleton-project/
├── example
│   └── __init__.py
├── tests
│   └── __init__.py
├── README.md
├── pyproject.toml
├── setup.cfg
└── setup.py
```

Feel free to copy this, modify it as you wish, and use it as a starting point for your own projects.

Alternative tools

Before we finish the chapter, let's have a quick look at some alternative options that you have for packaging your projects. Before PEP 517 and PEP 518, it was very difficult to use anything other than `setuptools` to build packages. There was no way for projects to specify what libraries were required to build them or how they should be built, so pip and other tools just assumed that packages should be built using `setuptools`.

Thanks to the build-system information in the `pyproject.toml` file, it is now easy to use any packaging library you want. There aren't that many alternatives yet, but there are a few that are worth mentioning:

- The Flit project (`https://flit.readthedocs.io/en/latest/index.html`) was instrumental in inspiring the development of the PEP 517 and PEP 518 standards (the creator of Flit was a co-author of PEP 517). Flit aims to make packaging simple, pure Python projects that don't require complex build steps (like compiling C code) as easy as possible. Flit also provides a command-line interface for building packages and uploading them to PyPI (so you don't need the `build` tool or `twine`).

- Poetry (`https://python-poetry.org/`) also provides both a command-line interface for building and publishing packages, and a lightweight PEP 517 build backend. Where Poetry really shines, though, is in its advanced dependency management features. Poetry can even manage virtual environments for you.

- Enscons (`https://github.com/dholth/enscons`) is rather different from the other tools we've seen, in that it is based on the SCons (`https://scons.org/`) general-purpose build system. This means that unlike Flit or Poetry, Enscons can be used to build distributions that include C language extensions.

The tools we've discussed in this chapter are all focused on distributing packages through PyPI. Depending on your target audience, this might not always be the best choice, though. PyPI exists mainly for distributing projects such as libraries and development tools for use by Python developers. Installing and using packages from PyPI also requires having a working Python installation and at least enough Python knowledge to know how to install packages with pip.

If your project is an application aimed at a less technically adept audience, you may want to consider other options. The Python Packaging User Guide has a useful overview of various options for distributing applications, at `https://packaging.python.org/overview/#packaging-applications`.

Further reading

That brings us to the end of our packaging journey. We will finish this chapter with a few links to resources where you can read more about packaging:

- The Python Packaging Authority's Packaging History page (`https://www.pypa.io/en/latest/history/`) is a useful resource for understanding the evolution of Python packaging.

- The Python Packaging User Guide (`https://packaging.python.org/`) has useful tutorials and guides, as well as a packaging glossary, links to packaging specifications, and a summary of various interesting projects related to packaging.

- The `setuptools` documentation (`https://setuptools.readthedocs.io/`) contains a lot of useful information.

As you read these (and other packaging resources), it is worth bearing in mind that PEP 517 and PEP 518 were only finalized a few years ago and much of the available documentation still refers to older ways of doing things.

Summary

In this chapter, we looked at how to package and distribute Python projects through the Python Package Index. We started with some theory about packaging and introduced the concepts of projects, releases, and distributions on PyPI.

We talked about `setuptools`, the most widely used packaging library for Python, and worked through the process of preparing a project for packaging with `setuptools`. In the process, we saw various files that need to be added to a project to package it and what each of them is for. We discussed the metadata that you should provide to describe your project and help users find it on PyPI, as well as how to add our code and data files to our distribution, how to specify our dependencies, and how to define entry points so that pip will automatically generate scripts for us. We also looked at the tools that Python gives us to query the distribution metadata from our code and how to access packaged data resources in our code.

We moved on to talk about how to build distribution packages and how to use `twine` to upload those packages to PyPI. We also gave you some advice on starting new projects. We concluded our tour of packaging by briefly talking about some alternatives to `setuptools` and pointing you to some resources where you can learn more about packaging.

We really do encourage you to start distributing your code on PyPI. No matter how trivial you think it might be, someone else somewhere in the world will probably find it useful. It really does feel good to contribute and give back to the community, and besides, it looks good on your CV.

`packt.com`

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals

- Learn better with Skill Plans built especially for you

- Get a free eBook or video every month

- Fully searchable for easy access to vital information

- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.Packt.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `customercare@packtpub.com` for more details.

At `www.Packt.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:
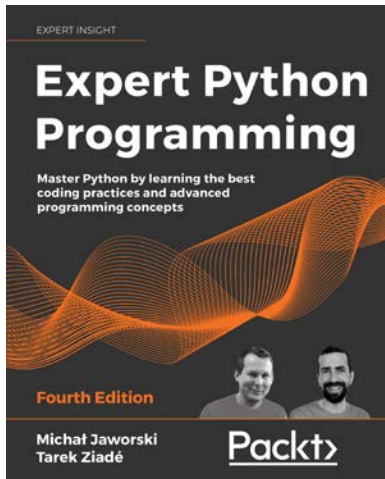
Expert Python Programming – Fourth Edition

Michał Jaworski

Tarek Ziadé

ISBN: 978-1-80107-110-9

- Explore modern ways of setting up repeatable and consistent Python development environments

- Effectively package Python code for community and production use

- Learn modern syntax elements of Python programming, such as f-strings, enums, and lambda functions

- Demystify metaprogramming in Python with metaclasses

- Write concurrent code in Python

- Extend and integrate Python with code written in C and C++

Python Object-Oriented Programming – Fourth Edition

Steven F. Lott

Dusty Phillips

ISBN: 978-1-80107-726-2

- Implement objects in Python by creating classes and defining methods

- Extend class functionality using inheritance

- Use exceptions to handle unusual situations cleanly

- Understand when to use object-oriented features, and more importantly, when not to use them

- Discover several widely used design patterns and how they are implemented in Python

- Uncover the simplicity of unit and integration testing and understand why they are so important

- Learn to statically type check your dynamic code

- Understand concurrency with asyncio and how it speeds up programs

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished Learn Python Programming, Third edition, we'd love to hear your thoughts! If you purchased the book from Amazon, please `click here to go straight to the Amazon review page` for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

code duplication, reducing 117
complex task, splitting 117
example 153, 154
guidelines 144, 145
implementation details, hiding 118
readability, improving 119
recursive functions 145
return values 141-143
traceability, improving 120
function signatures 139
functools.update_wrapper function
reference link 200

G

generator-based context managers 253-255
generator expressions 171, 178-181
name localization 188, 189
generator functions 171-174
generator.__next__() method 174, 175
generators 170
behavior 190
overdo, avoiding 184-188
generator.send() method 176, 177
GET method 427
getters 229, 231
Git 33
global scope 27, 121
global statement 122
granularity 334
graphical user interface (GUI) 365
FAQ, reference link 20
gray-box testing 316
greatest common divisor (GCD) 185
grid 377
GUI application 375-377
business logic 382
imports 378
improving 389, 390
layout logic 378-382

H

handlers 351
Has-A type of relationship 210, 215
hash 296
hashability 59

hash-based message authentication
code 300
hashing 298
Hashlib 296
Haskell
URL 164
Heinrich 36
help built-in function 150
HMAC algorithm 300
Httpie utility 449
reference link 450
HTTP requests 279
performing 280-282
HTTP status codes
reference link 428
Hypertext Transfer Protocol (HTTP) 426, 427
reference link 428

I

IDEs 35, 36
if statement 84
Img Frame 377
immutable object 4, 39
immutable sequences 48
bytes 48
strings 48, 49
tuples 52, 53
importlib-metadata
reference link 495
impressions 417
indexing 80
infinite iterators 111
infinite loop 97
inheritance 210-214
initializer 32, 209
in-memory stream
using 278, 279
input/output (I/O) 278
input parameters 124
combining 137, 138
insertion sort 57
inside-out technique 90
instance
initializing 209, 210
instance attributes 206
integer division 41