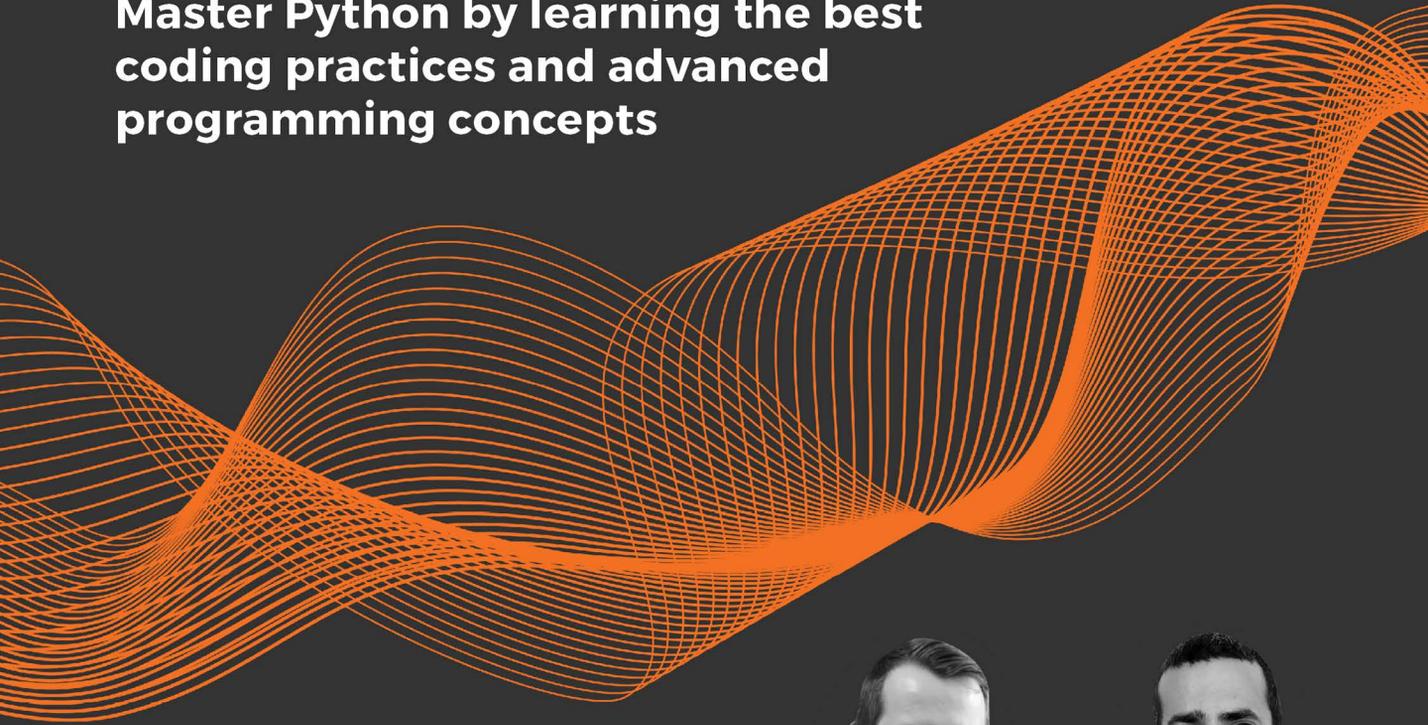# Expert Python Programming

Master Python by learning the best coding practices and advanced programming concepts

Fourth Edition

Michał Jaworski
Tarek Ziadé

Packt>

Expert Python Programming

Fourth Edition

Master Python by learning the best coding practices
and advanced programming concepts

Michał Jaworski

Tarek Ziadé

Expert Python Programming
Fourth Edition

Contributors

About the authors

Michał Jaworski has more than 10 years of professional experience in writing software using various programming languages. Michał has spent most of his career writing high-performance and distributed backend services for web applications. He has served in various roles at multiple companies: from an ordinary software engineer to lead software architect. His beloved language of choice has always been Python.

Tarek Ziadé is a software engineer, located in Burgundy, France. He works at Elastic, building tools for developers. Before Elastic, he worked at Mozilla for 10 years, and he founded a French Python User group, called *AFPy*. Tarek has also written several articles about Python for various magazines, and a few books in French and English.

About the reviewer

Tal Einat has been developing software for nearly 20 years, of which Python has always been a major element. He's been a core developer of the Python language since 2010. Tal holds a B.Sc. in Math & Physics from Tel Aviv University. He likes hiking, computer games, philosophical sci-fi, and spending time with his family.

For the past eight years, Tal has been developing educational technology, first at *Compedia* where he built a group developing VR and AR education apps, and later at the startup *FullProof* of which he was a co-founder.

Tal currently works at *Rhino Health*, a startup working to the enable development and use of medical AI models with patient data from across the globe while preserving patient privacy.

> *I dedicate my part of the work on this book to my grandfather Jacob "Yanek" Friedman who passed away recently — you live on in our thoughts, making us strong and putting a smile on our faces. I'd like to thank my wife, children, brother, parents and extended family, for being ever supportive and filling my life with so much joy.*

Table of Contents

Preface

Python rocks!

From the earliest version in the late 1980s to the current 3.9 version, Python has evolved with the same philosophy: providing a multi-paradigm programming language with readability and productivity in mind.

Initially, people used to see Python as yet another scripting language. Many of them didn't believe it could be used to build large and complex systems. But over the years, and thanks to some pioneer companies, it became obvious that Python could be used to build almost any kind of a software.

Although writing Python code is easy, making it readable, reusable, and easy to maintain is challenging. You can achieve those qualities only through good software artistry and technique, which you will build gradually by constantly learning and gaining experience.

This book was written to express many years of professional experience in building all kinds of applications with Python, from small system scripts done in a couple of hours to very large applications written by dozens of developers over several years.

This book is divided into three parts:

1. **Knowing your tools**: *Chapters 1* to *4* focus on basic elements of Python programmer's toolbelt. From productivity tools, through modern environments, to the newest syntax elements introduced in the latest Python releases. It also offers a safe landing zone for programmers who have experience with other programming languages and are just starting to learn more advanced Python.

2. **Building applications with Python**: *Chapters 5* to *9* are all about design patterns, programming paradigms, and metaprogramming techniques. We will try to build some small but useful programs and will be often taking a deeper look into application architecture. We will also go a bit beyond Python and see how we can integrate code written using other programming languages.

3. **Maintaining Python applications**: *Chapters 10* to *13* will be discussing all the things that usually happen after the application "goes live". We will showcase tools and techniques that help to keep applications easily maintainable and show how to approach common problems with packaging, deployment, monitoring, and performance optimization.

Who this book is for

The Python programming book is intended for expert programmers who want to learn about Python's advanced-level concepts and latest features.

This book is written for Python developers who wish to go further in mastering Python. And by developers, I mean mostly professional programmers who write Python software for a living. This is because it focuses mostly on tools and practices that are crucial for creating performant, reliable, and maintainable software in Python.

However, this does not mean that hobbyists won't find anything interesting. This book is great for anyone who is interested in learning advanced-level concepts with Python. Anyone who has basic Python skills should be able to follow the content of the book, although it might require some additional effort from less experienced programmers. It should also be a good introduction to the newest releases of Python for those who are still a bit behind and continue to use older versions of Python.

What this book covers

Chapter 1, *Current Status of Python*, showcases the current state of the Python language and its community. We will see how Python is constantly changing and why it is changing. We will learn what to do with old Python 2 code and how to be constantly up to date with what is currently happening in the Python community.

Chapter 2, *Modern Python Development Environments*, describes modern ways of setting up repeatable and consistent development environments for Python programmers. We will learn differences between application-level and system-level isolation. We will concentrate on two popular tools for environment isolation, virtualenv-type environments and Docker containers, but will also review other alternatives. At the end of the chapter, we will discuss common productivity tools that are extremely useful during development.

Chapter 3, *New Things in Python*, showcases recent Python language additions. We will review the most important Python syntax changes that happened in the last four releases of Python. We will also take a look at exciting changes that are scheduled for the next major Python release—Python 3.10.

Chapter 4, *Python in Comparison with Other Languages*, shows how Python compares to other languages. We will learn what programming idioms are and how to recognize them in code. We will take a deeper look into key elements of Python's object-oriented programming model and how it is different from other object-oriented programming languages but will also discuss other popular programming language features like descriptors, decorators, and dataclasses. This chapter should allow programmers with experience in other languages to safely land in the Python ecosystem.

Chapter 5, *Interfaces, Patterns, and Modularity*, discusses elements of Python that allow for implementing various reusable design patterns. If focuses on the concept of class interfaces and how they can be implemented in Python. It also discusses inversion of control and dependency injection—two extremely useful but not necessarily popular programming techniques.

Chapter 6, *Concurrency*, explains how to implement concurrency in Python using different approaches and libraries. It features three major concurrency models: multithreading, multiprocessing and asynchronous programming. In this chapter we will learn key differences between those models and how to use them effectively.

Chapter 7, *Event-Driven Programming*, describes what event-driven programming is and how it relates to asynchronous programming and different concurrency models. We will present various approaches to event-driven programming along with useful libraries.

Chapter 8, *Elements of Metaprogramming*, presents an overview of common approaches to metaprogramming available to Python programmers. We will learn about common metaprogramming techniques like decorators, as well as metaclasses and code generation patterns.

Chapter 9, *Bridging Python with C and C++*, explains how to integrate code written in different languages in your Python application. We will learn when extensions in C can be useful and how to create them.

Chapter 10, *Testing and Quality Automation*, is about providing automated testing and quality processes. We will learn about a popular testing framework—`Pytest`—and many useful testing techniques. We will also cover tools that can be used to assess code quality metrics and improve code style in fully automated way.

Chapter 11, *Packaging and Distributing Python Code*, describes the current state of Python packaging and best practices for creating packages that are to be distributed as open source code in the **Python Package Index** (**PyPI**). We will also cover the topics of packaging applications for web development and creating standalone Python executables for desktop applications.

Chapter 12, *Observing Application Behavior and Performance*, discusses the topic of application observability. We will learn about Python logging systems, how to monitor application metrics and perform distributed transaction tracing. We will also learn how to scale simple observability practices to large-scale distributed systems.

Chapter 13, *Code Optimization*, discusses the basic rules of optimization that every developer should be aware of. We will learn how to identify application performance bottlenecks and how to use common profiling tools. We will also learn common optimization techniques and strategies that can be easily applied in many situations once you know where the bottleneck is.

To get the most out of this book

This book is written for developers who work under any operating system for which Python 3 is available.

This is not a book for beginners, so I assume you have Python installed in your environment or know how to install it. Anyway, this book takes into account the fact that not everyone needs to be fully aware of the latest Python features or officially recommended tools. This is why *Chapter 2*, *Modern Python Development Environments* provides an overview of recommended techniques and tools (such as virtual environments and `pip`) for setting up development environments.

Download the example code files

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition`. In case there's an update to the code, it will be updated on the existing GitHub repository.

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/9781801071109_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, and user input. Here is an example: "Any attempt to run the code that has such issues will immediately cause the interpreter to fail, raising a `SyntaxError exception`."

A block of code is set as follows:

```python
print("hello world")
```

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

```
$ python3 script.py
```

Some code examples will be representing input of shells. You can recognize them by specific prompt characters:

- `>>>` for interactive Python shell
- `$` for Bash shell (macOS and Linux)
- `>` for CMD or PowerShell (Windows)

Some code or command-line examples will require providing your own name or values in provided placeholders. Placeholders will be surrounded with `<>` characters as in following example:

```
$ python <my-module-name>
```

> 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: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@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 `www.packt.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@packt.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 `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1

Current Status of Python

Python is amazing. It is amazing because it has managed to stay relevant and keep growing for so many years.

For a very long time, one of the most important virtues of Python was interoperability. Thanks to that feature it didn't matter what operating system you or your customers were using. If the Python interpreter was available for a given operating system, your software written in Python would work there. And, most importantly, your software would always work the same way regardless of that operating system. However, this is now a common trait. Many modern programming languages provide similar interoperability capabilities. Also, with the advent of cloud computing, web-based applications, and reliable virtualization software, it isn't that important to have a programming language that works in many operating systems.

What seems to be increasingly more important for programmers nowadays is productivity. With a constant search for innovation, it is often important to build something that you can test in the field with real users and then iterate quickly from that point until you build a really valuable product. And Python allows programmers to iterate really fast. The Official Python Package Index is a tremendous library of software libraries and frameworks that can be easily reused in your software. It greatly reduces the amount of time and effort required to get your job done. This vast availability of community libraries together with clear and concise syntax that favors readability greatly limits the time and effort that has to be spent on creating and maintaining the software. That makes Python really shine in the area of programming productivity.

Python stayed relevant for so long because it was constantly evolving. And it still keeps evolving. That's why in this chapter we will take a brief look at the current status of Python and learn how to stay up to date with changes happening in the Python ecosystem and community.

In this chapter, we will cover the following topics:

- Where are we now and where are we going?
- What to do with Python 2
- Keeping up to date

Let's start by considering the history and development of Python, and where we're currently at.

Where are we now and where are we going?

Python isn't a young language. Its history starts somewhere in the late 1980s and the official 1.0 release happened in year 1994. We could discuss the whole timeline of major Python releases mentioned here, but we really only need to be concerned about a few dates:

- October 16, 2000: official release of Python 2.0
- December 3, 2008: official release of Python 3.0
- November 9, 2011: announcement of Python 2.8 release un-schedule
- January 1, 2020: official sunsetting of Python 2

So, at the time of writing, Python 3 is almost half as old as Python itself. It has also been active for longer than Python 2 was when it comes to active development of new language features.

Although Python 3 has been around for quite some time, its adoption was so slow that the initial end of life for Python 2 was postponed by 5 years. It was mostly due to a lot of backward compatibility issues that didn't always allow an easy and straightforward transition. A big part of Python's greatness comes from the vast number of freely available libraries. So, it was even harder to make the Python 3 transition if your software critically relied on a third-party Python package that wasn't compatible with Python 3 yet.

Fortunately, Python 2 is finally behind us and the Python community can finally breathe a sigh of relief. Many developers of open-source Python libraries stopped providing compatibility with Python 2 already a long time ago. Also, the official sunsetting of Python 2 provided an incentive to strategically prioritize transitions to Python 3 in corporate environments, where there is usually limited desire to do so. That incentive can be attributed mainly to the fact that there is absolutely no chance of security patches for Python 2 being delivered anymore.

What to do with Python 2

We know that Python 2 is no longer officially supported by language developers. Due to the lack of security patches, it should be considered unsafe. But is it dead yet?

Probably not. Even relatively popular open-source libraries can lose the interest of their authors and contributors over time. It also happens that some better alternatives appear, and there's simply no point in developing the original library anymore. Such libraries are often left unmaintained, so no one could update them for compatibility with Python 3.

One of the important reasons why Python 3 couldn't take off for a long time was the sluggish release processes of many Linux distributions. Distribution maintainers often aren't inclined toward the fast adoption of new language versions, especially if these versions break backward compatibility and require patches of other software. Many Python programmers are Linux users, and if they don't have access to the latest version of the language through the system package repository they are less likely to upgrade it on their own.

But the place where Python 2 will definitely linger for a few more years is corporate environments. When money is at stake, it is usually hard to convince the stakeholders that something that's already been done needs to be revisited just for the sake of being up to date with upstream changes, especially if the software works.

Python 2 code probably won't be that common for the core components of actively developed software, but it can still be found quite often in utility scripts, internal tools, or services that haven't seen active development for a long time.

If you're in a situation where you still need to maintain bits of software written in Python 2 you would be better off migrating to Python 3 soon. This often can't be done overnight, and sometimes you'll have to convince others first before you're ready to proceed.

If you're in such a situation, consider the following strategy:

1. **Identify what you need to migrate and why**: Depending on the situation, you'll be dealing with various pieces of code that have different uses. Not everything may need migration after all. If code isn't actually used by anyone, there is no value in keeping it updated.

2. **Identify what is holding you back**: Sometimes you'll be working with code that has dependencies that cannot be easily migrated. This will make the process a bit harder, so you'll have to know this in advance in order to create a good migration plan.

3. **Convince the stakeholders**: If you're developing an open-source library, you'll have to convince fellow contributors that will help you make this migration; if you're working at a software company, you'll have to convince the stakeholders who are paying for the job to be done that this is more important than, for instance, working on a new feature.

Usually the hardest thing to do is to get buy-in from stakeholders, especially if you're writing code professionally and need to find a way to squeeze such a project into the day-to-day development of new features. You'll need to be well prepared before raising such an issue. That's why the first two steps are essential. They allow you to estimate the amount of work required to make the change and construct a convincing reason for doing so. After all, the best way to convince someone is by presenting the list of benefits of doing a migration. The following are common positive reasons for doing a migration:

- **Ability to use newer and better libraries**: After the official sunsetting of Python 2 there is a very low chance that new (and possibly better) libraries will be compatible with Python 2.

- **Reduced cost of ownership**: If the team is using the latest version of Python in other projects/components, it will be cheaper to converge multiple projects to a single version as this will reduce the overall complexity.

- **Simpler maintenance and operations**: Different runtime environments and operating systems will gradually lose default built-in support for Python 2. Using a newer version of Python will limit the operational costs required to deploy software.

- **Easier new hire onboarding**: Sticking to a single version of Python makes it easier to onboard new team members as it will be easier for them to work with the whole codebase from the very beginning

Another tool for convincing stakeholders is explaining the risks related to not migrating to a newer version of Python:

- **Big security risk**: After the official end of life, there's no guarantee of any official security fixes. This risk is, of course, only speculative until real vulnerabilities are discovered. Still, by using Python 2 you're already limiting your ability to accept updates to third-party libraries and many open-source projects that dropped support for Python 2 a few years ago.

- **High security-related cost**: Although it is possible to fork open-source code and do security fixes on your own, the cost of doing this continuously is usually much bigger than the overall cost of migrating to a newer version of the language.

- **Troubles in hiring new people**: This is problematic regardless of target programmer seniority. Younger developers may not be as familiar with Python 2 as they are with Python 3. This will make their onboarding harder and also increase the risk of making rookie mistakes with potentially disastrous outcomes. Senior engineers on the other hand may not want to work in an environment that relies on outdated technology.

Regardless of your communication strategy, you'll have to create a reasonable tactical plan that carefully balances the promise of positive outcomes, risks, and the ability to reach team goals at a sustainable pace. For instance, investing in reducing future maintenance costs usually pays for itself only after a longer period of time. That's why it makes sense to also spread such investment over time. Known and exploitable security vulnerabilities, on the other hand, should always be prioritized.

Keeping up to date

Technology evolves constantly. People are constantly looking for tools that will allow them to solve their problems more easily than before. Every few months, either a completely new language pops up almost out of nowhere, or one of the well-established languages introduces a completely new syntax feature. This also happens to Python. We will briefly consider a few of the most important recent additions to the Python language in *Chapter 3*, *New Things in Python*.

New languages or language features drive the creation of novel libraries and frameworks. These in turn open the way for new programming paradigms and new design patterns. The success of one such pattern or paradigm in one language ecosystem often encourages programmers to adopt them in other languages. That's how new ideas spread from language to language.

We see that happening to Python too. We will be able to see how seemingly different programming languages share similar features or concepts in *Chapter 4*, *Python in Comparison with Other Languages*.

This process of language evolution is constant, inevitable, and seems to be only speeding up. Starting from Python 3.9 (released October 5, 2020), new major releases of Python will be happening annually. These releases more often than not lead to explosions of new libraries and frameworks that are trying to test and explore new tools. It's great for the Python community because it gives it constant fuel to innovate. But it can be overwhelming for anyone that wants to keep an eye on what's happening in Python. And being up to date is extremely important for every Python professional.

In the following few sections, we will discuss a few ways in which you can source information on what is happening in Python. It will allow you to better anticipate changes happening to the language and the community around it. This will keep you up to date with the latest best practices, and new tools that are worth investing in.

PEP documents

The Python community has a well-established way of dealing with changes. While speculative Python language ideas are mostly discussed on a specific mailing list (`python-ideas@python.org`), nothing major ever gets changed without the existence of a new document, called a **Python Enhancement Proposal** (**PEP**).

> You can subscribe to `python-ideas@python.org` and other mailing lists at `https://mail.python.org/mailman3/lists/`.

A PEP is a formalized document that describes, in detail, the proposal of change to be made in Python. It is also the starting point for the community discussion. A few of the topics covered in this chapter were in fact already extensively described in their dedicated PEP documents:

- PEP 373 — Python 2.7 Release Schedule
- PEP 404 — Python 2.8 Un-release Schedule
- PEP 602 — Annual Release Cycle for Python

The whole purpose, format, and workflow of these documents is also standardized in the form of a standalone PEP—the PEP 1 document.

PEP documentation is very important for Python, and, depending on the topic, it serves different purposes:

- **Informing**: They summarize the information needed by core Python developers, and notify about Python release schedules
- **Standardizing**: They provide code style, documentation, or other guidelines
- **Designing**: They describe the proposed features

A list of all proposed PEPs is available in a living (by which we mean *continuously updated*) PEP 0 document. It is a great source of information for those who are wondering what direction Python is heading in but do not have time to track every discussion on Python mailing lists. It shows which ideas have been accepted, which ideas have been implemented, and also which ideas are still under consideration.

> Online URLs for PEP documents take the following form:
>
> `http://www.python.org/dev/peps/pep-XXXX`
>
> XXXX is a 4-digit PEP number with leading zeros. It should also be easy to find them through web search engines using the `Python PEP XXX` search term.
>
> The PEP 0 document is available at `https://www.python.org/dev/peps/`.

The index of all officially discussed proposals serves an additional purpose. Very often, people ask questions like the following:

- Why does feature A work that way?
- Why does Python not have feature B?

In most such cases, the answer is already available in the specific PEP where the feature was discussed. Instead of having the same conversation about adding some language feature over and over again, people can be redirected to a specific PEP. And there are a lot of PEPs describing Python language features that have been proposed but not accepted.

When it comes to being up to date, the most important section of the PEP index is definitely open PEPs, which describes ideas that are still under active consideration. For instance, the following are selected interesting PEPs that were still open at the time of writing:

- PEP 603—Adding a frozenmap type to collections
- PEP 634—Structural Pattern Matching: Specification
- PEP 638—Syntactic Macros
- PEP 640—Unused variable syntax

These proposals vary from relatively small extensions of the existing standard library (such as PEP 603) to completely new and complex syntax features (such as PEP 638). If you wonder what the Python community is considering including in future Python releases, then open PEPs are the best source of such information.

Active communities

The non-profit organization behind Python is **Python Software Foundation** (**PSF**). It holds intellectual property rights for Python and manages its licensing. A brief excerpt from its mission statement is as follows:

> *The mission of the Python Software Foundation is to promote, protect, and advance the Python programming language, and to support and facilitate the growth of a diverse and international community of Python programmers.*

> You can find the full PSF mission statement text at `https://www.python.org/psf/mission/`.

Support for the community of Python programmers is important to the PSF mission because it is, after all, the community that drives the development of Python. The community does that by extending the language transparently and openly (as explained in the *PEP documents* section), but also through expanding and maintaining a rich ecosystem of third-party packages and frameworks. So, one of the best ways to know what's happening in Python is to get in touch with its community.

Like any other programming language, there are a lot of independent online communities dedicated to Python. They are usually focused on specific frameworks or software development areas, like web development, data science, machine learning, and so on.

One could assume that there is at least a single place online where all important discussions about the core of the language and its interpreter happen. Unfortunately, it's not that simple.

Due to many reasons, some of which are historic, the landscape of official Python mailing lists and discussion boards can be very confusing. There are numerous official and semi-official mailing lists and discussion boards where Python developers hang out. It is especially confusing when it comes to mailing lists because official mailing lists are spread over two different mailing list managers and thus have two separate sets of list archives. Those archives are:

- **Mailman 2**: This is an older and smaller archive available at `https://mail.python.org/mailman/listinfo`. Historically all `python.org` mailing list archives could be accessed through the Mailman 2 archive but most of them have been migrated to the Mailman 3 archive. Anyway, there are still some mailing lists with active discussions that are managed through Mailman 2.

- **Mailman 3**: This is a younger archive available at `https://mail.python.org/archives`. It is currently a primary archive of `python.org` mailing lists and is the home for more active mailing lists. It has a more modern and convenient interface, but it doesn't include mailing lists that have not been migrated from Mailman 2 yet.

When it comes to actual mailing lists, there's plenty of them, but sadly the majority of them aren't active anymore. Some mailing lists are dedicated to specific projects (like `scikit-image@python.org`) or focus on specific areas of interest of their members (like `code-quality@python.org`). Besides lists with very specific themes, there are few general mailing lists that should be very interesting for every Python programmer. These are:

- `python-ideas@python.org`: This is a staple when it comes to Python mailing lists. It is a place for discussing a wide variety of ideas regarding Python. Most PEPs start as a speculative idea discussion on this mailing list. It is a great place for discussing potential "what ifs" and finding out what features people would like to see in the near future.

- `python-dev@python.org`: This is a mailing list specifically for the purpose of core Python development (mainly CPython interpreter). It is also a place where first drafts of new PEPs are discussed before being officially announced through other channels. It isn't a place where you should be asking for general Python help, but it is an essential resource if you would like to try your hand at fixing known bugs in the CPython interpreter or the Python standard library.

- `python-announce-list@python.org`: This is a mailing list for various announcements. You can find announcements of conferences and meetups here, as well as notifications about new releases of your favorite packages and frameworks or new PEPs. It is also great for discovering new and exciting projects.

Besides classic mailing lists, there is an official internet forum on the Discourse platform that is available at `https://discuss.python.org/`. It is a relatively new addition to the Python official discussion landscape and its purpose overlaps with many of the pre-existing mailing lists as it has dedicated categories for discussing new ideas, PEPs, and topics of core Python development. It has a lower entry barrier for those unfamiliar with the concept of mailing lists, and a much more modern user experience.

Unfortunately, not every new discussion happens on `discuss.python.org`, so if you want to know about everything that happens at the heart of Python development you will have to keep track of discussion both on the forum and mailing lists. Hopefully, these will eventually converge in a single place.

Besides the official message boards and mailing lists, there are a couple of open Python communities based on popular discussion and messaging platforms. The most notable ones are:

- **PySlackers Slack workspace** (`pyslackers.com`): A large community of Python enthusiasts using the Slack messaging platform to discuss anything Python-related
- **Python Discord server** (`pythondiscord.com`): Another open Python community but using Discord as their messaging platform
- **/r/python subreddit** (`www.reddit.com/r/Python/`): A single subreddit on the Reddit platform dedicated to Python

These three communities are open in the sense that you can freely join them as long as you are a user of their underlying platforms (which are, of course, free). Whatever you choose will probably be dictated by your preference for the specific messaging or discussion platform. The chances are high that you or one of your friends is already using one of these platforms.

The unquestionable advantage of such open communities is that they gather a very large number of members, and you can almost always find someone online that you can talk to. This provides the opportunity for ad hoc and loose discussions about various Python-related topics and allows you to seek quick help in case of simple programming problems.

The downside is that it is impossible to keep track of every discussion happening there. Fortunately, these communities often create systems of separate subchannels or tags that you can subscribe to if you want to be notified about the content of your specific interest. Also, these communities aren't officially endorsed and curated by the PSF. As a result, information found on Reddit or other online communities may sometimes be biased or inaccurate.

Other resources

Reading all the new PEPs, following mailing lists, and joining communities are the best ways to know what is happening at the moment. Unfortunately, doing this consistently requires a lot of time and effort as you will have to filter through huge amounts of information. Also, sources like mailing lists, message boards, and messaging platforms can be full of emotions because discussion is essentially a human interaction. And surprisingly, some tech discussions on contentious topics can be so heated that they are almost indistinguishable from social network drama.

If you are really busy or quickly get overloaded with social interactions between online strangers, there is another way. Instead of filtering online information all by yourself, you can turn to curated content like blogs, newsletters, and so-called "awesome lists."

Newsletters are especially good if you want to be up to date. These are some interesting newsletters that are worth subscribing to:

- **Python Weekly** (`http://www.pythonweekly.com/`) is a popular newsletter that delivers to its subscribers dozens of new, interesting Python packages and resources every week
- **PyCoder's Weekly** (`https://pycoders.com`) is another popular weekly newsletter with a digest of new packages and interesting articles

These newsletters will keep you informed about the most important Python announcements. They will also help you discover new blogs or especially interesting discussions from other discussion platforms, like Reddit or Hacker News. Keep in mind that the content of many general Python newsletters often overlaps, so probably there is no need to subscribe to all of them.

A completely different way to source information is via **awesome lists**. These are curated lists of links to valuable or important resources on specific topics maintained usually as Git repositories on GitHub. These lists are often very long and split into several categories.

These are examples of popular Python-related awesome lists curated by various GitHub users:

- **awesome-python by vinta** (`https://github.com/vinta/awesome-python`): This contains numerous references to interesting projects (mostly hosted on GitHub) and standard library modules divided into over 80 thematic categories. Categories range from basic programming concepts—like caching, authentication, and debugging—to whole engineering areas where Python is often used, like web development, data science, robotics, and penetration testing. The list of projects is supplemented with a collection of links to newsletters, podcasts, books, and tutorials.
- **pycrumbs by kirang89** (`https://github.com/kirang89/pycrumbs`): This is focused on interesting and valuable articles. Articles are divided into over 100 categories dedicated to specific Python features, general programming techniques, and self-development topics.
- **pythonidae by svaksha** (`https://github.com/svaksha/pythonidae`): This is focused on specific fields of science and technology where Python is frequently used, like mathematics, biology, chemistry, web development, physics, image processing, and many more. It has a tree-like structure. The main page contains a list of over 20 main category pages. These categories contain more granular subcategories listing useful libraries and resources.

Awesome lists usually grow over time to enormous sizes. This means they are not a good resource for keeping yourself constantly updated. This is because they are simply snapshots of what the curators considered awesome at the time. Nevertheless, if you need to jump into a completely new area, let's say artificial intelligence, they serve as a good starting point for further research.

Summary

In this chapter, we've discussed the current status of Python and the process of change that is visible throughout the history of that language. We've learned why Python is changing and why it is important to follow that change.

Staying relevant is one of the biggest and most stressful challenges that professional programmers face regardless of the programming language they choose. Due to over 30 years of Python history and the ever-growing Python community, it isn't always clear how to efficiently stay up to date with the changes in the Python ecosystem. That's why we've looked into all the places where you can follow important discussions about Python's future.

What also changes together with the language are development tools that aim to ease and improve the software development processes. In the next chapter, we will continue with the topic of change and take a look at modern development environments. We will learn how to create repeatable and consistent runtime environments both for production and development use. We will also get familiar with various productivity tools provided by the Python community.

2

Modern Python Development Environments

A deep understanding of the programming language of choice is the most important part of being a programming expert. Still, it is really hard to develop good software efficiently without knowing the best tools and practices that are common within the given language community. Python has no single feature that cannot be found in some other language. So, when comparing the syntax, expressiveness, or performance, there will always be a solution that is better in one or more fields. But the area in which Python really stands out from the crowd is the whole ecosystem built around the language. The Python community has spent many years polishing standard practices and libraries that help to create high-quality software in a shorter time.

Writing new software is always an expensive and time-consuming process. However, being able to reuse existing code instead of *reinventing the wheel* greatly reduces development times and costs. For some companies, it is the only reason why their projects are economically feasible. That's why the most important part of the ecosystem is a huge collection of reusable packages that solve a multitude of problems. A tremendous number of these packages are available as open-source through the **Python Package Index (PyPI)**.

Because of the importance of Python's open-source community, Python developers put a lot of effort into creating tools and standards to work with Python packages that have been created by others—starting from virtual isolated environments, improved interactive shells, and debuggers, to utilities that help you to discover, search, and analyze the huge collection of packages that are available on PyPI.

In this chapter, we will cover the following topics:

- Overview of the Python packaging ecosystem
- Isolating the runtime environment
- Using Python's `venv`
- System-level environment isolation
- Popular productivity tools

Before we get into some specific elements of the Python ecosystem, let's begin by considering the technical requirements.

Technical requirements

You can install the free system virtualization tools that are mentioned in this chapter from the following sites:

- Vagrant: `https://www.vagrantup.com`
- Docker: `https://www.docker.com`
- VirtualBox: `https://www.virtualbox.org/`

The following are the Python packages that are mentioned in this chapter that you can download from PyPI:

- `poetry`
- `flask`
- `wait-for-it`
- `watchdog`
- `ipython`
- `ipdb`

Information on how to install packages is included in the *Installing Python packages using pip* section.

The code files for this chapter can be found at `https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition/tree/main/Chapter%202`.

Python's packaging ecosystem

The core of Python's packaging ecosystem is the Python Packaging Index. PyPI is a vast public repository of (mostly) free-to-use Python projects that at the time of writing hosts almost three and a half million distributions of more than 250,000 packages. That's not the biggest number among all package repositories (npm surpassed a million packages in 2019) but it still places Python among the leaders of packaging ecosystems.

Such a large ecosystem of packages doesn't come without a price. Modern applications are often built using multiple packages from PyPI that often have their own dependencies. Those dependencies can also have their own dependencies. In large applications, such dependency chains can go on and on. Add the fact that some packages may require specific versions of other packages and you may quickly run into **dependency hell**—a situation where it is almost impossible to resolve conflicting version requirements manually.

That's why it is crucial to know the tools that can help you work with packages available on PyPI.

Installing Python packages using pip

Nowadays, a lot of operating systems come with Python as a standard component. Most Linux distributions and UNIX-like systems (such as FreeBSD, NetBSD, OpenBSD, and macOS) come with Python either installed by default or available through system package repositories. Many of them even use it as part of some core components—Python powers the installers of Ubuntu (Ubiquity), Red Hat Linux (Anaconda), and Fedora (Anaconda again). Unfortunately, the Python version preinstalled with operating systems is often older than the latest Python release.

Due to Python's popularity as an operating system component, a lot of packages from PyPI are also available as native packages managed by the system's package management tools, such as `apt-get` (Debian, Ubuntu), `rpm` (Red Hat Linux), or `emerge` (Gentoo). It should be remembered, however, that the list of available libraries is often very limited, and they are mostly outdated compared to PyPI. Sometimes they may be evenly distributed with platform-specific patches to make sure that they will properly support other system components.

Due to these facts, when building your own applications, you should always rely on package distributions available on PyPI. The **Python Packaging Authority** (**PyPA**)—a group of maintainers of standard Python packaging tools—recommends `pip` for installing packages. This command-line tool allows you to install packages directly from PyPI. Although it is an independent project, starting from versions 2.7.9 and 3.4 of CPython, every Python release comes with an `ensurepip` module. This simple utility module ensures `pip` installation in your environment, regardless of whether release maintainers decided to bundle `pip`. The `pip` installation can be bootstrapped using the `ensurepip` module as in the following example:

```
$ python3 -m ensurepip

Looking in links: /var/folders/t6/n6lw_s3j4nsd8qhsl1jhgd4w0000gn/T/
tmpouvorgu0
Requirement already satisfied: setuptools in ./.venv/lib/python3.9/
site-packages (49.2.1)
Processing /private/var/folders/t6/n6lw_s3j4nsd8qhsl1jhgd4w0000gn/T/
tmpouvorgu0/pip-20.2.3-py2.py3-none-any.whl
Installing collected packages: pip
Successfully installed pip-20.2.3
```

When you have `pip` available, installing a new package is as simple as this:

```
$ pip install <package-name>
```

So, if you want to install a package named `django`, you simply run:

```
$ pip install django
```

Among other features, `pip` allows specific versions of packages to be installed (using `pip install <package-name>==<version>`) or upgraded to the latest version available (using `pip install --upgrade <package-name>`).

`pip` is not just a package installer. Besides the `install` command, it offers additional commands that allow you to inspect packages, search through PyPI, or build your own package distributions. The list of all available commands can be obtained by `pip --help` as in the following command:

```
$ pip --help
```

And it will produce the following output:

```
Usage:
  pip <command> [options]

Commands:
  install                     Install packages.
  download                    Download packages.
  uninstall                   Uninstall packages.
  freeze                      Output installed packages in requirements
format.
  list                        List installed packages.
  show                        Show information about installed
packages.
  check                       Verify installed packages have compatible
dependencies.
  config                      Manage local and global configuration.
  search                      Search PyPI for packages.
  cache                       Inspect and manage pip's wheel cache.
  wheel                       Build wheels from your requirements.
  hash                        Compute hashes of package archives.
  completion                  A helper command used for command
completion.
  debug                       Show information useful for debugging.
  help                        Show help for commands.
(...)
```

The most up-to-date information on how to install `pip` for older Python versions is available on the project's documentation page at `https://pip.pypa.io/en/stable/installing/`.

Isolating the runtime environment

When you use `pip` to install a new package from PyPI, it will be installed into one of the available **site-packages directories**. The exact location of site-packages directories is specific to the operating system. You can inspect paths where Python will be searching for modules and packages by using the `site` module as a command as follows:

```
$ python3 -m site
```

The following is an example output of running `python3 -m site` on macOS:

```
sys.path = [
    '/Users/swistakm',
    '/Library/Frameworks/Python.framework/Versions/3.9/lib/python39.
zip',
    '/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9',
    '/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/
lib-dynload',
    '/Users/swistakm/Library/Python/3.9/lib/python/site-packages',
    '/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/
site-packages',
]
USER_BASE: '/Users/swistakm/Library/Python/3.9' (exists)
USER_SITE: '/Users/swistakm/Library/Python/3.9/lib/python/site-
packages' (exists)
ENABLE_USER_SITE: True
```

The `sys.path` variable in the preceding output is a list of module search locations. These are locations that Python will attempt to load modules from. The first entry is always the current working directory (in this case, `/users/swistakm`) and the last is the global site-packages directory, often referred to as the **dist-packages directory**.

The `USER_SITE` in the preceding output describes the location of the **user site-packages directory**, which is always specific to the user that is currently invoking the Python interpreter. Packages installed in a local site-packages directory will take precedence over packages installed in the global site-packages directory.

An alternative way to obtain the site-packages is by invoking `sys.getsitepackages()`. The following is an example of using that function in an interactive shell:

```
>>> import site
>>> site.getsitepackages()
['/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages']
```

You can also obtain user site-packages directories by invoking the `sys.getusersitepackages()` function like so:

```
>>> import site
>>> site.getusersitepackages()
/Users/swistakm/Library/Python/3.9/lib/python/site-packages
```

When running `pip install`, packages will be installed in either the user or the global site-packages directory depending on several conditions evaluated in the following order:

1. **user site-packages** if the `--user` switch is specified
2. **global site-packages** if the global site-packages directory is writable to the user invoking `pip`
3. **user site-packages** otherwise

The preceding conditions simply mean that without the `--user` switch, `pip` will always attempt to install packages to a global site-packages directory and only fall back to user site-packages if that is not possible. On most operating systems where Python is available by default (many Linux distributions, macOS), the global site-packages directory of the system's Python distribution is protected from writes from non-privileged users. This means that in order to install a package in the global site-packages directory using a system's Python distributions, you will have to use a command that grants you superuser privileges, like `sudo`. On UNIX-like and Linux systems, such superuser invocation of `pip` will be as follows:

```
$ sudo -H pip install <package-name>
```

> Superuser privileges for installing system-wide Python packages are not required on Windows since it does not provide the Python interpreter by default. Also, for some other operating systems (like macOS) if you install Python from the installer available on the `python.org` website, it will be installed in such a way that the global site-packages directory will be writable to normal users.

Although installing packages directly from PyPI into the global site-packages directory is possible and in certain environments will be happening by default, it is usually not recommended and should be avoided. Bear in mind that `pip` will only install a single version of a package in the site-packages directory. If an older version is already available, the new installation will overwrite it. This may be problematic, especially if you are planning to build different applications with Python. Recommending not to install anything in the global site-packages directory may sound confusing because this is the semi-default behavior of `pip`, but there are some serious reasons for that.

As we mentioned earlier, Python is often an important part of many packages that are available through operating system package repositories and may power a lot of important services. System distribution maintainers put in a lot of effort to select the correct versions of packages to match various package dependencies.

Very often, Python packages that are available from a system's package repositories (like `apt`, `yum`, or `rpm`) contain custom patches or are purposely kept outdated to ensure compatibility with some other system components. Forcing an update of such a package, using `pip`, to a version that breaks some backward compatibility might cause bugs in some crucial system service.

Last but not least, if you're working on multiple projects in parallel, you'll notice that maintaining a single list of package versions that works for all of your projects is practically impossible. Packages evolve fast and not every change is backward compatible. You will eventually run into a situation where one of your new projects desperately needs the latest version of some library, but some other project cannot use it because there is some backward-incompatible change. If you install a package into global site-packages you will be able to use only one version of that package.

Fortunately, there is an easy solution to this problem: environment isolation. There are various tools that allow the isolation of the Python runtime environment at different levels of system abstraction. The main idea is to isolate project dependencies from packages that are required by different projects and/or system services. The benefits of this approach are as follows:

- It solves the project X depends on package 1.x but project Y needs package 4.x dilemma. The developer can work on multiple projects with different dependencies that may even collide without the risk of affecting each other.

- The project is no longer constrained by versions of packages that are provided in the developer's system distribution repositories (like `apt`, `yum`, `rpm`, and so on).

- There is no risk of breaking other system services that depend on certain package versions, because new package versions are only available inside such an environment.

- A list of packages that are project dependencies can be easily locked. Locking usually captures exact versions of all packages within all dependency chains so it is very easy to reproduce such an environment on another computer.

If you're working on multiple projects in parallel, you'll quickly find that it is impossible to maintain their dependencies without some kind of isolation.

Let's discuss the difference between application-level isolation and system-level isolation in the next section.

Application-level isolation versus system-level isolation

The easiest and most lightweight approach to isolation is to use application-level isolation through virtual environments. Python has a built-in venv module that greatly simplifies the usage and creation of such virtual environments.

Virtual environments focus on isolating the Python interpreter and the packages available within it. Such environments are very easy to set up but aren't portable, mostly because they rely on absolute system paths. This means that they cannot be easily copied between computers and operating systems without breaking things. They cannot even be moved between directories on the same filesystem. Still, they are robust enough to ensure proper isolation during the development of small projects and packages. Thanks to built-in support within Python distributions, they can also be easily replicated by your peers.

Virtual environments are usually sufficient for writing focused libraries that are independent of the operating system or projects of low complexity that don't have too many external dependencies. Also, if you write software that is to be run only on your own computer, virtual environments should be enough to provide sufficient isolation and reproducibility.

Unfortunately, in some cases, this may not be enough to ensure enough consistency and reproducibility. Despite the fact that software written in Python is usually considered very portable, not every package will behave the same on every operating system. This is especially true for packages that rely on third-party shared libraries (DLL on Windows, .so on Linux, .dylib on macOS) or make heavy use of compiled Python extensions written in either C or C++, but can also happen for pure Python libraries that use APIs that are specific to a given operating system.

In such cases, system-level isolation is a good addition to the workflow. This kind of approach usually tries to replicate and isolate complete operating systems with all of their libraries and crucial system components, either with classical operating system virtualization tools (for example, VMware, Parallels, and VirtualBox) or container systems (for example, Docker and Rocket). Some of the available solutions that provide this kind of isolation are detailed later in the *System-level environment isolation* section.

System-level isolation should be your preferred option for the development environment if you're writing software on a different computer than the one you'll be executing it on. If you are running your software on remote servers, you should definitely consider system-level isolation from the very beginning as it may save you from portability issues in the future. And you should do that regardless of whether your application relies on compiled code (shared libraries, compiled extensions) or not. Using system-level isolation is also worth considering if your application makes heavy use of external services like databases, caches, search engines, and so on. That's because many system-level isolation solutions allow you to easily isolate those dependencies too.

Since both approaches to environment isolation have their place in modern Python development, we will discuss them both in detail. Let's start with the simpler one— virtual environments using Python's `venv` module.

Application-level environment isolation

Python has built-in support for creating virtual environments. It comes in the form of a `venv` module that can be invoked directly from your system shell. To create a new virtual environment, simply use the following command:

```
$ python3.9 -m venv <env-name>
```

Here, `env-name` should be replaced with the desired name for the new environment (it can also be an absolute path). Note how we used the `python3.9` command instead of plain `python3`. That's because depending on the environment, `python3` may be linked to different interpreter versions and it is always better to be very explicit about the Python version when creating new virtual environments. The `python3.9 -m venv` commands will create a new `env-name` directory in the current working directory path. Inside, it will contain a few sub-directories:

- `bin/`: This is where the new Python executable and scripts/executables provided by other packages are stored.

> **Note for Windows users**
>
> The `venv` module under Windows uses a different naming convention for its internal structure of directories. You need to use `Scripts/`, `Libs/`, and `Include/`, instead of `bin/`, `lib/`, and `include/`, to match the development conventions commonly used on that operating system. The commands that are used for activating/deactivating the environment are also different; you need to use `ENV-NAME/Scripts/activate.bat` and `ENV-NAME/Scripts/deactivate.bat` instead of using `source` on `activate` and `deactivate` scripts.

- lib/ and include/: These directories contain the supporting library files for the new Python interpreter inside the virtual environment. New packages will be installed in ENV-NAME/lib/pythonX.Y/site-packages/.

> Many developers keep their virtual environments together with the source code and pick a generic path name like .venv or venv. Many Python **Integrated Development Environments (IDEs)** are able to recognize that convention and automatically load the libraries for syntax completion. Generic names also allow you to automatically exclude virtual environment directories from code versioning, which is generally a good idea. Git users can, for instance, add this path name to their global .gitgnore file, which lists path patterns that should be ignored when versioning the source code.

Once the new environment has been created, it needs to be activated in the current shell session. If you're using Bash as a shell, you can activate the virtual environment using the source command:

```
$ source env-name/bin/activate
```

There's also a shorter version that should work under any POSIX-compatible system regardless of the shell:

```
$ . env-name/bin/activate
```

This changes the state of the current shell session by affecting its environment variables. In order to make the user aware that they have activated the virtual environment, it will change the shell prompt by appending the (ENV-NAME) string at its beginning. To illustrate this, here is an example session that creates a new environment and activates it:

```
$ python3 -m venv example
$ source example/bin/activate
(example) $ which python
/home/swistakm/example/bin/python
(example) $ deactivate
$ which python
/usr/local/bin/python
```

The important thing to note about venv is that it does not provide any additional abilities to track what packages should be installed in it. Virtual environments are also not portable and should not be moved to another system/machine or even a different filesystem path. This means that a new virtual environment needs to be created every time you want to install your application on a new host.

Because of this, there is a best practice that's used by `pip` users to store the definition of all project dependencies in a single place. The easiest way to do this is by creating a `requirements.txt` file (this is the naming convention), with contents as shown in the following code:

```
# lines followed by hash (#) are treated as a comment.

# pinned version specifiers are best for reproducibility
eventlet==0.17.4
graceful==0.1.1

# for projects that are well tested with different
# dependency versions the version ranges are acceptable
falcon>=0.3.0,<0.5.0

# packages without versions should be avoided unless
# latest release is always required/desired
pytz
```

With such a file, all dependencies can be easily installed in a single step. The `pip install` command understands the format of such requirements files. You can specify the path to a requirements file using the `-r` flag as in the following example:

```
$ pip install -r requirements.txt
```

Remember that requirements files specify only packages to be installed and not packages that are currently in your environment. If you install something manually in your environment, it won't be reflected in your requirements file automatically. So, great care needs to be taken to keep your requirements file up to date, especially for large and complex projects.

There is the `pip freeze` command, which prints all packages in the current environment together with their versions, but it should be used carefully. This list will also include dependencies of your dependencies, so for large projects, it will quickly become very large. You will have to carefully inspect whether the list contains anything installed accidentally or by mistake.

For projects that require better reproductivity of virtual environments and strict control of installed dependencies, you may need a more sophisticated tool. We will discuss such a tool—Poetry—in the following section.

Poetry as a dependency management system

Poetry is quite a novel approach to dependency and virtual environment management in Python. It is an open-source project that aims to provide a more predictable and convenient environment for working with the Python packaging ecosystem.

As Poetry is a package on PyPI, you can install it using `pip`:

```
$ pip install --user poetry
```

> Be aware that Poetry takes care of creating Python virtual environments so it should not be installed inside of a virtual environment itself. You can install it in either user site-packages or global site-packages although user site-packages is the recommended option (see the *Isolating the runtime environment* section).

As already highlighted in the *Installing Python packages using pip* section, the above command will install the `poetry` package in your site-packages directory. Depending on your system configuration it will be either the global site-packages directory or the user site-packages directory. To avoid this ambiguity, the Poetry project creators recommend using an alternative bootstrapping method.

On macOS, Linux, and other POSIX-compatible systems Poetry can be installed using the `curl` utility:

```
$ curl -sSL https://raw.githubusercontent.com/python-poetry/poetry/
master/get-poetry.py | python
```

On Windows it can be installed using PowerShell:

```
> (Invoke-WebRequest -Uri https://raw.githubusercontent.com/python-
poetry/poetry/master/get-poetry.py -UseBasicParsing).Content | python -
```

Once installed, Poetry can be used to:

- Create new Python projects together with virtual environments
- Initialize existing projects with a virtual environment
- Manage project dependencies
- Package libraries

To create a completely new project with Poetry, you can use the `poetry new` command as in the following example:

```
$ poetry new my-project
```

The above command will create a new `my-project` directory with some initial files in it. The structure of that directory will be roughly as follows:

```
my-project/
├── README.rst
├── my_project
│   └── __init__.py
├── pyproject.toml
└── tests
    ├── __init__.py
    └── test_my_project.py
```

As you can see, it creates some files that can be used as stubs for further development. If you have a preexisting project, you can initialize Poetry within it using the `poetry init` command inside of your project directory. The difference is that it won't create any new project files except the `pyproject.toml` configuration file.

The core of Poetry is the `pyproject.toml` file, which stores the project configuration. For the `my-project` example it may have the following content:

```
[tool.poetry]
name = "my-project"
version = "0.1.0"
description = ""
authors = ["Michał Jaworski <swistakm@gmail.com>"]

[tool.poetry.dependencies]
python = "^3.9"

[tool.poetry.dev-dependencies]
pytest = "^5.2"

[build-system]
requires = ["poetry-core>=1.0.0"]
build-backend = "poetry.core.masonry.api"
```

As you can see, the `pyproject.toml` file is divided into four sections. Those are:

- `[tool.poetry]`: This is a set of basic project metadata like name, version description, and author. This information is necessary if you would like to publish your project as a package on PyPI.

- `[tool.poetry.dependencies]`: This is a list of project dependencies. On fresh projects, it lists only the Python version but can also include all package versions that normally would be described in the `requirements.txt` file.

- `[tool.poetry.dev-dependencies]`: This is a list of dependencies that require local development, like testing frameworks or productivity tools. It is common practice to have a separate list of such dependencies as they are usually not required in production environments.

- `[build-system]`: Describes Poetry as a build system used to manage the project.

> The `pyproject.toml` file is part of the official Python standard described in the PEP 518 document. You can read more information about its structure at `https://www.python.org/dev/peps/pep-0518/`.

If you create a new project or initialize an existing one using Poetry, it will be able to create a new virtual environment in a shared location whenever you need it. You can activate it using Poetry instead of "sourcing" the `activate` scripts. That's more convenient than using the plain `venv` module because you don't need to remember where the actual virtual environment is stored. The only thing you need to do is to move your shell to any place in your project source tree and use the `poetry shell` command as in the following example:

```
$ cd my-project
$ poetry shell
```

From that moment on, the current shell will have Poetry's virtual environment activated. You can verify it with either the `which python` or `python -m site` command.

Another thing that Poetry changes is how you manage dependencies. As we already mentioned, `requirements.txt` files are a very basic way of managing dependencies. They describe what packages to install but do not automatically track what has been installed in the environment through the development. If you install something with `pip` but forget to reflect that change in the `requirements.txt` file, other programmers may have a problem recreating your environment.

With Poetry, that problem is gone. There's only one way of adding dependencies to your project and it is with the `poetry add <package-name>` command. It will:

- Resolve whole dependency trees if other packages share dependencies
- Install all packages from the dependency tree in the virtual environment associated with your project
- Reflect the change in the `pyproject.toml` file

The following transcript presents the process of installation of the Flask framework within the `my-project` environment:

```
$ poetry add flask
```

This will produce an output like the following:

```
Using version ^1.1.2 for Flask

Updating dependencies
Resolving dependencies... (38.9s)

Writing lock file

Package operations: 15 installs, 0 updates, 0 removals

  • Installing markupsafe (1.1.1)
  • Installing pyparsing (2.4.7)
  • Installing six (1.15.0)
  • Installing attrs (20.3.0)
  • Installing click (7.1.2)
  • Installing itsdangerous (1.1.0)
  • Installing jinja2 (2.11.2)
  • Installing more-itertools (8.6.0)
  • Installing packaging (20.4)
  • Installing pluggy (0.13.1)
  • Installing py (1.9.0)
  • Installing wcwidth (0.2.5)
  • Installing werkzeug (1.0.1)
  • Installing flask (1.1.2)
  • Installing pytest (5.4.3)
```

And the following is the resulting `pyproject.toml` file with highlighted changes to the project dependencies:

```
[tool.poetry]
name = "my-project"
version = "0.1.0"
description = ""
authors = ["Michał Jaworski <swistakm@gmail.com>"]

[tool.poetry.dependencies]
python = "^3.9"
Flask = "^1.1.2"

[tool.poetry.dev-dependencies]
pytest = "^5.2"

[build-system]
requires = ["poetry-core>=1.0.0"]
```

The preceding transcript shows that Poetry has installed 15 packages when we asked for only one dependency. That's because Flask has its own dependencies and those dependencies have their own dependencies. Such dependencies of dependencies are called **transitive dependencies**. Libraries often have lax version specifiers like `six >=1.0.0` to denote that they are able to accept a wide range of versions. Poetry implements a **dependency resolution** algorithm to find out which set of versions can satisfy all dependency transitive dependency constraints.

The problem with transitive dependencies is their ability to change over time. Remember that libraries can have lax version specifiers for their dependencies. It is thus possible that two environments created on different dates will have different final versions of packages installed. The inability to reproduce exact versions of all transitive dependencies can be a big problem for large projects and manually tracking them in `requirements.txt` files is usually a big challenge.

Poetry solves the problem of transitive dependencies by using so-called **dependency lock files**. Whenever you are sure that your environment has a working and tested set of package versions, you can issue the following command:

```
$ poetry lock
```

This will create a really verbose `poetry.lock` file that is a complete snapshot of the dependency resolution process. That file will be then used to determine versions of transitive dependencies instead of the ordinary dependency process.

Whenever new packages are added with the `poetry add` command, Poetry will evaluate the dependency tree and update the `poetry.lock` file. The lock file approach is so far the best and most reliable way of handling transitive dependencies in your project.

> You can find more information about advanced usage of Poetry in the official documentation under `https://python-poetry.org`.

System-level environment isolation

The key enabler to the rapid iteration of software implementation is the reuse of existing software components. Don't repeat yourself—this is a common mantra of many programmers. Using other packages and modules to include them in the codebase is only a part of that mindset. What can also be considered as reused components are binary libraries, databases, system services, third-party APIs, and so on. Even whole operating systems should be considered as a component that is being reused.

The backend services of web-based applications are a great example of how complex such applications can be. The simplest software stack usually consists of a few layers. Consider some imaginary application that allows you to store some information of its users and exposes it to the internet over the HTTP protocol. It could have at least the three following layers (starting from the lowest):

- A database or other kind of storage engine
- The application code implemented in Python
- An HTTP server working in reverse proxy mode, such as Apache or NGINX

Although very simple applications can be single-layered, it rarely happens for complex applications or applications that are designed to handle very large traffic. In fact, big applications are sometimes so complex that they cannot be represented as a stack of layers but rather as a patchwork or mesh of interconnected services. Both small and big applications can use many different databases, be divided into multiple independent processes, and use many other system services for caching, queuing, logging, service discovery, and so on. Sadly, there are no limits to this complexity.

What is really important is that not all software stack elements can be isolated on the level of Python runtime environments. No matter whether it is an HTTP server, such as NGINX, or an RDBMS, such as PostgreSQL, or a shared library, those elements are usually not part of the Python distribution or Python package ecosystem and can't be encapsulated within Python's virtual environments. That's why they are considered **external dependencies** of your software.

What is very important is that external dependencies are usually available in different versions and flavors on different operating systems. For instance, if two developers are using completely different Linux distributions, let's say Debian and Gentoo, it is really unlikely that at any given time they will have access to the same version of software like NGINX through their system's package repositories. Moreover, they can be compiled using different compile-time flags (for instance, enabling specific settings), or be provided with custom extensions or distribution-specific patches.

So, making sure that everyone in a development team uses the same versions of every component is very hard without the proper tools. It is theoretically possible that all developers in a team working on a single project will be able to get the same versions of services on their development boxes. But all this effort is futile if they do not use the same operating system as they do in their **production environment**. Forcing a programmer to work on something else rather than their beloved system of choice is also not always possible.

> The **production environment**, or **production** for short, is the actual environment where your application is installed and running to serve its very purpose. For instance, the production environment for a desktop application would be the actual desktop computer on which your users install their applications. The production environment of a backend server for a web application available through the internet is usually a remote server (sometimes virtual) operating in some sort of datacenter.

The problem lies in the fact that portability is still a big challenge. Not all services will work exactly the same in the production environments as they do on the developer's machines. And this is unlikely to change. Even Python can behave differently on different systems, despite how much work is put into making it cross-platform. Usually, for Python, this is well-documented and happens only in places that interact directly with the operating system. Still, relying on the programmer's ability to remember a long list of compatibility quirks is quite an error-prone strategy.

A popular solution to this problem is isolating whole systems as the application environment. This is usually achieved by leveraging different types of system virtualization tools. Virtualization, of course, may have an impact on performance; but with modern CPUs that have hardware support for virtualization, the performance loss is greatly reduced. On the other hand, the list of possible gains is very long:

- The development environment can exactly match the system version, services, and shared libraries used in production, which helps to solve compatibility issues.

- Definitions for system configuration tools, such as Puppet, Chef, or Ansible (if used), can be reused to configure both the production and development environments.

- The newly hired team members can easily hop into the project if the creation of such environments is automated.

- The developers can work directly with low-level system features that may not be available on operating systems they use for work. For example, **Filesystem in Userspace** (**FUSE**) is a feature of Linux operating systems that you could not work with on Windows without virtualization.

In the next section, we'll take a look at two different approaches to achieving the system-level isolation of development environments.

Containerization versus virtualization

There are two main ways that system-level isolation techniques can be used for development purposes:

- **Machine virtualization**, which emulates the whole computer system
- **Operating system-level virtualization**, known also as **containerization**, which isolates complete user spaces within a single operating system

Machine virtualization techniques concentrate on emulating whole computer systems within other computer systems. Think of it as providing virtual hardware that can be run as a piece of software on your own computer. As this is full hardware emulation, it gives you the possibility to run any operating system within your host environments. This is the technology that drives the infrastructure of **Virtual Private Server** (**VPS**) and cloud computing providers, as it allows you to run multiple independent and isolated operating systems within a single host computer.

This is also a convenient method of running many operating systems for development purposes, as starting a new operating system does not require rebooting your computer. You can also easily dispose of virtual machines when not needed. That's something that cannot be done easily with typical multi-boot system installation.

Operating system-level virtualization, on the other hand, does not rely on emulating the hardware. It encapsulates a user-space environment (shared libraries, resource constraints, filesystem volumes, code, and so on) in the form of containers that cannot operate outside the strictly defined container environment. All containers are running on the same operating system kernel but cannot interfere with each other unless you explicitly allow them to.

Operating system-level virtualization does not require emulation of the hardware. Still, it can set specific constraints on the use of system resources like storage space, CPU time, RAM, or network. These constraints are managed only by the system kernel, so the performance overhead is usually smaller than in machine virtualization. That's why operating system-level virtualization is often called **lightweight virtualization**.

Usually, a container contains only application code and its system-level dependencies, mostly shared libraries or runtime binaries like the Python interpreter, but can be as large as you want. Images for Linux containers are often based on whole system distributions like Debian, Ubuntu, or Fedora. From the perspective of processes running inside a container, it looks like a completely isolated system environment.

When it comes to system-level isolation for development purposes, both methods provide a similarly sufficient level of isolation and reproducibility. Nevertheless, due to its more lightweight nature, operating system-level virtualization seems to be more favored by developers as it allows cheaper, faster, and more streamlined usage of such environments together with convenient packaging and portability. This is especially useful for programmers that work on multiple projects in parallel or need to share their environments with other programmers.

There are two leading tools for providing system-level isolation of development environments:

- Docker for operating system-level virtualization
- Vagrant for machine virtualization

Docker and Vagrant seem to overlap in features. The main difference between them is the reason why they were built. Vagrant was built primarily as a tool for development. It allows you to bootstrap the whole virtual machine with a single command but is rarely used to simply pack such an environment as a complete artifact that could be easily delivered to a production environment and executed as is. Docker, on the other hand, is built exactly for that purpose—preparing complete containers that can be sent and deployed to production as a complete package. If implemented well, this can greatly improve the process of product deployment.

Due to some implementation nuances, the environments that are based on containers may sometimes behave differently than environments based on virtual machines. They also do not package the operating system kernel, so for code that is highly operating system-specific, they may not always behave the same on every host. Also, if you decide to use containers for development, but don't decide to use them on target production environments, you'll lose some of the consistency guarantees that were the main reason for environment isolation.

But, if you already use containers in your target production environments, then you should always replicate production conditions in the development stage using the same technique. Fortunately, Docker, which is currently the most popular container solution, provides an amazing `docker-compose` tool that makes the management of local containerized environments extremely easy.

Containers are a great alternative to full machine virtualization. It is a lightweight method of virtualization, where the kernel and operating system allow multiple isolated user-space instances to be run. If your operating system supports containers natively, this method of virtualization will require less overhead than full machine virtualization.

Virtual environments using Docker

Software containers got their popularity mostly thanks to Docker, which is one of the available implementations for the Linux operating system.

Docker allows you to describe an image of the container in the form of a simple text document called a **Dockerfile**. Images from such definitions can be built and stored in image repositories. Image repositories allow multiple programmers to reuse existing images without the need to build them all by themselves. Docker also supports incremental changes, so if new things are added to the container then it does not need to be recreated from scratch.

Docker is an operating system virtualization method for Linux operating systems, so it isn't natively supported by kernels of Windows and macOS. Still, this doesn't mean that you can't use Docker on Windows or macOS. On those operating systems, Docker becomes kind of a hybrid between machine virtualization and operating system-level virtualization. Docker installation on those two systems will create an intermediary virtual machine with the Linux operating system that will act as a host for your containers. The Docker daemon and command-line utilities will take care of proxying any traffic and images between your own operating system and containers running on that virtual machine seamlessly.

> You can find Docker installation instructions on `https://www.docker.com/get-started`.

The existence of an intermediary virtual machine means that Docker on Windows or macOS isn't as lightweight as it is on Linux. Still, the performance overhead shouldn't be noticeably higher than the performance overhead of other development environments based strictly on machine virtualization.

Writing your first Dockerfile

Every Docker-based environment starts with a Dockerfile. A Dockerfile is a description of how to create a Docker image. You can think about the Docker images in a similar way to how you would think about images of virtual machines. It is a single file (composed of many layers) that encapsulates all system libraries, files, source code, and other dependencies that are required to execute your application.

Every layer of a Docker image is described in the Dockerfile by a single instruction in the following format:

```
INSTRUCTION arguments
```

Docker supports plenty of instructions, but the most basic ones that you need to know in order to get started are as follows:

- `FROM <image-name>`: This describes the base image that your image will be based on. They are often based on common Linux system distributions and usually come with additional libraries and software installed. The default Docker images repository is called Docker Hub. It can be accessed for free and browsed at `https://hub.docker.com/`.

- `COPY <src>... <dst>`: This copies files from the local build context (usually project files) and adds them to the container's filesystem.

- `ADD <src>... <dst>`: This works similarly to `COPY` but automatically unpacks archives and allows `<src>` to be URLs.

- `RUN <command>`: This runs a specified command on top of previous layers. After execution, it commits changes that this command made to the filesystem as a new image layer.

- `ENTRYPOINT ["<executable>", "<param>", ...]`: This configures the default command to be run as your container starts. If no entry point is specified anywhere in the image layers, then Docker defaults to `/bin/sh -c`, which is the default shell of a given image (usually Bash but can also be another shell).

- `CMD ["<param>", ...]`: This specifies the default parameters for image entry points. Knowing that the default entry point for Docker is `/bin/sh -c`, this instruction can also take the form of `CMD ["<executable>", "<param>", ...]`. It is recommended to define the target executable directly in the `ENTRYPOINT` instruction and use `CMD` only for default arguments.

- `WORKDIR <dir>`: This sets the current working directory for any of the following `RUN`, `CMD`, `ENTRYPOINT`, `COPY`, and `ADD` instructions.

To properly illustrate the typical structure of a Dockerfile, we will try to **dockerize** a simple Python application. Let's imagine we want to create an HTTP echo web server that replies back with details of the HTTP request it received. We will use Flask, which is a very popular Python web microframework.

> Flask isn't a part of the Python standard library. You can install it in your environment using `pip` as follows:
>
> ```
> $ pip install flask
> ```
>
> You can find more information about the Flask framework at `https://flask.palletsprojects.com/`.

The code of our application, which would be saved in a Python script, `echo.py`, could be as follows:

```python
from flask import Flask, request
app = Flask(__name__)


@app.route('/')
def echo():
    return (
        f"METHOD: {request.method}\n"
        f"HEADERS:\n{request.headers}"
        f"BODY:\n{request.data.decode()}"
    )


if __name__ == '__main__':
    app.run(host="0.0.0.0")
```

Our script starts with the import of the `Flask` class and the `request` object. The instance of the `Flask` class represents our web application. The `request` object is a special global object that always represents the context of the currently processed HTTP request.

`echo()` is a so-called **view function**, which is responsible for handling incoming requests. `@app.route('/')` registers the `echo()` view function under the / path. This means that only requests that match the / path will be dispatched to this view function. Inside of our view, we read incoming request details (method, headers, and body) and return them in text form. Flask will include that text output in the request response body.

Our script ends with the call to the `app.run()` method. It starts the local development server of our application. This development server is not intended for production environment use but is good enough for development purposes and greatly simplifies our example.

If you have the Flask package installed, you can run your application using the following command:

```
$ python3 echo.py
```

The above command will start the Flask development server on port `5000`. You can either visit the `http://localhost:5000` address in your browser or use the command-line utility.

The following is an example of invoking a GET request using `curl`:

```
$ curl localhost:5000
METHOD: GET
HEADERS:
Host: localhost:5000
User-Agent: curl/7.64.1
Accept: */*

BODY:
```

As we confirmed that our application returned the HTTP details of the request it received, we're almost ready to dockerize it. The structure of our project files could be as follows:

```
.
├── Dockerfile
├── echo.py
└── requirements.txt
```

The `requirements.txt` file will contain only one entry, `flask==1.1.2`, to make sure our image will always use the same version of Flask. Before we jump to the Dockerfile, let's decide how we want our image to work. What we want to achieve is the following:

- Hide some complexity from the user—especially the fact that we use Python and Flask
- Package the Python 3.9 executable with all its dependencies
- Package all project dependencies defined in the `requirements.txt` file

Knowing the above requirements, we are ready to write our first Dockerfile. It will take the following form:

```
FROM python:3.9-slim
WORKDIR /app/

COPY requirements.txt .
RUN pip install -r requirements.txt

COPY echo.py .
ENTRYPOINT ["python", "echo.py"]
```

`FROM python:3.9-slim` defines the base image for our custom container image. Python has a collection of official images on Docker Hub and `python:3.9-slim` is one of them. `3.9-slim` is the tag of the image including Python 3.9 with only a minimal set of system packages needed to run Python. It is usually a sensible starting point for Python-based application images.

In the next section, we will learn how to build a Docker image from the above Dockerfile and how to run our container.

Running containers

Before your container can be started, you'll first need to build an image defined in the Dockerfile. You can build the image using the following command:

```
$ docker build -t <name> <path>
```

The `-t <name>` argument allows us to name the image with a readable identifier. It is totally optional, but without it, you won't be able to easily reference a newly created image. The `<path>` argument specifies the path to the directory where your Dockerfile is located. Let's assume that we were already running the command from the root of the project presented in the previous section. We also want to tag our image with the name `echo`. The `docker build` command invocation will be the following:

```
$ docker build -t echo .
```

Its output may be as follows:

```
Sending build context to Docker daemon   16.8MB
Step 1/6 : FROM python:3.9-slim
3.9-slim: Pulling from library/python
bb79b6b2107f: Pull complete
35e30c3f3e2b: Pull complete
b13c2c0e2577: Pull complete
263be93302fa: Pull complete
30e7021a7001: Pull complete
Digest: sha256:c13fda093489a1b699ee84240df4f5d0880112b9e09ac21c5d687500
3d1aa927
Status: Downloaded newer image for python:3.9-slim
 ---> a90139e6bc2f
Step 2/6 : WORKDIR /app/
 ---> Running in fd85d9ac44a6
Removing intermediate container fd85d9ac44a6
 ---> b781318cdec7
```

```
Step 3/6 : COPY requirements.txt .
 ---> 6d56980fedf6
Step 4/6 : RUN pip install -r requirements.txt
 ---> Running in 5cd9b86ac454
(...)
Successfully installed Jinja2-2.11.2 MarkupSafe-1.1.1 Werkzeug-1.0.1
click-7.1.2 flask-1.1.2 itsdangerous-1.1.0
Removing intermediate container 5cd9b86ac454
 ---> 0fbf85e8f6da
Step 5/6 : COPY echo.py .
 ---> a546d22e8c98
Step 6/6 : ENTRYPOINT ["python", "echo.py"]
 ---> Running in 0b4e57680ac4
Removing intermediate container 0b4e57680ac4
 ---> 0549d15959ef
Successfully built 0549d15959ef
Successfully tagged echo:latest
```

Once created, you can inspect the list of available images using the `docker images` command:

```
$ docker images
REPOSITORY      TAG       IMAGE ID       CREATED           SIZE
echo            latest    0549d15959ef   About a minute ago 126MB
python          3.9-slim  a90139e6bc2f   10 days ago       115MB
```

> **The shocking size of container images**
>
> Our image has a size of 126 MB because it actually captures the whole Linux system distribution needed for running our Python application. It may sound like a lot, but it isn't really anything to worry about. For the sake of brevity, we have used a base image that is simple to use. There are other images that have been crafted specially to minimize this size, but these are usually dedicated to more experienced Docker users. Also, thanks to the layered structure of Docker images, if you're using many containers, the base layers can be cached and reused, so an eventual space overhead is rarely an issue. In the preceding example, the total size of storage used for both images will be only 126 MB because the `echo:latest` image only adds 11 MB on top of the `python:3.9-slim` image.

Once your image is built and tagged, you can run a container using the `docker run` command. Our container is an example of a web service, so we will have to additionally tell Docker that we want to publish the container's ports by binding them locally:

```
docker run -it --rm --publish 5000:5000 echo
```

Here is an explanation of the specific arguments of the preceding command:

- `-it`: These are actually two concatenated options: `-i` and `-t`. The `-i` (for interactive) keeps STDIN open, even if the container process is detached, and `-t` (for `tty`) allocates pseudo-TTY for the container. TTY stands for teletypewriter and on Linux and UNIX-like operating systems represents the terminal connected to a program's standard input and output. In short, thanks to these two options, we will be able to see live logs from our application and ensure that the keyboard interrupt will cause the process to exit. It will simply behave the same way as we would start Python, straight from the command line.

- `--rm`: Tells Docker to automatically remove the container when it exits. Without this option, the container will be kept so you can reattach to it in order to diagnose its state. By default, Docker does not remove containers just to make debugging easier. They can quickly pile up on your disk so good practice is to use `--rm` by default unless you really need to keep the exited container for later review.

- `--publish 5000:5000`: Tells Docker to publish the container's port `5000` by binding port `5000` on the host's interface. You can use this option to also remap application ports. If you would like, for instance, to expose the echo application on port `8080` locally, you could use the `--publish 8080:5000` argument.

Building and running your own images using the `docker` command is quite simple and straightforward but can become cumbersome after a while. It requires using quite long command invocations and remembering a lot of custom identifiers. It can be quite inconvenient for more complex environments. In the next section, we will see how a Docker workflow can be simplified with the Docker Compose utility.

Setting up complex environments

While the basic usage of Docker is pretty straightforward for basic setups, it can be a bit overwhelming once you start to use it in multiple projects. It is really easy to forget about specific command-line options, or which ports should be published on which images.

But things start to get really complicated when you have one service that needs to communicate with others. A single Docker container should only contain one running process.

This means that you really shouldn't put any additional process supervision tools, such as Supervisor and Circus, into the container image, and instead set up multiple containers that communicate with each other. Each service may use a completely different image, provide different configuration options, and expose ports that may or may not overlap. If you want to run multiple different processes, each process should be a separate container.

Large production deployments of containers use dedicated container orchestration systems like Kubernetes, Nomad, or Docker Swarm to keep track of all containers and their execution details like images, ports, volumes, ports, configuration, and so on. You could use one of those tools locally, but that would be overkill for development purposes.

The best container development tool that you can use on your computer that works well for both simple and complex scenarios is Docker Compose. Docker Compose is usually distributed with Docker, but in some Linux distributions (for example, Ubuntu), it may not be available by default. In such a case, it must be installed as a separate package from the system package repository. Docker Compose provides a powerful command-line utility named `docker-compose` and allows you to describe multi-container applications using the YAML syntax.

Compose expects the specially named `docker-compose.yml` file to be in your project root directory. An example of such a file for our previous project could be as follows:

```yaml
version: '3.8'

services:
  echo-server:
    # this tell Docker Compose to build image from
    # local (.) directory
    build: .

    # this is equivalent to "-p" option of
    # the "docker run" command
    ports:
    - "5000:5000"

    # this is equivalent to "-t" option of
    # the "docker run" command
    tty: true
```

If you create such a `docker-compose.yml` file in your project, then your whole application environment can be started and stopped with two simple commands:

- `docker-compose up`: This starts all containers defined in the `docker-compose.yml` file and actively prints their standard output
- `docker-compose down`: This stops all containers started by `docker-compose` in the current project directory

Docker Compose will automatically build your image if it hasn't been built yet. That's a great way of encoding the development environment in the configuration file. If you work with other programmers, you can provide one `docker-compose.yml` file for your project. This way, setting up a fully working local development environment will be a matter of one `docker-compose up` command. The `docker-compose.yml` file should definitely be versioned together with the rest of your code if you use the code versioning tools.

Moreover, if your application requires additional external services, you can easily add them to your Docker Compose environment instead of installing them on your host system. Consider the following example that adds one instance of a PostgreSQL database and Redis memory storage using official Docker Hub images:

```yaml
version: '3.8'

services:
  echo-server:
    build: .
    ports:
    - "5000:5000"
    tty: true

  database:
    image: postgres

  cache:
    image: redis
```

> Docker Hub is the official repository of Docker images. Many open-source developers host their official project images there. You can find more info about Docker Hub at `https://hub.docker.com`.

It is as simple as that. To ensure better reproducibility, you should always specify version tags of external images (like `postgres:13.1` and `redis:6.0.9`). That way you will ensure everyone using your `docker-compose.yml` file will be using exactly the same versions of external services. Thanks to Docker Compose you can use multiple versions of the same service simultaneously without any interference. That's because different Docker Compose environments are by default isolated on the network level.

Useful Docker and Docker Compose recipes for Python

Docker and containers in general are such a vast topic that it is impossible to cover them in one short section of this book. Thanks to Docker Compose, it is really easy to start working with Docker without knowing a lot about how it works internally. If you're new to Docker, you'll have to eventually slow down a bit, take the Docker documentation, and read it thoroughly.

> The official Docker documentation can be found at `https://docs.docker.com/`.

The following are some quick tips and recipes that allow you to defer that moment and solve most of the common problems that you may have to deal with sooner or later.

Reducing the size of containers

A common concern of new Docker users is the size of their container images. It's true that containers provide a lot of space overhead compared to plain Python packages, but it is usually nothing if we compare this to the size of images for virtual machines. However, it is still very common to host many services on a single virtual machine, but with a container-based approach, you should definitely have a separate image for every service. This means that with a lot of services, the overhead may become noticeable.

If you want to limit the size of your images, you can use two complementary techniques:

- **Use a base image that is designed specifically for that purpose**: Alpine Linux is an example of a compact Linux distribution that is specifically tailored to provide very small and lightweight Docker images. The base image is around 5 MB in size and provides an elegant package manager that allows you to keep your images compact.

- **Take into consideration the characteristics of the Docker overlay filesystem**: Docker images consist of layers where each layer encapsulates the difference in the root filesystem between itself and the previous layer. Once the layer is committed, the size of the image cannot be reduced. This means that if you need a system package as a build dependency, and it may be later discarded from the image, then instead of using multiple RUN instructions, it may be better to do everything in a single RUN instruction with chained shell commands to avoid excessive layer commits.

These two techniques can be illustrated by the following Dockerfile:

```
FROM alpine:3.13
WORKDIR /app/

RUN apk add --no-cache python3

COPY requirements.txt .
RUN apk add --no-cache py3-pip && \
    pip3 install -r requirements.txt && \
    apk del py3-pip

COPY echo.py .
CMD ["python", "echo.py"]
```

> The above example uses the `alpine:3.12` base image to illustrate the technique of cleaning up needless dependencies before committing the layer. Unfortunately, the `apk` manager in the Alpine distribution doesn't give proper control of which version of Python will be installed. That's why recommended Alpine base images for Python projects come from the official Python repository. For Python 3.9 that would be the `python:3.9-alpine` base image.

The `--no-cache` flag of `apk` (Alpine's package manager) has two effects. First, it will cause `apk` to ignore the existing cache of package lists so it will install the latest package version that is available officially in the package repository. Second, it won't update the existing package lists cache, so the layer created with this instruction will be slightly smaller than using the `--update-cache` flag that is otherwise necessary to install the package in its latest version. The difference is not that big (probably around 2 MB) but those small chunks of cache can add up in bigger images that have many layers of `apk add` invocations. Package managers of different Linux distributions usually offer a similar way of disabling their package list caches.

The second `RUN` instruction is an example of taking into account the way Docker image layers work. On Alpine, the Python package doesn't come with `pip` installed so we need to install it on our own. Generally, after all the required Python packages have been installed, `pip` is no longer required and can be removed. We could use the `ensurepip` module to bootstrap `pip` but then we wouldn't have an obvious way of removing it. Instead, we use a long-chained instruction that relies on `apk` to install the `py3-pip` package and remove it after installing the other Python packages. This trick on Alpine 3.13 may even save us up to 16 MB.

If you run the Docker `images` command, you will see that there is a substantial size difference between images based on Alpine and `python:slim` base images:

```
$ docker images
REPOSITORY    TAG       IMAGE ID        CREATED            SIZE
echo-alpine   latest    e7e3a2bc7b71    About a minute ago  53.7MB
echo          latest    6b036d212e8f    40 minutes ago      126MB
```

The resulting image is now more than two times smaller than the one based on the `python:3.9-slim` image. That's mostly due to a streamlined Alpine distribution that is around 5 MB in total. Without our trick of deleting `pip` and using the `--no-cache` flag, the image size would probably be around 72 MB (package lists caches are around 2 MB, `py3-pip` around 16 MB). In total it allowed us to save almost 25% of the size. Such a size reduction will not be that meaningful for larger applications with more dependencies where 18 MB doesn't make that much of a difference. Still, this technique can be used for other build-time dependencies. Some packages, for instance, require additional compilers like `gcc` (GNU Compiler Collection) and extra header files at the time of installation. In such a situation, you could use the same pattern to avoid having the full GNU Compiler Collection in the final image. And that actually can have quite a big impact on the image size.

Addressing services inside of a Docker Compose environment

Complex applications often consist of multiple services that communicate with each other. Compose allows us to define such applications with ease. The following is an example `docker-compose.yml` file that defines the application as a composition of two services:

```yaml
version: '3.8'

services:
  echo-server:
    build: .
    ports:
```

```
    - "5000:5000"
    tty: true

  database:
    image: postgres
    restart: always
```

The preceding configuration defines two services:

- `echo-server`: This is our echo application service container with the image built from the local Dockerfile
- `database`: This is a PostgreSQL database container from an official `postgres` Docker image

We assume that the `echo-server` service wants to communicate with the `database` service over the network. In order to set up such communications, we need to know the service IP address or hostname so that it can be used as an application configuration. Thankfully, Docker Compose is a tool that was designed exactly for such scenarios, so it will make it a lot easier for us.

Whenever you start your environment with the `docker-compose up` command, Docker Compose will create a dedicated Docker network by default and will register all services in that network using their names as their hostnames. This means that the `echo-server` service can use the `database:5432` address to communicate with the database (5432 is the default PostgreSQL port), and any other service in that Docker Compose environment will be able to access the HTTP endpoint of the `echo-server` service under the `http://echo-server:80` address.

Even though the service hostnames in Docker Compose are easily predictable, it isn't good practice to hardcode any addresses in your application code. The best approach would be to provide them as environment variables that can be read by your application on startup. The following example shows how arbitrary environment variables can be defined for each service in a `docker-compose.yml` file:

```
version: '3.8'

services:
  echo-server:
    build: .
    ports:
    - "5000:5000"
    tty: true
    environment:
      - DATABASE_HOSTNAME=database
```

```
      - DATABASE_PORT=5432
      - DATABASE_PASSWORD=password


  database:
    image: postgres
    restart: always
    environment:
      POSTGRES_PASSWORD: password
```

The highlighted lines provide environment variables that tell our echo server what the hostname and port of the database are. Environment variables are the most recommended way of providing configuration parameters for containers.

> Docker containers are ephemeral. This means that once the container is removed (usually on exit), its internal filesystem changes are lost. For databases, this means that if you don't want to lose data in the database running in the container, you should mount a volume inside a container under the directory where the data is supposed to be stored. Maintainers of Docker images for databases usually document how to mount such volumes, so always refer to the documentation of the Docker image you are using if you want to keep database data safe. An example of using Docker volumes for slightly different purposes is shown in the *Adding live reload for absolutely any code* section.

Communicating between Docker Compose environments

If you build a system composed of multiple independent services and/or applications, you will very likely want to keep their code in multiple independent code repositories (projects). The docker-compose.yml files for every Docker Compose application are usually kept in the same code repository as the application code. The default network that was created by Compose for a single application is isolated from the networks of other applications. So, what can you do if you suddenly want your multiple independent applications to communicate with each other?

Fortunately, this is another thing that is extremely easy with Compose. The syntax of the docker-compose.yml file allows you to define a named external Docker network as the default network for all services defined in that configuration.

The following is an example configuration that defines an external network
named `my-interservice-network`:

```yaml
version: '3.8'

networks:
  default:
    external:
      name: my-interservice-network

services:
  webserver:
    build: .
    ports:
    - "80:80"
    tty: true
    environment:
      - DATABASE_HOSTNAME=database
      - DATABASE_PORT=5432
      - DATABASE_PASSWORD=password

  database:
    image: postgres
    restart: always
    environment:
      POSTGRES_PASSWORD: password
```

Such external networks are not managed by Docker Compose, so you'll have to
create it manually with the `docker network create` command, as follows:

```
$ docker network create my-interservice-network
```

Once you have done this, you can use this external network in other `docker-compose.yml` files for all applications that should have their services registered in the
same network. The following is an example configuration for other applications that
will be able to communicate with both `database` and `webserver` services over `my-interservice-network`, even though they are not defined in the same `docker-compose.yml` file:

```yaml
version: '3.8'

networks:
  default:
```

```yaml
    external:
      name: my-interservice-network

services:
  other-service:
    build: .
    ports:
    - "80:80"
    tty: true
    environment:
      - DATABASE_HOSTNAME=database
      - DATABASE_PORT=5432
      - ECHO_SERVER_ADDRESS=http://echo-server:80
```

The above approach allows you to start two independent Docker Compose environments in separate shells. All services will be able to communicate with each other through a shared Docker network.

Delaying code startup until service ports are open

If you run `docker-compose up`, all services will be started at the same time. You can control to some extent the service startup using the `depends_on` key in the service definition as in the following example:

```yaml
version: '3.8'

services:
  echo-server:
    build: .
    ports:
    - "5000:5000"
    tty: true
    depends_on:
      - database

  database:
    image: postgres
    environment:
      POSTGRES_PASSWORD: password
```

The preceding setup will make sure that our echo server will be started after the database service. Unfortunately, it is not always enough to ensure proper startup ordering of services within the development environment.

Consider a situation where `echo-server` would have to read something from the database immediately after starting. Docker Compose will make sure that services will be started in order but will not make sure that PostgreSQL will be ready to actually accept connections from the echo server. That's because PostgreSQL initialization can take a couple of seconds.

The solution for this is pretty simple. There are numerous scripting utilities that allow you to test if a specific network port is open before proceeding with the execution of a command. One such utility is named `wait-for-it` and is actually written in Python so you can easily install it with `pip`.

You can invoke `wait-for-it` using the following syntax:

```
$ wait-for-it --service <service-address> -- command [...]
```

The `-- command [...]` usage pattern is a common pattern for utilities that wrap different command execution where `[...]` represents any set of arguments for `command`. The `wait-for-it` process will try to create a TCP connection and when it succeeds, it will execute `command [...]`. For instance, if we would like to wait for localhost connection on port `2000` before starting the `python echo.py` command we would simply execute:

```
$ wait-for-it --service localhost:2000 -- python echo.py
```

The following is an example of a modified `docker-compose.yml` file that elegantly overrides the default Docker image command and decorates it with the call to the `wait-for-it` utility to ensure our echo server starts only when it would be able to connect to the database:

```yaml
version: '3.8'

services:
  echo-server:
    build: .
    ports:
    - "5000:5000"
    tty: true
    depends_on:
      - database
    command:
      wait-for-it --service database:5432 --
      python echo.py

  database:
```

```
    image: postgres
    environment:
      POSTGRES_PASSWORD: password
```

`wait-for-it` by default times out after 15 seconds. After that timeout, it will start the process after the `--` mark regardless of whether it succeeded in connecting or not. You can disable timeout using the `--timeout 0` argument. Without the timeout, `wait-for-it` will wait indefinitely.

Adding live reload for absolutely any code

When developing a new application, we usually work with code iteratively. We implement changes and see results. We either verify the code manually or run the tests. There is a constant feedback loop.

With Docker, we need to enclose our code in the container image to make it work. But running `docker build` or `docker-compose build` every time you make a change in your host system would be highly counterproductive.

That's why the best way to provide code to the container while working with Docker in the development stage is through Docker volumes. The idea is to bind your local filesystem directory to the container's internal filesystem path. That way any changes made to the files in the host's filesystem will be automatically reflected inside of the container. With Docker Compose, it is extremely easy as it allows you to define volumes in the service configuration. The following is a modified version of our `docker-compose.yml` file for the echo service that mounts the project's root directory under the `/app/` path:

```yaml
version: '3.8'

services:
  echo-server:
    build: .
    ports:
    - "5000:5000"
    tty: true
    volumes:
      - .:/app/
```

Changes happening on mounted Docker volumes are actively propagated on both sides. Many Python frameworks or servers support active hot reloading whenever they notice that your code has changed. This dramatically improves the development experience because you can see how the behavior of your application changes as you write it and without the need for manual restarts.

Probably not every piece of code you write will be built using a framework that supports active reloading. Fortunately, there is a great Python package named `watchdog` that allows you to reload any application watching code changes. This package provides a handful `watchmedo` utility that similarly to `wait-for-it` can wrap any process execution.

> The `watchmedo` utility from the `watchdog` package requires some additional dependencies in order to execute. To install that package with extra dependencies use the following `pip install` syntax:
>
> ```
> pip install watchdog[watchmedo]
> ```

The following is the basic usage format for reloading specified processes whenever there is a change to any Python file in the current working directory:

```
$ watchmedo auto-restart --patterns "*.py" --recursive -- command [...]
```

The `--patterns "*.py"` options indicate which files the `watchmedo` process should monitor for changes. The `--recursive` flag makes it traverse the current working directory recursively so it will be able to pick up changes made even if they are nested deep down in the directory tree. The `-- command [...]` usage pattern is the same as the `wait-for-it` command mentioned in *Delaying code startup until service ports are open*. It simply means that everything after the `--` mark will be treated as a single command with (optional) arguments. `watchmedo` starts that command and restarts it whenever it discovers a change in the monitored files.

If you install the `watchdog` package in your Docker image, you will be able to elegantly include it in your `docker-compose.yml` as in the following example:

```yaml
version: '3.8'

services:
  echo-server:
    build: .
    ports:
    - "5000:5000"
    tty: true
    depends_on:
      - database
    command:
      watchmedo auto-restart --patterns "*.py" --recursive --
      python echo.py
    volumes:
      - .:/app/
```

The above Docker Compose setup will restart the process inside of a container every time there is a change to your Python code. In our example, this will be any file with the `.py` extension that lives under the `/app/` path. Thanks to mounting the source directory as a Docker volume, the `watchmedo` utility will be able to pick up any change made on the host filesystem and restart as soon as you save changes in your editor.

Development environments with Docker and Docker Compose are extremely useful and convenient but have their limitations. The obvious one is that they only allow you to run your code under the Linux operating system. Even though Docker is available for macOS and Windows, it still relies on a Linux virtual machine as an intermediary layer, so your Docker containers will still be running under Linux. If you need to develop your application as if it were running exactly on a specific system that is different from Linux, you need a completely different approach to environment isolation. In the next section, we will learn about one such tool.

Virtual development environments using Vagrant

Although Docker together with Docker Compose provides a very good foundation for creating reproducible and isolated development environments, there are cases where a real virtual machine will simply be a better (or only) choice. An example of such a situation may be a need to do some system programming for an operating system different than Linux.

Vagrant currently seems to be one of the most popular tools for developers to manage virtual machines for the purpose of local development. It provides a simple and convenient way to describe development environments with all system dependencies in a way that is directly tied to the source code of your project. It is available for Windows, macOS, and a few popular Linux distributions (refer to `https://www.vagrantup.com`).

It does not have any additional dependencies. Vagrant creates new development environments in the form of virtual machines or containers. The exact implementation depends on a choice of virtualization providers. VirtualBox is the default provider, and it is bundled with the Vagrant installer, but additional providers are available as well. The most notable choices are VMware, Docker, **Linux Containers** (**LXC**), and Hyper-V.

The most important configuration is provided to Vagrant in a single file named a `Vagrantfile`. It should be independent for every project. The following are the most important things it provides:

- Choice of virtualization provider
- A box, which is used as a virtual machine image
- Choice of provisioning method
- Shared storage between the virtual machine and the virtual machine's host
- Ports that need to be forwarded between the virtual machine and its host

The syntax language for a `Vagrantfile` is Ruby. The example configuration file provides a good template to start the project and has excellent documentation, so knowledge of this language is not required. The template configuration can be created using a single command:

```
$ vagrant init
```

This will create a new file named `Vagrantfile` in the current working directory. The best place to store this file is usually the root of the related project sources. This file is already a valid configuration that will create a new virtual machine using the VirtualBox provider and box image based on an Ubuntu Linux distribution. The default `Vagrantfile` content that's created with the `vagrant init` command contains a lot of comments that will guide you through the complete configuration process.

The following is a minimal example of a `Vagrantfile` for the Python 3.9 development environment based on the Ubuntu operating system, with some sensible defaults that, among others, enable `port 80` forwarding in case you want to do some web development with Python:

```ruby
Vagrant.configure("2") do |config|
  # Every Vagrant development environment requires a box.
  # You can search for boxes at https://vagrantcloud.com/search.
  # Here we use Bionic version Ubuntu system for x64 architecture.
  config.vm.box = "ubuntu/bionic64"

  # Create a forwarded port mapping which allows access to a specific
  # port within the machine from a port on the host machine and only
  # allow access via 127.0.0.1 to disable public access
  config.vm.network "forwarded_port", guest: 80, host: 8080, host_ip:
"127.0.0.1"

  config.vm.provider "virtualbox" do |vb|
    vb.gui = false
    # Customize the amount of memory on the VM:
    vb.memory = "1024"
  end
```

```
  # Enable provisioning with a shell script.
  config.vm.provision "shell", inline: <<-SHELL
    apt-get update
    apt-get install python3.9 -y
  SHELL
end
```

In the preceding example, we have set the additional provisioning of system packages with a simple shell script inside of the `config.vm.provision` section. The default virtual machine image provided by the `ubuntu/bionic64` "box" does not include the Python 3.9 version, so we need to install it using the `apt-get` package manager.

When you feel that the `Vagrantfile` is ready, you can run your virtual machine using the following command:

```
$ vagrant up
```

The initial startup can take a few minutes, because the actual box image must be downloaded from the web. There are also some initialization processes that may take a while every time the existing virtual machine is brought up, and the amount of time depends on the choice of provider, image, and your system's performance. Usually, once the image has been downloaded, this takes only a couple of seconds. When the Vagrant environment is up and running, you can connect to it through SSH using the following shell shorthand:

```
$ vagrant ssh
```

This can be done anywhere in the project source tree below the location of the `Vagrantfile`. For the developers' convenience, Vagrant will traverse all directories above the user's current working directory in the filesystem tree, looking for the configuration file and matching it with the related virtual machine instance. Then, it establishes the secure shell connection, so the development environment can be interacted with just like an ordinary remote machine. The only difference is that the whole project source tree (root defined as the location of the `Vagrantfile`) is available on the virtual machine's filesystem under `/vagrant/`. This directory is automatically synchronized with your host filesystem, so you can normally use an IDE or code editor of your choice on the host and simply treat the SSH session to your Vagrant virtual machine just like a normal local shell session.

Popular productivity tools

Almost every open-source Python package that has been released on PyPI is a kind of productivity booster—it provides ready-to-use solutions to some problem. That way we don't have to reinvent the wheel all the time. Some could also say that Python itself is all about productivity. Almost everything in this language and the community surrounding it seems to be designed to make software development as productive as possible.

This creates a positive feedback loop. Since writing code with Python is fun and easy, a lot of programmers use their free time to create tools that make it even easier and more fun. And this fact will be used here as a basis for a very subjective and non-scientific definition of a productivity tool—a piece of software that makes development easier and more fun.

By nature, productivity tools focus mainly on certain elements of the development process, such as testing, debugging, and managing packages, and are not core parts of the products that they help to build. In some cases, they may not even be referred to anywhere in the project's codebase, despite being used on a daily basis.

We've already discussed tools revolving around package management and the isolation of virtual environments. These are undoubtedly productivity tools as their aim is to simplify and ease the tedious processes of setting up your local working environment. Later in the book, we will discuss more productivity tools that help to solve specific problems, such as profiling and testing. This section is dedicated to other tools that are really worth mentioning but have no specific chapter in this book where they could be introduced.

Custom Python shells

Python programmers spend a lot of time in interactive interpreter sessions. These sessions are very good for testing small code snippets, accessing documentation, or even debugging code at runtime. The default interactive Python session is very simple and does not provide many features, such as tab completion or code introspection helpers. Fortunately, the default Python shell can be easily extended and customized.

If you use an interactive shell very often, you can easily modify the behavior of its prompt. Python at startup reads the `PYTHONSTARTUP` environment variable, looking for the path of the custom initializations script. Some operating system distributions where Python is a common system component (for example, Linux or macOS) may already be preconfigured to provide a default startup script. It is commonly found in the user's home directory under the `.pythonstartup` name.

These scripts often use the `readline` module (based on the GNU `readline` library) together with `rlcompleter` in order to provide interactive tab completion and command history. Both modules are part of the Python standard library.

> The `readline` module is not available on Windows. Windows users often use the `pyreadline` package available on PyPI as a substitution for the missing module.

If you don't have a default Python startup script, you can easily build your own. A basic script for command history and tab completion can be as simple as the following:

```python
# python startup file

import atexit
import os

try:
    import readline
except ImportError:
    print("Completion unavailable: readline module not available")
else:
    import rlcompleter
    # tab completion
    readline.parse_and_bind('tab: complete')

    # Path to history file in user's home directory.
    # Can use your own path.
    history_file = os.path.expanduser('~/.python_shell_history')
    try:
        readline.read_history_file(history_file)
    except IOError:
        pass

    atexit.register(readline.write_history_file, history_file)
    del os, history_file, readline, rlcompleter
```

Create this file in your home directory and call it `.pythonstartup`. Then, add a `PYTHONSTARTUP` variable in your environment using the path of your file.

If you are running Linux or macOS, you can create the Python startup script in your home folder. Then, link it with a `PYTHONSTARTUP` environment variable that's been set in the system shell startup script. For example, the Bash and Korn shells use the `.profile` file, where you can insert a line, as follows:

```
export PYTHONSTARTUP=~/.pythonstartup
```

If you are running Windows, it is easy to set a new environment variable as an administrator in the system preferences and save the script in a common place instead of using a specific user location.

Writing on the `PYTHONSTARTUP` script may be a good exercise but creating a good custom shell all alone is a challenge that few can find time for. Fortunately, there are a few custom Python shell implementations that immensely improve the experience of interactive sessions in Python. In the next section, we will take a closer look at one that is particularly popular—IPython.

Using IPython

IPython provides an extended Python command shell. It is available as a package on PyPI so you can easily install it with either `pip` or `poetry`. Among all the features it provides, some interesting ones are as follows:

- Dynamic object introspection
- System shell access from the prompt
- Multiline code editing
- Syntax highlighting
- Copy-paste helpers
- Direct profiling support
- Debugging facilities

Now, IPython is a part of a larger project called Jupyter, which provides interactive notebooks with live code that can be written in many different languages. Jupyter notebooks are really popular within the data science community where Python really shines. So it is good to know their shell sibling.

The IPython shell is invoked through the `ipython` command. After starting IPython you will immediately notice that the standard Python prompt is replaced by a colorful number of execution cells:

```
$ ipython
Python 3.9.0 (v3.9.0:9cf6752276, Oct  5 2020, 11:29:23)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.19.0 -- An enhanced Interactive Python. Type '?' for help.
In [1]:
```

There are two really handy properties of an IPython shell:

- It allows you to easily work with multiline code including one that has been pasted from the clipboard
- Provides shortcuts for inspecting docstrings, module documentation, and code of imported modules

These two features alone make IPython great for learning purposes. First, if you find any useful snippets of code (including ones in this book), you can easily paste them from system's clipboard and modify them as if the Python interpreter were a code editor. The following is a screenshot of a terminal with an interactive IPython session that the source code of the echo application was pasted into:

```
00:30 $ ipython
Python 3.9.0 (v3.9.0:9cf6752276, Oct  5 2020, 11:29:23)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.19.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from flask import Flask, request
In [1]: from flask import Flask, request
   ...: app = Flask(__name__)
In [1]: from flask import Flask, request
   ...: app = Flask(__name__)
   ...:
   ...:
   ...: @app.route('/')
   ...: def echo():
   ...:     print(request.headers)
   ...:     return (
   ...:         f"METHOD: {request.method}\n"
   ...:         f"HEADERS:\n{request.headers}"
   ...:         f"BODY:\n{request.data.decode()}"
   ...:     )
   ...:
   ...:
   ...: if __name__ == '__main__':
   ...:     app.run(host="0.0.0.0")
   ...:
```

Figure 2.1: Pasting code into IPython

When it comes to code introspection, IPython provides a really quick way of looking into the documentation and source code of imported modules, functions, and classes. Simply type a name you want to inspect and follow it with ? to see the docstring. The following terminal transcript presents an example exploration session of the urlunparse() function from the urllib.parse module:

```
In [1]: urllib.parse.urlunparse?
Signature: urllib.parse.urlunparse(components)
Docstring:
Put a parsed URL back together again.  This may result in a
slightly different, but equivalent URL, if the URL that was parsed
originally had redundant delimiters, e.g. a ? with an empty query
(the draft states that these are equivalent).
File:      /Library/Frameworks/Python.framework/Versions/3.9/lib/
python3.9/urllib/parse.py
Type:      function
```

Use ?? after the function name instead and you'll see the whole source code:

```
In [2]: urllib.parse.urlunparse??
Signature: urllib.parse.urlunparse(components)
Source:
def urlunparse(components):
    """Put a parsed URL back together again.  This may result in a
    slightly different, but equivalent URL, if the URL that was parsed
    originally had redundant delimiters, e.g. a ? with an empty query
    (the draft states that these are equivalent)."""
    scheme, netloc, url, params, query, fragment, _coerce_result = (
                                                  _coerce_
args(*components))
    if params:
        url = "%s;%s" % (url, params)
    return _coerce_result(urlunsplit((scheme, netloc, url, query,
fragment)))
File:      /Library/Frameworks/Python.framework/Versions/3.9/lib/
python3.9/urllib/parse.py
Type:      function
```

> IPython is not the only enhanced Python shell at your disposal. You may want to look at the `btpython` and `ptpython` projects, which have similar capabilities but a slightly different user experience.

Interactive sessions are great for experimentation and module exploration but sometimes can also be useful in final applications. In the next section, you will learn how to embed them inside of your own code.

Incorporating shells in your own scripts and programs

Sometimes, there is a need to incorporate a **read-eval-print loop** (**REPL**), similar to Python's interactive session, inside of your own software. This allows easier experimentation with your code and inspection of its internal state. Sometimes it is simply easier to embed an interactive terminal instead of designing a custom **Command-Line Interface** (**CLI**) for your application (especially if it has to be used on rare occasions). Interactive interpreters are often embedded in web application frameworks to allow developers to interact with data stored within applications using Python REPL instead of database-specific terminal utilities.

The simplest module that allows emulating Python's interactive interpreter already comes with the standard library and is named `code`.

The script that starts interactive sessions consists of one import and a single function call:

```
import code
code.interact()
```

You can easily do some minor tuning, such as modify a prompt value or add banner and exit messages, but anything fancier will require a lot more work. If you want to have more features, such as code highlighting, completion, or direct access to the system shell, it is always better to use something that was already built by someone. Fortunately, the IPython shell mentioned in the previous section can be embedded in your own program as easily as the `code` module.

The following are examples of how to invoke all of the previously mentioned shells inside of your code:

```python
# Example for IPython
import IPython
IPython.embed()

# Example for bpython
import bpython
bpython.embed()

# Example for ptpython
from ptpython.repl import embed
embed(globals(), locals())
```

The first two arguments to the `embed()` function are dictionaries of objects that will be available as global and local namespaces during the interactive session. This can be used to prepopulate the interactive session with modules, variables, functions, or classes that are likely to be used during that session.

Interactive sessions are great for providing a low-level interface of an application directly to the user. Sometimes they can be used to manually inspect the internal state of an application by providing access to either local or global variables. Still, if you want to interactively trace how your application executes the code, you will probably need to use a debugger. Fortunately, Python comes with a built-in debugger that offers such a possibility in the form of an interactive session.

Interactive debuggers

Code debugging is an integral element of the software development process. Many programmers can spend most of their life using only extensive logging and `print()` functions as their primary debugging tools, but most professional developers prefer to rely on some kind of debugger.

Python already ships with a built-in interactive debugger called `pdb`. It can be invoked from the command line on the existing script, so Python will enter post-mortem debugging if the program exits abnormally:

```
$ python3 -m pdb -c continue script.py
```

Another way to achieve similar behavior is running the interpreter with the `-i` flag:

```
$ python3 -i script.py
```

The preceding code will open an interactive session at the moment where Python would normally exit. From there, you can start a post-mortem debugging session by importing the `pdb` module and using the `pdb.pm()` function as in the following example:

```
>>> import pdb
>>> pdb.pm()
```

Post-mortem debugging, while useful, does not cover every scenario. It is useful only when the application exits with some exception if the bug occurs. In many cases, faulty code behaves abnormally but does not exit unexpectedly. In such cases, custom breakpoints can be set on a specific line of code using the `breakpoint()` function. The following is an example of setting a breakpoint inside of a simple function:

```python
import math

def circumference(r: float):
    breakpoint()
    return 2 * math.pi * r
```

> The `breakpoint()` function was not available prior to Python 3.7 so you may see some older Python developers using the following idiom:
>
> ```
> import pdb; pdb.set_trace()
> ```
>
> This will cause the Python interpreter to start the debugger session on this line during runtime.

The `pdb` module is very useful for tracing issues, and at first glance, it may look very similar to the well-known **GNU Debugger** (**GDB**). Because Python is a dynamic language, the `pdb` session is very similar to an ordinary interpreter session. This means that the developer is not limited to tracing code execution but can call any code and even perform module imports.

Sadly, because of its roots (`gdb`), your first experience with `pdb` can be a bit overwhelming due to the existence of cryptic short-letter debugger commands such as `h`, `b`, `s`, `n`, `j`, and `r`. When in doubt, the `help pdb` command, which can be typed during the debugger session, will provide extensive usage information. You can also use the `h` shortcut.

The debugger session in `pdb` is very simple and does not provide additional features such as tab completion or code highlighting. Fortunately, same as with enhanced Python shells, there are a couple of enhanced debugging shells available on PyPI. There is even one based on IPython. Its name is `ipdb`.

If you want to use `ipdb` instead of plain `pdb`, you can either use a modified debugging idiom (`import ipdb; ipdb.set_trace()`) or set the `PYTHONBREAKPOINT` environment variable to the `ipdb.set_trace` value.

Last but not least, many IDEs offer visual debuggers and some developers find them extremely useful. These debuggers allow you to set breakpoints in multiple places of your application without the need for modifying the code with manual `breakpoint()` calls. They also often allow adding **variable watches** that stop program execution when the selected variable has a specific value.

Other productivity tools

We've concentrated so far on the productivity tools that are specific to Python. But the real truth is that programming in different languages is not that different. It doesn't matter what languages programmers use, they often face the same problems and tedious tasks like massaging the data in various formats, downloading network artifacts, searching through filesystems, and navigating projects.

Probably the most flexible productivity tool of all time will be Bash together with common standard utilities found in every POSIX and UNIX-like operating system. Knowing them all thoroughly is probably impossible for an ordinary human. But knowing a few well is something that will make you really productive.

Simply put, sometimes there's no need to write a sophisticated Python script for a one-off job if you can quickly wire and pipe together a few invocations of the `curl`, `grep`, `sed`, and `sort` commands. Sometimes, there is already a specialized tool for a specific and non-trivial job (counting lines of code, for instance) that would take a lot of time to implement from scratch.

The following table gives a short list of such useful utilities that I find invaluable when working with any code. Think of it as a mini awesome list of programming productivity tools:

Utility	Description

`jq`
 	
`https://stedolan.github.io/jq/`
	Utility for manipulating data in the form of JSON documents. Extremely useful for manipulating the output of web APIs directly in the shell. Data is read from standard input and results are printed on standard output. Manipulation is described through a custom domain-specific language that is very easy to learn.

`yq`
 	
`https://pypi.org/project/yq/`
	Sibling of `jq` that uses the same syntax for manipulating YAML documents.

`curl`
 	
`https://curl.se`
	Old-fashioned classic for transferring data through URLs. Most often used for interfacing with HTTP but actually supports over 20 protocols.

HTTPie `https://httpie.io`	Python-based utility for interfacing with HTTP servers. Many developers find it more convenient to use than `curl`.

`autojump`
 	
`https://github.com/wting/autojump`
	Shell utility that allows users to quickly navigate to most recently visited directories. Indispensable for programmers working on dozens of projects in parallel. Simply type `j` and a few characters of the desired directory name and you will probably land in the right place. Plays nicely with Poetry workflows.

`cloc`
 	
`https://github.com/AlDanial/cloc`
	One of the best and most complete utilities for counting lines of code. Sometimes you need to see how big a project is and how many programming or markup languages it uses. `cloc` will give you the right answer quickly.

`ack-grep`
 	
`https://beyondgrep.com`
	`grep` on steroids. Allows you to quickly search through large codebases looking for a specific string. Allows filtering by programming language and often is simply faster and better than opening a project in an IDE.

GNU parallel `https://www.gnu.org/software/parallel/`	Enhanced replacement of `xargs`. Really invaluable if you want to do many things in parallel inside of in a shell or Bash script, especially if you want to do it reliably and efficiently.

Summary

This chapter was all about development environments for Python programmers. We've discussed the importance of environment isolation for Python projects. You've learned two different levels of environment isolation (application-level and system-level), and multiple tools that allow you to create them in a consistent and repeatable manner. We've also discussed some essential topics for managing Python dependencies in your projects. This chapter ended with a review of a few tools that improve the ways in which you can experiment with Python or debug your programs and work effectively.

Once you have all of these tools in your tool belt, you are well-prepared for the next few chapters, where we will discuss multiple features of modern Python syntax. You're probably already hungry for Python code so we will start with a quick overview of the new things that were included in Python over the last few releases.

If you're quite up to date with what's happening in Python, you can probably skip the next chapter. Still, take a quick look at the headings—it is possible that you have missed something, as Python evolves really fast.

3
New Things in Python

One of the most important steps in the history of Python was probably the release of Python 3.0. The most notable changes that happened in that release were:

- Resolving multiple issues regarding text, data, and Unicode handling
- Getting rid of old-style classes
- Starting standard library reorganizations
- Introducing function annotations
- Introducing new syntax for exception handling

As we know from *Chapter 1*, *Current Status of Python*, Python 3 isn't backward-incompatible with Python 2. This is the main reason why it took so many years for the Python community to fully embrace it. That was a tough, albeit necessary, lesson for Python core developers and the Python community.

Fortunately, problems associated with the adoption of Python 3 didn't stop the process of language evolution. Since December 3, 2008 (the official release of Python 3.0), we've seen a stable inflow of new major Python updates. Every new release brought new improvements to the language, its standard library, and its interpreter. Moreover, beginning with version 3.9, Python has adopted an annual release cycle. This means we will have access to new features and improvements every year.

> If you want to learn more about the Python release cycle, read the *PEP 602 – Annual Release Cycle for Python* document, available at `https://www.python.org/dev/peps/pep-0602/`.

In this chapter, we will take a closer look at the recent Python evolution. We will review a number of important additions across the latest few releases. We will also take a speculative look into the future and present a few features that have been accepted in the PEP process and will become an official part of the Python programming language in the very near future. Along the way, we'll cover the following topics:

- Recent language additions
- Not that new, but still shiny
- What may come in the future?

But before we review those features, let's begin by considering the technical requirements.

Technical requirements

The following are the Python packages that are mentioned in this chapter that you can download from PyPI:

- `mypy`
- `pyright`

Information on how to install packages is included in *Chapter 2*, *Modern Python Development Environments*.

The code files for this chapter can be found at `https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition/tree/main/Chapter%203`.

Recent language additions

Every release of Python comes with it a lot of changes of different types. Almost every release of Python brings some new syntax elements. However, the majority of the changes are related to Python's standard library, the CPython interpreter, the Python API, and CPython's C API. Due to space limitations, it is impossible to cover all of these in this book. That is why we will focus just on new syntax features and new additions to the standard library.

In terms of the two latest versions of Python, we can distinguish four main syntax updates:

- Dictionary and merge update operators (added in Python 3.9)
- Assignment expressions (added in Python 3.8)

- Type hinting generics (added in Python 3.9)
- Positional-only arguments (added in Python 3.8)

These four features would best be described as quality-of-life improvements. They do not introduce any new programming paradigms, nor drastically change the way your code can be written. They simply allow for better coding patterns or enable stricter API definition.

In recent years, Python core developers have been primarily focused on removing dead or redundant modules from the standard library rather than adding anything new. Still, from time to time, we see some standard library additions. In the last two releases, we have been the beneficiaries of two completely new modules:

- The `zoneinfo` module for supporting the **IANA** (**Internet Assigned Numbers Authority**) time zone database (added in Python 3.9)
- The `graphlib` module for operating with graph-like structures (added in Python 3.8)

Both modules are fairly small with regards to their API size. Later, we will discuss some example areas where you could apply them. But first, let's zoom into the syntax updates incorporated in Python 3.8 and Python 3.9.

Dictionary merge and update operators

Python allows the use of a number of selected arithmetic operators to manipulate the built-in container types, including lists, tuples, sets, and dictionaries.

For lists and tuples, you can use the + (addition) operator to concatenate two variables as long as they are the same type. There is also the += operator, which allows for the in-place modification of existing variables. The following transcript presents examples of the concatenation of lists and tuples in an interactive session:

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> (1, 2, 3) + (4, 5, 6)
(1, 2, 3, 4, 5, 6)
>>> value = [1, 2, 3]
>>> value += [4, 5, 6]
>>> value
[1, 2, 3, 4, 5, 6]
>>> value = (1, 2, 3)
>>> value += (4, 5, 6)
>>> value
(1, 2, 3, 4, 5, 6)
```

When it comes to sets, there are exactly four binary operators (having two operands) that produce a new set:

- **Intersection operator**: Represented by & (bitwise OR). This produces a set with elements common to both sets.

- **Union operator**: Represented by | (bitwise OR). This produces a set of all elements in both sets.

- **Difference operator**: Represented by - (subtraction). This produces a set with elements in the left-hand set that are not in the right-hand set.

- **Symmetric difference**: Represented by ^ (bitwise XOR). This produces a set with elements of both sets that are in either of the sets but not both.

The following transcript presents examples of intersection and union operations on sets in an interactive session:

```
>>> {1, 2, 3} & {1, 4}
{1}
>>> {1, 2, 3} | {1, 4}
{1, 2, 3, 4}
>>> {1, 2, 3} - {1, 4}
{2, 3}
>>> {1, 2, 3} ^ {1, 4}
{2, 3, 4}
```

For a very long time, Python didn't have a dedicated binary operator that would permit the production of a new dictionary from two existing dictionaries. Starting with Python 3.9, we can use the | (bitwise OR) and |= (in-place bitwise OR) operators to perform a dictionary merge and update operations on dictionaries. That should be the idiomatic way of producing a union of two dictionaries. The reasoning behind adding new operators was outlined in the *PEP 584 – Add Union Operators To Dict* document.

> **A programming idiom** is the common and most preferable way of performing specific tasks in a given programming language. Writing idiomatic code is an important part of Python culture. The Zen of Python says: "*There should be one – and preferably only one – obvious way to do it.*"
>
> We will discuss more idioms in *Chapter 4, Python in Comparison with Other Languages*.

In order to merge two dictionaries into a new dictionary, use the following expression:

```
dictionary_1 | dictionary_2
```

The resulting dictionary will be a completely new object that will have all the keys of both source dictionaries. If both dictionaries have overlapping keys, the resulting object will receive values from the rightmost object.

Following is an example of using this syntax on two dictionary literals, where the dictionary on the left is updated with values from the dictionary on the right:

```
>>> {'a': 1} | {'a': 3, 'b': 2}
{'a': 3, 'b': 2}
```

If you prefer to update the dictionary variable with the keys coming from a different dictionary, you can use the following in-place operator:

```
existing_dictionary |= other_dictionary
```

The following is an example of usage with a real variable:

```
>>> mydict = {'a': 1}
>>> mydict |= {'a': 3, 'b': 2}
>>> mydict
{'a': 3, 'b': 2}
```

In older versions of Python, the simplest way to update an existing dictionary with the contents of another dictionary was to use the `update()` method, as in the following example:

```
existing_dictionary.update(other_dictionary)
```

This method modifies `existing_dictionary` in place and returns no value. This means that it does not allow the straightforward production of a merged dictionary as an expression and is always used as a statement.

> The difference between **expressions** and **statements** will be explained in the *Assignment expressions* section.

Alternative – Dictionary unpacking

It is a little-known fact that Python already supported a fairly concise way to merge two dictionaries before version 3.9 through a feature known as **dictionary unpacking**. Support for dictionary unpacking in `dict` literals was introduced in Python 3.5 with *PEP 448 Additional Unpacking Generalizations*. The syntax for unpacking two (or more) dictionaries into a new object is as follows:

```
{**dictionary_1, **dictionary_2}
```

The example involving real literals is as follows:

```
>>> a = {'a': 1}; b = {'a':3, 'b': 2}
>>> {**a, **b}
{'a': 3, 'b': 2}
```

This feature, together with list unpacking (with `*value` syntax), may be familiar for those who have experience of writing functions that can accept an undefined set of arguments and keyword arguments, also known as **variadic functions**. This is especially useful when writing decorators.

> We will discuss the topic of variadic functions and decorators in detail in *Chapter 4*, *Python in Comparison with Other Languages*.

You should remember that dictionary unpacking, while extremely popular in function definitions, is an especially rare method of merging dictionaries. It may confuse less experienced programmers who are reading your code. That is why you should prefer the new merge operator over dictionary unpacking in code that runs in Python 3.9 and newer versions. For older versions of Python, it is sometimes better to use a temporary dictionary and a simple `update()` method.

Alternative – ChainMap from the collections module

Yet another way to create an object that is, functionally speaking, a merge of two dictionaries is through the `ChainMap` class from the `collections` module. This is a wrapper class that takes multiple mapping objects (dictionaries in this instance) and acts as if it was a single mapping object.

The syntax for merging two dictionaries with `ChainMap` is as follows:

```
new_map = ChainMap(dictionary_2, dictionary_1)
```

Note that the order of dictionaries is reversed compared to the | operator. This means that if you try to access a specific key of the new_map object, it will perform lookups over wrapped objects in a left-to-right order. Consider the following transcript, which illustrates examples of operations using the ChainMap class:

```
>>> from collections import ChainMap
>>> user_account = {"iban": "GB71BARC20031885581746", "type":
"account"}
>>> user_profile = {"display_name": "John Doe", "type": "profile"}
>>> user = ChainMap(user_account, user_profile)
>>> user["iban"]
'GB71BARC20031885581746'
>>> user["display_name"]
'John Doe'
>>> user["type"]
'account'
```

In the preceding example, we can clearly see that the resulting user object of the ChainMap type contains keys from both the user_account and user_profile dictionaries. If any of the keys overlap, the ChainMap instance will return the value of the leftmost mapping that has the specific key. That is the complete opposite of the dictionary merge operator.

ChainMap is a wrapper object. This means that it doesn't copy the contents of the source mappings provided, but stores them as a reference. This means that if underlying objects change, ChainMap will be able to return modified data. Consider the following continuation of the previous interactive session:

```
>>> user["display_name"]
'John Doe'
>>> user_profile["display_name"] = "Abraham Lincoln"
>>> user["display_name"]
'Abraham Lincoln'
```

Moreover, ChainMap is writable and populates changes back to the underlying mapping. What you need to remember is that writes, updates, and deletes only affect the leftmost mapping. If used without proper care, this can lead to some confusing situations, as in the following continuation of the previous session:

```
>>> user["display_name"] = "John Doe"
>>> user["age"] = 33
>>> user["type"] = "extension"
>>> user_profile
{'display_name': 'Abraham Lincoln', 'type': 'profile'}
```

```
>>> user_account
{'iban': 'GB71BARC20031885581746', 'type': 'extension', 'display_name':
'John Doe', 'age': 33}
```

In the preceding example, we can see that the `'display_name'` key was populated back to the `user_account` dictionary, where `user_profile` was the initial source dictionary holding such a key. In many contexts, such backpropagating behavior of `ChainMap` is undesirable. That's why the common idiom for using it for the purpose of merging two dictionaries actually involves explicit conversion to a new dictionary. The following is an example that uses previously defined input dictionaries:

```
>>> dict(ChainMap(user_account, user_profile))
{'display_name': 'John Doe', 'type': 'account', 'iban':
'GB71BARC20031885581746'}
```

If you want to simply merge two dictionaries, you should prefer a new merge operator over `ChainMap`. However, this doesn't mean that `ChainMap` is completely useless. If the back and forth propagation of changes is your desired behavior, `ChainMap` will be the class to use. Also, `ChainMap` works with any mapping type. So, if you need to provide unified access to multiple objects that act as if they were dictionaries, `ChainMap` will enable the provision of a single merge-like unit to do so.

> If you have a custom dict-like class, you can always extend it with the special `__or__()` method to provide compatibility with the | operator instead of using `ChainMap`. Overriding special methods will be covered in *Chapter 4*, *Python in Comparison with Other Languages*. Anyway, using `ChainMap` is usually easier than writing a custom `__or__()` method and will allow you to work with pre-existing object instances of classes that you cannot modify.

Usually, the most important reason for using `ChainMap` over dictionary unpacking or the union operator is backward compatibility. On Python versions older than 3.9, you won't be able to use the new dictionary merge operator syntax. So, if you have to write code for older versions of Python, use `ChainMap`. If you don't, it is better to use the merge operator.

Another syntax change that has a big impact on backward compatibility is assignment expressions.

Assignment expressions

Assignment expressions are a fairly interesting feature because their introduction affected the fundamental part of Python syntax: the distinction between expressions and statements. Expressions and statements are the key building blocks of almost every programming language. The difference between them is really simple: expressions have a value, while statements do not.

Think of **statements** as consecutive actions or instructions that your program executes. So, value assignments, `if` clauses, together with `for` and `while` loops, are all statements. Function and class definitions are statements, too.

Think of **expressions** as anything that can be put into an `if` clause. Typical examples of expressions are literals, values returned by operators (excluding in-place operators), and comprehensions, such as list, dictionary, and set comprehensions. Function calls and method calls are expressions, too.

There are some elements of the many programming languages that are often inseparably bound to statements. These are often:

- Functions and class definitions
- Loops
- `if...else` clauses
- Variable assignments

Python was able to break that barrier by providing syntax features that were expression counterparts of such language elements, namely:

- Lambda expressions for anonymous functions as a counterpart for function definitions:

  ```
  lambda x: x**2
  ```

- Type object instantiation as a counterpart for class definition:

  ```
  type("MyClass", (), {})
  ```

- Various comprehensions as a counterpart for loops:

  ```
  squares_of_2 = [x**2 for x in range(10)]
  ```

- Compound expressions as a counterpart for `if … else` statements:

  ```
  "odd" if number % 2 else "even"
  ```

For many years, however, we haven't had access to syntax that would convey the semantics of assigning a value to a variable in the form of an expression, and this was undoubtedly a conscious design choice on the part of Python's creators. In languages such as C, where variable assignment can be used both as an expression and as a statement, this often leads to situations where the assignment operator is confused by the equality comparison. Anyone who has programmed in C can attest to the fact that this is a really annoying source of errors. Consider the following example of C code:

```c
int err = 0;
if (err = 1) {
    printf("Error occured");
}
```

And compare it with the following:

```c
int err = 0;
if (err == 1) {
    printf("Error occured");
}
```

Both are syntactically valid in C because `err = 1` is an expression in C that will evaluate to the value 1. Compare this with Python, where the following code will result in a syntax error:

```python
err = 0
if err = 1:
    printf("Error occured")
```

On rare occasions, however, it may be really handy to have a variable assignment operation that would evaluate to a value. Luckily, Python 3.8 introduced the dedicated `:=` operator, which assigns a value to the variable but acts as an expression instead of a statement. Due to its visual appearance, it was quickly nicknamed the **walrus** operator.

The use cases for this operator are, quite frankly, limited. They help to make code more concise. And often, more concise code is easier to understand because it improves the signal-to-noise ratio. The most common scenario for the walrus operator is when a complex value needs to be evaluated and then immediately used in the statements that follow.

A commonly referenced example is working with regular expressions. Let's imagine a simple application that reads source code written in Python and scans it with regular expressions looking for imported modules.

Without the use of assignment expressions, the code could appear as follows:

```python
import os
import re
import sys

import_re = re.compile(
    r"^\s*import\s+\.{0,2}((\w+\.)*(\w+))\s*$"
)
import_from_re = re.compile(
    r"^\s*from\s+\.{0,2}((\w+\.)*(\w+))\s+import\s+(\w+|\*)+\s*$"
)

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print(f"usage: {os.path.basename(__file__)} file-name")
        sys.exit(1)

    with open(sys.argv[1]) as file:
        for line in file:
            match = import_re.search(line)
            if match:
                print(match.groups()[0])

            match = import_from_re.search(line)
            if match:
                print(match.groups()[0])
```

As you can observe, we had to repeat twice the pattern that evaluates the match of complex expressions and then retrieves grouped tokens. That block of code could be rewritten with assignment expressions in the following way:

```python
if match := import_re.match(line):
    print(match.groups()[0])

if match := import_from_re.match(line):
    print(match.groups()[0])
```

As you can see, there is a small improvement in terms of readability, but it isn't dramatic. This type of change really shines in situations where you need to repeat the same pattern multiple times. The continuous assignment of temporary results to the same variable can make code look unnecessarily bloated.

Another use case could be reusing the same data in multiple places in larger expressions. Consider the example of a dictionary literal that represents some predefined data of an imaginary user:

```python
first_name = "John"
last_name = "Doe"
height = 168
weight = 70

user = {
    "first_name": first_name,
    "last_name": last_name,
    "display_name": f"{first_name} {last_name}",
    "height":  height,
    "weight": weight,
    "bmi": weight / (height / 100) ** 2,
}
```

Let's assume that in our situation, it is important to keep all the elements consistent. Hence, the display name should always consist of a first name and a last name, and the BMI should be calculated on the basis of weight and height. In order to prevent us from making a mistake when editing specific data components, we had to define them as separate variables. These are no longer required once a dictionary has been created. Assignment expressions enable the preceding dictionary to be written in a more concise way:

```python
user = {
    "first_name": (first_name := "John"),
    "last_name": (last_name := "Doe"),
    "display_name": f"{first_name} {last_name}",
    "height": (height := 168),
    "weight": (weight := 70),
    "bmi": weight / (height / 100) ** 2,
}
```

As you can see, we had to wrap assignment expressions with parentheses. Unfortunately, the := syntax clashes with the : character used as an association operator in dictionary literals and parentheses are a way around that.

Assignment expressions are a tool for polishing your code and nothing more. Always make sure that once applied, they actually improve readability, instead of making it more obscure.

Type-hinting generics

Type-hinting annotations, although completely optional, are an increasingly popular feature of Python. They allow you to annotate variable, argument, and function return types with type definitions. These type annotations serve documentational purposes, but can also be used to validate your code using external tools. Many programming IDEs are able to understand typing annotations and visually highlight potential typing problems. There are also static type checkers, such as **mypy** or **pyright**, that can be used to scan through the whole code base and report all typing errors of code units that use annotations.

> The story of the mypy project is very interesting. It began life as the Ph.D. research of Jukka Lehtosalo, but it really started to take shape when he started working on it together with Guido van Rossum (Python creator) at Dropbox. You can learn more about that story from the farewell letter to Guido on Dropbox's tech blog at `https://blog.dropbox.com/topics/company/thank-you--guido`.

In its simplest form, type hinting can be used with a conjunction of the built-in or custom types to specify desired types, function input arguments, and return values, as well as local variables. Consider the following function, which allows the performance of the case-insensitive lookup of keys in a string-keyed dictionary:

```python
from typing import Any

def get_ci(d: dict, key: str) -> Any:
    for k, v in d.items():
        if key.lower() == k.lower():
            return v
```

> The preceding example is, of course, a naïve implementation of a case-sensitive lookup. If you would like to do this in a more performant way, you would probably require a dedicated class. We will eventually revisit this problem later in the book.

The first statement imports from the `typing` module the `Any` type, which defines that the variable or argument can be of any type. The signature of our function specifies that the first argument, `d`, should be a dictionary, while the second argument, `key`, should be a string. The signature ends with the specification of a return value, which can be of any type.

If you're using type checking tools, the preceding annotations will be sufficient to detect many mistakes. If, for instance, a caller switches the order of positional arguments, you will be able to detect the error quickly, as the key and d arguments are annotated with different types. However, these tools will not complain in a situation where a user passes a dictionary that uses different types for keys.

For that very reason, generic types such as `tuple`, `list`, `dict`, `set`, `frozenset`, and many more can be further annotated with types of their content. For a dictionary, the annotation has the following form:

```
dict[KeyType, ValueType]
```

The signature of the `get_ci()` function, with more restrictive type annotations, would be as follows:

```
def get_ci(d: dict[str, Any], key: str) -> Any: ...
```

In older versions of Python, built-in collection types could not be annotated so easily with types of their content. The `typing` module provides special types that can be used for that purpose. These types include:

- `typing.Dict` for dictionaries
- `typing.List` for lists
- `typing.Tuple` for tuples
- `typing.Set` for sets
- `typing.FrozenSet` for frozen sets

These types are still useful if you need to provide functionality for a wide spectrum of Python versions, but if you're writing code for Python 3.9 and newer releases only, you should use the built-in generics instead. Importing those types from typing modules is deprecated and they will be removed from Python in the future.

> We will take a closer look at typing annotations in *Chapter 4*, *Python in Comparison with Other Languages*.

Positional-only parameters

Python is quite flexible when it comes to passing arguments to functions. There are two ways in which function arguments can be provided to functions:

- As a **positional argument**
- As a **keyword argument**

For many functions, it is the choice of the caller in terms of how arguments are passed. This is a good thing because the user of the function can decide that a specific usage is more readable or convenient in a given situation. Consider the following example of a function that concatenates the strings using a delimiter:

```python
def concatenate(first: str, second: str, delim: str):
    return delim.join([first, second])
```

There are multiple ways in terms of how this function can be called:

- With positional arguments: `concatenate("John", "Doe", " ")`
- With keyword arguments: `concatenate(first="John", second="Doe", delim=" ")`
- With a mix of positional and keyword arguments: `concatenate("John", "Doe", delim=" ")`

If you are writing a reusable library, you may already know how your library is intended to be used. Sometimes, you may know from your experience that specific usage patterns will make the resulting code more readable, or quite the opposite. You may not be certain about your design yet and want to make sure that the API of your library may be changed within a reasonable time frame without affecting your users. In either case, it is a good practice to create function signatures in a way that supports the intended usage and also allows for future extension.

Once you publish your library, the function signature forms a usage contract with your library. Any change to the argument names and their ordering can break applications of the programmer using that library.

If you were to realize at some point in time that the argument names `first` and `second` don't properly explain their purpose, you cannot change them without breaking backward compatibility. That's because there may be a programmer who used the following call:

```python
concatenate(first="John", second="Doe", delim=" ")
```

If you want to convert the function into a form that accepts any number of strings, you can't do that without breaking backward compatibility because there might be a programmer who used the following call:

```python
concatenate("John", "Doe", " ")
```

Fortunately, Python 3.8 added the option to define specific arguments as positional-only. This way, you may denote which arguments cannot be passed as keyword arguments in order to avoid issues with backward compatibility in the future. You can also denote specific arguments as keyword-only. Careful consideration as to which arguments should be passed as position-only and which as keyword-only serves to make the definition of functions more susceptible to future changes. Our `concatenate()` function, defined with the use of positional-only and keyword-only arguments, could look as follows:

```python
def concatenate(first: str, second: str, /, *, delim: str):
    return delim.join([first, second])
```

The way in which you read this definition is as follows:

- All arguments preceding the / mark are positional-only arguments
- All arguments following the * mark are keyword-only arguments

The preceding definition ensures that the only valid call to the `concatenate()` function would be in the following form:

```python
concatenate("John", "Doe", delim=" ")
```

And if you were to try to call it differently, you would receive a `TypeError` error, as in the following example:

```python
>>> concatenate("John", "Doe", " ")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: concatenate() takes 2 positional arguments but 3 were given
```

Let's assume that we've published our function in a library in the last format and now we want to make it accept an unlimited number of positional arguments. As there is only one way in which this function can be used, we can now use argument unpacking to implement the following change:

```python
def concatenate(*items, delim: str):
    return delim.join(items)
```

The `*items` argument will capture all the positional arguments in the `items` tuple. Thanks to such changes, users will be able to use the function with a variable number of positional items, as in the following examples:

```
>>> concatenate("John", "Doe", delim=" ")
'John Doe'
>>> concatenate("Ronald", "Reuel", "Tolkien", delim=" ")
'Ronald Reuel Tolkien'
>>> concatenate("Jay", delim=" ")
'Jay'
>>> concatenate(delim=" ")
''
```

Positional-only and keyword-only arguments are a great tool for library creators as they create some space for future design changes that won't affect their users. But they are also a great tool for writing applications, especially if you work with other programmers. You can utilize positional-only and keyword-only arguments to make sure that functions will be invoked as intended. This may help in future code refactoring.

zoneinfo module

Handling time and time zones is one of the most challenging aspects of programming. The main reasons are numerous common misconceptions that programmers have about time and time zones. Another reason is the never-ending stream of updates to actual time zone definitions. And these changes happen every year, often for political reasons.

Python, starting from version 3.9, makes access to the information regarding current and historical time zones easier than ever. The Python standard library provides a zoneinfo module that is an interface to the time zone database either provided by your operating system or obtained as a first-party tzdata package from PyPI.

> Packages from PyPI are considered third-party packages, while standard library modules are considered first-party packages. tzdata is quite unique because it is maintained by CPython's core developers. The reason for extracting the contents of the IANA database to separate packages on PyPI is to ensure regular updates that are independent from CPython's release cadence.

Actual usage involves creating ZoneInfo objects using the following constructor call:

```
ZoneInfo(timezone_key)
```

Here, `timezone_key` is a filename from IANA's time zone database. These filenames resemble the way in which time zones are often presented in various applications. Examples include:

- `Europe/Warsaw`
- `Asia/Tel_Aviv`
- `America/Fort_Nelson`
- `GMT-0`

Instances of the `ZoneInfo` class can be used as a `tzinfo` parameter of the `datetime` object constructor, as in the following example:

```python
from datetime import datetime
from zoneinfo import ZoneInfo

dt = datetime(2020, 11, 28, tzinfo=ZoneInfo("Europe/Warsaw"))
```

This allows you to create so-called time zone-aware datetime objects. Time zone-aware datetime objects are essential in properly calculating the time differences in specific time zones because they are able to take into account things such as changes between standard and daylight-saving time, together with any historical changes made to IANA's time zone database.

You can obtain a full list of all the time zones available in your system using the `zoneinfo.available_timezones()` function.

graphlib module

Another interesting addition to the Python standard library is the `graphlib` module, added in Python 3.9. This is a module that provides utilities for working with graph-like data structures.

A **graph** is a data structure consisting of nodes connected by edges. Graphs are a concept from the field of mathematics known as **graph theory**. Depending on the edge type, we can distinguish between two main types of graphs:

- An **undirected graph** is a graph where every edge is undirected. If a graph was a system of cities connected by roads, the edges in an undirected graph would be two-way roads that can be traversed from either side. So, if two nodes, *A* and *B*, are connected to edge *E* in an undirected graph, you can traverse from *A* to *B* and from *B* to *A* using the same edge, *E*.

- A **directed graph** is a graph where every edge is directed. Again, if a graph was a system of cities connected by roads, the edges in a directed graph would be a single-way road that can be traversed from a single point of origin only. If two nodes, *A* and *B*, are connected to a single edge, *E*, that starts from node *A*, you can traverse from *A* to *B* using that edge, but can't traverse from *B* to *A*.

Moreover, graphs can be either cyclic or acyclic. A **cyclic graph** is a graph that has at least one cycle—a closed path that starts and ends at the same node. An **acyclic graph** is a graph that does not have any cycles. *Figure 3.1* presents example representations of directed and undirected graphs:
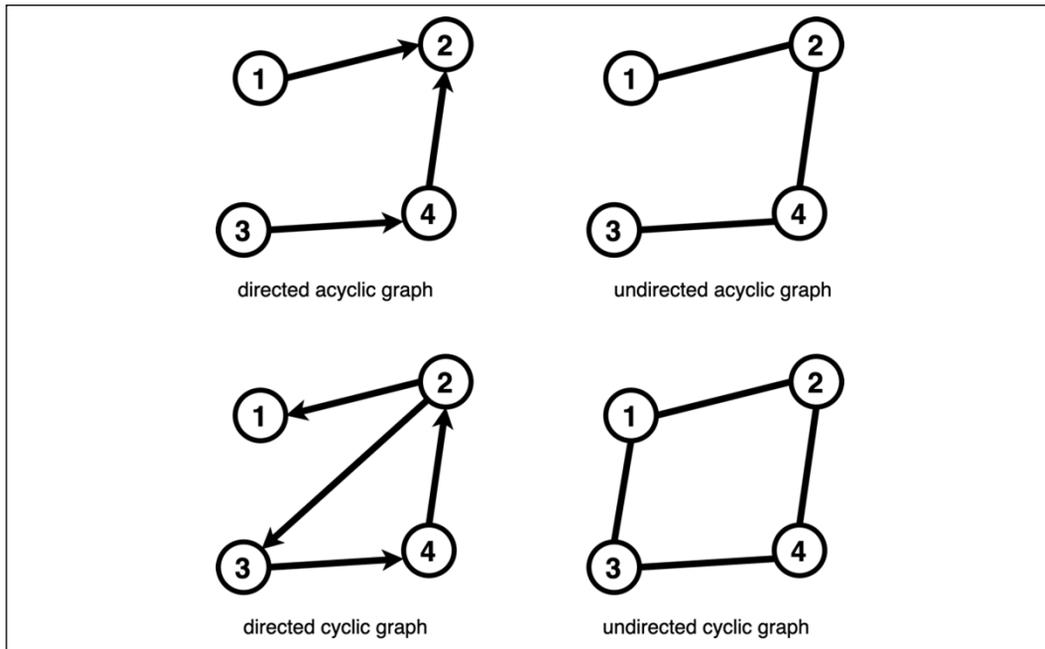
Figure 3.1: Visual representations of various graph types

Graph theory deals with many mathematical problems that can be modeled using graph structures. In programming, graphs are used to solve many algorithmic problems. In computer science, graphs can be used to represent the flow of data or relationships between objects. This has many practical applications, including:

- Modeling dependency trees
- Representing knowledge in a machine-readable format
- Visualizing information
- Modeling transportation systems

The `graphlib` module is supposed to aid Python programmers when working with graphs. This is a new module, so it currently only includes a single utility class named `TopologicalSorter`. As the name suggests, this class is able to perform a topological sort of directed acyclic graphs.

Topological sorting is the operation of ordering nodes of a **Directed Acyclic Graph (DAG)** in a specific way. The result of topological sorting is a list of all nodes where every node appears before all the nodes that you can traverse to from that node, in other words:

- The first node will be the node that cannot be traversed to from any other node
- Every next node will be a node from which you cannot traverse to previous nodes
- The last node will be a node from which you cannot traverse to any node

Some graphs may have multiple orderings that satisfy the requirements of topological sorting. *Figure 3.2* presents an example DAG with three possible topological orderings:
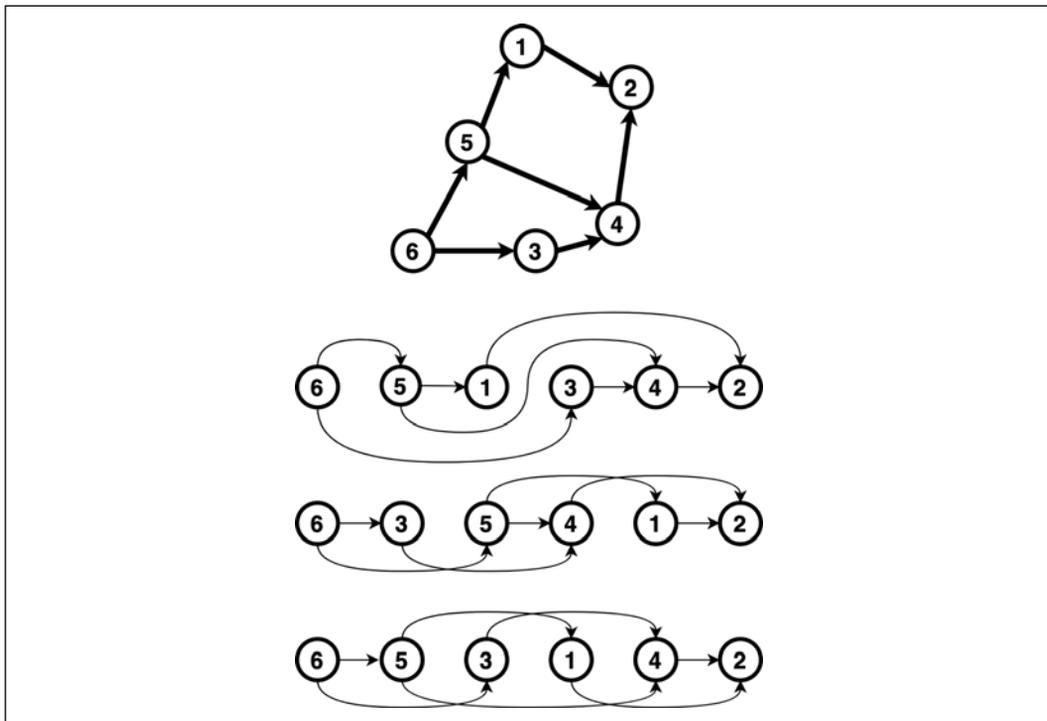
Figure 3.2: Various ways to sort the same graph topologically

To better understand the use of topological sorting, let's consider the following problem. We have a complex operation to execute that consists of multiple dependent tasks. This job could be, for instance, migrating multiple database tables between two different database systems. This is a well-known problem, and there are already multiple tools that can migrate data between various database management systems. But for the sake of illustration, let's assume that we don't have such a system and need to build something from scratch.

In relational database systems, rows in tables are often cross-referenced, and the integrity of those references is guarded by **foreign key constraints**. If we would like to ensure that, at any given point in time, the target database is referentially integral, we would have to migrate our all the tables in specific order. Let's assume we have the following database tables:

- A `customers` table, which holds personal information pertaining to customers.

- An `accounts` table, which holds information about user accounts, including their balances. A single user can have multiple accounts (for instance, personal and business accounts), and the same account cannot be accessed by multiple users.

- A `products` table, which holds information on the products available for sale in our system.

- An `orders` table, which holds individual orders of multiple products within a single account made by a single user.

- An `order_products` table, which holds information regarding the quantities of individual products within a single order.

Python does not have any special data type dedicated to represent graphs. But it has a dictionary type that is great at mapping relationships between keys and values. Let's define references between our imaginary tables:

```python
table_references = {
    "customers": set(),
    "accounts": {"customers"},
    "products": set(),
    "orders": {"accounts", "customers"},
    "order_products": {"orders", "products"},
}
```

If our reference graph does not have cycles, we can topologically sort it. The result of that sorting would be a possible table migration order. The constructor of the `graphlib.TopologicalSorter` class accepts as input a single dictionary in which keys are origin nodes and values are sets of destination nodes. This means that we can pass our `table_references` variable directly to the `TopologicalSorter()` constructor. To perform a topological sort, we can use the `static_order()` call, as in the following transcript from an interactive session:

```python
>>> from graphlib import TopologicalSorter
>>> table_references = {
...     "customers": set(),
...     "accounts": {"customers"},
...     "products": set(),
...     "orders": {"accounts", "customers"},
...     "order_products": {"orders", "products"},
... }
>>> sorter = TopologicalSorter(table_references)
>>> list(sorter.static_order())
['customers', 'products', 'accounts', 'orders', 'order_products']
```

Topological sorting can be performed only on DAGs. `TopologicalSorter` doesn't check for the existence of cycles during initialization, although it will detect cycles during sorting. If a cycle is found, the `static_order()` method will raise a `graphlib.CycleError` exception.

> Our example was, of course, straightforward and fairly easy to solve by hand. However, real databases often consist of dozens or even hundreds of tables. Preparing such a plan manually for databases that big would be a very tedious and error-prone task.

The features we've reviewed so far are quite new, so it will take some time until they become the mainstream elements of Python. That's because they are not backward compatible, and older versions of Python are still supported by many library maintainers.

In the next section, we will review a number of important Python elements introduced in Python 3.6 and Python 3.7, so we will definitely have wider Python version coverage. Not all of these new elements are popular though, so I hope you will still learn something.

Not that new, but still shiny

Every Python release brings something new. Some changes are real revelations; they greatly improve the way we can program and are adopted almost instantly by the community. The benefits of other changes, however, may not be obvious at the beginning and they may require a little more time to really take off.

We've seen this happening with function annotations that were part of Python from the very first 3.0 release. It took years to build an ecosystem of tools that would leverage them. Now, annotations seem almost ubiquitous in modern Python applications.

The core Python developers are very conservative about adding new modules to the standard library and we rarely see new additions. Still, chances are that you will soon forget about using the `graphlib` or `zoneinfo` modules if you don't have the opportunity to work with problems that require manipulating graph-like data structures or the careful handling of time zones. You may have already forgotten about other nice additions to Python that have happened over the past few years. That's why we will do a brief review of a few important changes that happened in versions older than Python 3.7. These will either be small but interesting additions that could easily be missed, or things that simply take time to get used to.

breakpoint() function

We discussed the topic of debuggers in *Chapter 2*, *Modern Python Development Environments*. The `breakpoint()` function was already mentioned there as an idiomatic way of invoking the Python debugger.

It was added in Python 3.7, so has already been available for quite some time. Still, it is one of those changes that simply takes some effort to get used to. We've been told and taught for many years that the simplest way to invoke the debugger from Python code is via the following snippet:

```
import pdb; pdb.set_trace()
```

It doesn't look pretty, nor does it look straightforward but, if you've been doing that every day for years, as many programmers have, you would have that in your muscle memory. Problem? Jump to the code, input a few keystrokes to invoke `pdb`, and then restart the program. Now you're in the interpreter shell at the very same spot as your error occurs. Done? Go back to the code, remove `import pdb; pdb.set_trace()`, and then start working on your fix.

So why should you bother? Isn't that something of a personal preference? Are breakpoints something that ever get to production code?

The truth is that debugging is often a solitary and deeply personal task. We often spend numerous hours struggling with bugs, looking for clues, and reading code over and over in a desperate attempt to locate that small mistake that is breaking our application. When you're deeply focused on finding the cause of a problem, you should definitely use something that you find the most convenient. Some programmers prefer debuggers integrated into IDEs. Some programmers don't even use debuggers, preferring elaborated `print()` calls spread all over the code instead. Always choose whatever you find the most convenient.

But if you're used to a plain old shell-based debugger, the `breakpoint()` can make your work easier. The main advantage of this function is that it isn't bound to a single debugger. By default, it invokes a `pdb` session, but this behavior can be modified with a `PYTHONBREAKPOINT` environment variable. If you prefer to use an alternative debugger (such as `ipdb`, as mentioned *Chapter 2*, *Modern Python Development Environments*), you can set this environment variable to a value that will tell Python which function to invoke.

Standard practice is to set your preferred debugger in a shell profile script so that you don't have to modify this variable in every shell session. For instance, if you're a Bash user and want to always use `ipdb` instead of `pdb`, you could insert the following statement in your `.bash_profile` file:

```
PYTHONBREAKPOINT=ipdb.set_trace()
```

This approach also works well when working together. For instance, if someone asks for your help with debugging, you can ask them to insert breakpoint statements in suspicious places. That way, when you run the code on your own computer, you will be using the debugger of your choice.

> If you don't know where to put your breakpoint, but the application exits upon an unhandled exception, you can use the postmortem feature of `pdb`. With the following command, you can start your Python script in a debugging session that will pause at the moment the exception was raised:
>
> ```
> python3 -m pdb -c continue script.py
> ```

Development mode

Since version 3.7, the Python interpreter can be invoked in dedicated development mode, which introduces additional runtime checks. These are helpful in diagnosing potential issues that may arise when running the code. In correctly working code, those checks would be unnecessarily expensive, so they are disabled by default.

Development mode can be enabled in two ways:

- Using the `-X dev` command-line option of the Python interpreter, for instance:

```
python -X dev my_application.py
```

- Using the `PYTHONDEVMODE` environment variable, for instance:

```
PYTHONDEVMODE=1 my_application
```

The most important effects that this mode enables are as follows:

- Memory allocation hooks: buffer under/overflow, violations of the memory allocator API, unsafe usage of the **Global Interpreter Lock** (**GIL**)
- Import warnings issued in relation to possible mistakes when importing modules
- Resource warnings issued in the case of improper handling of resources, for instance, not closing opened files
- Deprecation warnings regarding elements of the standard library that have been deprecated and will be removed in future releases
- Enabling a fault handler that outputs an application stack trace when the application receives `SIGSEGV`, `SIGFPE`, `SIGABRT`, `SIGBUS`, or `SIGILL` system signals

Warnings emitted in development mode are indications that something does not work the way it should. They may be useful in finding problems that are not necessarily manifested as errors during the normal operation of your code, but may lead to tangible defects in the long term.

The improper cleanup of opened files may lead at some point to resource exhaustion of the environment your application is running in. File descriptors are resources, the same as RAM or disk storage. Every operating system has a limited number of files that can be opened at the same time. If your application is opening new files without closing them, at some point, it won't be able to open new ones.

Development mode enables you to identify such problems in advance. This is why it is advised to use this mode during application testing. Due to the additional overhead of checks enabled by development mode, it is not recommended to use this in production environments.

Sometimes, development mode can be used to diagnose existing problems, too. An example of really problematic situations is when your application experiences a segmentation fault.

When this happens in Python, you usually won't get any details of the error, except the very brief message printed on your shell's standard output:

```
Segmentation fault: 11
```

When a segmentation fault occurs, the Python process receives a SIGSEGV system signal and terminates instantly. On some operating systems, you will receive a core dump, which is a snapshot of the process memory state recorded at the time of the crash. This can be used to debug your application. Unfortunately, in the case of CPython, this will be a memory snapshot of the interpreter process, so debugging will be taking place at the level of C code.

Development mode installs additional fault handler code that will output the Python stack trace whenever it receives a fault signal. Thanks to this, you will have a bit more information about which part of the code could lead to the problem. The following is an example of known code that will lead to a segmentation fault in Python 3.9:

```python
import sys

sys.setrecursionlimit(1 << 30)

def crasher():
    return crasher()

crasher()
```

If you execute this in Python interpreter with the -X dev flag, you will get output similar to the following:

```
Fatal Python error: Segmentation fault

Current thread 0x000000010b04edc0 (most recent call first):
  File "/Users/user/dev/crashers/crasher.py", line 6 in crasher
  File "/Users/user/dev/crashers/crasher.py", line 6 in crasher
  File "/Users/user/dev/crashers/crasher.py", line 6 in crasher
  File "/Users/user/dev/crashers/crasher.py", line 6 in crasher
  File "/Users/user/dev/crashers/crasher.py", line 6 in crasher
  ...
```

This fault handler can also be enabled outside of development mode. To do that, you can use the -X faulthandler command-line option or set the PYTHONFAULTHANDLER environment variable to 1.

> It's not easy to cause segmentation faults in Python. This often happens for some Python extensions written in C or C++ or functions called from shared libraries (such as DLLs, `.dylib`, or `.so` objects). Still, there are some known and well documented conditions where this problem can occur in pure Python code. The repository of the CPython interpreter includes a collection of such known "crashers." This can be found under `https://github.com/python/cpython/tree/master/Lib/test/crashers`.

Module-level __getattr__() and __dir__() functions

Every Python class can define the custom __getattr__() and __dir__() methods to customize the dynamic attribute access of objects. The __getattr__() function is invoked when a given attribute name is not found to capture a missing attribute lookup and possibly generate a value on the fly. The __dir__() method is called when an object is passed to the dir() function and it should return a list of object attribute names.

Starting from Python 3.7, the __getattr__() and __dir__() functions can be defined at module level. The semantics are similar to object methods. The __getattr__() module-level function, if defined, will be called on a failed module member lookup. The __dir__() function will be called when a module object is passed to the dir() function.

This feature may be useful for library maintainers when deprecating module functions or classes. Let's imagine that we exposed our get_ci() function from the *Type-hinting generics* section in an open source library called dict_helpers.py. If we would like to rename the function to lookup_ci() and still be allowed to import it under the old name, we could use the following deprecation pattern:

```python
from typing import Any
from warnings import warn

def ci_lookup(d: dict[str, Any], key: str) -> Any:
    ...

def __getattr__(name: str):
    if name == "get_ci":
        warn(f"{name} is deprecated", DeprecationWarning)
        return ci_lookup

    raise AttributeError(f"module {__name__} has no attribute {name}")
```

The preceding pattern will emit `DeprecationWarning`, regardless of whether the `get_ci()` function is imported directly from a module (such as via `from dict_helpers import get_ci`) or accessed as a `dict_helpers.get_ci` attribute.

> Deprecation warnings are not visible by default. You can enable them in development mode.

Formatting strings with f-strings

F-strings, also known as **formatted string literals**, are one of the most beloved Python features that came with Python 3.6. Introduced with PEP 498, they added a new way of formatting strings. Prior to Python 3.6, we already had two different string formatting methods. So right now, there are three different ways in which a single string can be formatted:

- Using % formatting: This is the oldest method and uses a substitution pattern that resembles the syntax of the `printf()` function from the C standard library:

```
>>> import math
>>> "approximate value of π: %f" % math.pi
'approximate value of π: 3.141593'
```

- Using the `str.format()` method: This method is more convenient and less error-prone than % formatting, although it is more verbose. It enables the use of named substitution tokens as well as reusing the same value many times:

```
>>> import math
>>> " approximate value of π: {:f}".format(pi=math.pi)
'approximate value of π: 3.141593'
```

- Using formatted string literals (so called **f-strings**). This is the most concise, flexible, and convenient option for formatting strings. It automatically substitutes values in literals using variables and expressions from local namespaces:

```
>>> import math
>>> f"approximate value of π: {math.pi:f}"
'approximate value of π: 3.141593'
```

Formatted string literals are denoted with the `f` prefix, and their syntax is closest to the `str.format()` method, as they use a similar markup for denoting replacement fields in formatted text. In the `str.format()` method, the text substitutions refer to positional and keyword arguments. What makes f-strings special is that replacement fields can be any Python expression, and it will be evaluated at runtime. Inside strings, you have access to any variable that is available in the same namespace as the formatted literal.

The ability to use expressions as replacement fields makes formatting code simpler and shorter. You can also use the same formatting specifiers of replacement fields (for padding, aligning, signs, and so on) as the `str.format()` method, and the syntax is as follows:

```
f"{replacement_field_expression:format_specifier}"
```

The following is a simple example of code executed in an interactive session that prints the first ten powers of the number 10 using f-strings and aligns the results using string formatting with padding:

```
>>> for x in range(10):
...     print(f"10^{x} == {10**x:10d}")
...
10^0 ==          1
10^1 ==         10
10^2 ==        100
10^3 ==       1000
10^4 ==      10000
10^5 ==     100000
10^6 ==    1000000
10^7 ==   10000000
10^8 ==  100000000
10^9 == 1000000000
```

> The full formatting specification of the Python string forms a separate mini language inside Python. The best reference source for this is the official documentation, which you can find under `https://docs.python.org/3/library/string.html`. Another useful internet resource regarding this topic is `https://pyformat.info/`, which presents the most important elements of this specification using practical examples.

Underscores in numeric literals

Underscores in numeric literals are probably one such feature that are the easiest to adopt, but still not as popular as they could be. Starting from Python 3.6, you can use the _ (underscore) character to separate digits in numeric literals. This facilitates the increased readability of big numbers. Consider the following value assignment:

```
account_balance = 100000000
```

With so many zeros, it is hard to tell immediately whether we are dealing with millions or billions. You can instead use an underscore to separate thousands, millions, billions, and so on:

```
account_balance = 100_000_000
```

Now, it is easier to tell immediately that `account_balance` equals one hundred million without carefully counting the zeros.

secrets module

One of the prevalent security mistakes perpetrated by many programmers is assuming randomness from the `random` module. The nature of random numbers generated by the `random` module is sufficient for statistical purposes. It uses the Mersenne Twister pseudorandom number generator. It has a known uniform distribution and a long enough period length that it can be used in simulations, modeling, or numerical integration.

However, Mersenne Twister is a completely deterministic algorithm, as is the `random` module. This means that as a result of knowing its initial conditions (the **seed** number), you can generate the same pseudorandom numbers. Moreover, by knowing enough consecutive results of a pseudorandom generator, it is usually possible to retrieve the seed number and predict the next results. This is true for Mersenne Twister as well.

> If you want to see how random numbers from Mersenne Twister can be predicted, you can review the following project on GitHub: `https://github.com/kmyk/mersenne-twister-predictor`.

That characteristic of pseudorandom number generators means that they should never be used for generating random values in a security context. For instance, if you need to generate a random secret that would be a user password or token, you should use a different source of randomness.

The `secrets` module serves exactly that purpose. It relies on the best source of randomness that a given operating system provides. So, on Unix and Unix-like systems, that would be the `/dev/urandom` device, and on Windows, it will be the `CryptGenRandom` generator.

The three most important functions are:

- `secrets.token_bytes(nbytes=None)`: This returns `nbytes` of random bytes. This function is used internally by `secrets.token_hex()` and `secrets.token_urlsafe()`. If `nbytes` is not specified, it will return a default number of bytes, which is documented as "reasonable."

- `secrets.token_hex(nbytes=None)`: This returns `nbytes` of random bytes in the form of a hex-encoded string (not a `bytes()` object). As it takes two hexadecimal digits to encode one byte, the resulting string will consist of *nbytes × 2* characters. If `nbytes` is not specified, it will return the same default number of bytes as `secrets.token_bytes()`.

- `secrets.token_urlsafe(nbytes=None)`: This returns `nbytes` of random bytes in the form of a URL-safe, base64-encoded string. As a single byte takes approximately 1.3 characters in base64 encoding, the resulting string will consist of *nbytes × 1.3* characters. If `nbytes` is not specified, it will return the same default number of bytes as `secrets.token_bytes()`.

Another important, but often overlooked, function is `secrets.compare_digest(a, b)`. This compares two strings or byte-like objects in a way that does not allow an attacker to guess if they at least partially match by measuring how long it took to compare them. A comparison of two secrets using ordinary string comparison (the `==` operator) is susceptible to a so-called timing attack. In such a scenario, the attacker can try to execute multiple secret verifications and, by performing statistical analysis, gradually guess consecutive characters of the original value.

What may come in the future?

At the time of writing this book, Python 3.9 is still only a few months old, but the chances are that when you're reading this book, Python 3.10 has either already been released or is right around the corner.

As the Python development processes are open and transparent, we have constant insight into what has been accepted in the PEP documents and what has already been implemented in alpha and beta releases. This allows us to review selected features that will be introduced in Python 3.10. The following is a brief review of the most important changes that we can expect in the near future.

Union types with the | operator

Python 3.10 will bring yet another syntax simplification for the purpose of type hinting. Thanks to this new syntax, it will be easier to construct union-type annotations.

Python is dynamically typed and lacks polymorphism. As a result of this, functions can easily accept the same argument, which can be a different type depending on the call, and properly process it if those types have the same interface. To better understand this, let's bring back the signature of a function that allowed case-insensitive loopback of string-keyed dictionary values:

```python
def get_ci(d: dict[str, Any], key: str) -> Any: ...
```

Internally, we used the `upper()` method of keys obtained from the dictionary. That's the main reason why we defined the type of the d argument as `dict[str, Any]`, and the type of `key` argument as `str`.

However, the `str` type is not the only built-in type that has the `upper()` method. The other type that has the same method is `bytes`. If we would like to allow our `get_ci()` function to accept both string-keyed and bytes-keyed dictionaries, we need to specify the union of possible types.

Currently, the only way to specify type unions is through the `typing.Union` hint. This hint allows the union of `bytes` and `str` types to be specified as `typing.Union[bytes, str]`. The complete signature of the `get_ci()` function would be as follows:

```python
def get_ci(
    d: dict[Union[str, bytes], Any],
    key: Union[str, bytes]
) -> Any:
    ...
```

That is already verbose, and for more complex functions, it can get only worse. This is why Python 3.10 will allow the union of types using the | operator to be specified. In the future, you will be able to simply write the following:

```python
def get_ci(d: dict[str | bytes, Any], key: str | bytes) -> Any: ...
```

In contrast to type-hinting generics, the introduction of a type union operator does not deprecate the `typing.Union` hint. This means that we will be able to use those two conventions interchangeably.

Structural pattern matching

Structural pattern matching is definitely the most controversial new Python feature of the last decade, and it is definitely the most complex one.

The acceptance of that feature was preceded by numerous heated debates and countless design drafts. The complexity of the topic is clearly visible if we take a look over all the PEP documents that tried to tackle the problem. The following is a table of all PEP documents related to structural pattern matching (statuses accurate as of March 2021):

Date	PEP	Title	Type	Status
23-Jun-2020	622	Structural Pattern Matching	Standards Track	Superseded by PEP 634
12-Sep-2020	634	Structural Pattern Matching: Specification	Standards Track	Accepted
12-Sep-2020	635	Structural Pattern Matching: Motivation and Rationale	Informational	Final
12-Sep-2020	636	Structural Pattern Matching: Tutorial	Informational	Final
26-Sep-2020	642	Explicit Pattern Syntax for Structural Pattern Matching	Standards Track	Draft
9-Feb-2021	653	Precise Semantics for Pattern Matching	Standards Track	Draft

That's a lot of documents, and none of them are short. So, what is structural pattern matching and how can it be useful?

Structural pattern matching introduces a **match statement** and two new soft keywords: `match` and `case`. As the name suggests, it can be used to match a given value against a list of specified "cases" and act accordingly to the match.

> A soft keyword is a keyword that is not reserved in every context. Both `match` and `case` can be used as ordinary variables or function names outside the match statement context.

For some programmers, the syntax of the match statement resembles the syntax of the switch statement found in languages such as C, C++, Pascal, Java, and Go. It can indeed be used to implement the same programming pattern, but is definitely much more powerful.

The general (and simplified) syntax for a match statement is as follows:

```
match expression:
    case pattern:
        ...
```

`expression` can be any valid Python expression. `pattern` represents an actual matching pattern that is a new concept in Python. Inside a `case` block, you can have multiple statements. The complexity of a match statement stems mostly from the introduction of **match patterns** that may initially be hard to understand. Patterns can also be easily confused with expressions, but they don't evaluate like ordinary expressions do.

But before we dig into the details of match patterns, let's take a look at a simple example of a match statement that replicates the functionality of switch statements from different programming languages:

```python
import sys

match sys.platform:
    case "windows":
        print("Running on Windows")
    case "darwin" :
        print("Running on macOS")
    case "linux":
        print("Running on Linux")
    case _:
        raise NotImplementedError(
            f"{sys.platform} not supported!"
        )
```

This is, of course, a very straightforward example, but already shows some important elements. First, we can use literals as patterns. Second, there is a special _ (underscore) wildcard pattern. Wildcard patterns and other patterns that, from the syntax alone, can be proven to match always create an **irrefutable case block**. An irrefutable case block can be placed only as the last block of a match statement.

The previous example can, of course, be implemented with a simple chain of `if`, `elif`, and `else` statements. A common entry-level recruitment challenge is writing a FizzBuzz program.

A FizzBuzz program iterates from 0 to an arbitrary number and, depending on the value, does three things:

- It prints `Fizz` if the value is divisible by 3
- It prints `Buzz` if the value is divisible by 5
- It prints `FizzBuzz` if the value is divisible by 3 and 5
- It prints the value in all other cases

This is indeed a minor problem, but you would be surprised how people can stumble on even the simplest things when under the stress of an interview. This can, of course, be solved with a couple of `if` statements, but the use of a match statement can give our solution some natural elegance:

```python
for i in range(100):
    match (i % 3, i % 5):
        case (0, 0): print("FizzBuzz")
        case (0, _): print("Fizz")
        case (_, 0): print("Buzz")
        case _: print(i)
```

In the preceding example, we are matching `(i % 3, i % 5)` in every iteration of the loop. We have to do both modulo divisions because the result of every iteration depends on both division results. A match expression will stop evaluating patterns once it finds a matching block and will execute only one block of code.

The notable difference from the previous example is that we used mostly sequence patterns instead of literal patterns:

- The `(0, 0)` pattern: This will match a two-element sequence if both elements are equal to 0.
- The `(0, _)` pattern: This will match a two-element sequence if the first element is equal to 0. The other element can be of any value and type.
- The `(_, 0)` pattern: This will match a two-element sequence if the second element is equal to 0. The other element can be of any value and type.
- The `_` pattern: This is a wildcard pattern that will match all values.

Match expressions aren't limited to simple literals and sequences of literals. You can also match against specific classes and actually, with class patterns, things start to get really magical. That's definitely the most complex part of the whole feature.

At the time of writing, Python 3.10 hasn't yet been released, so it's hard to show a typical and practical use case for class matching patterns. So instead, we will take a look at an example from an official tutorial. The following is a modified example from the PEP 636 document that includes a simple `where_is()` function, which can match against the structure of the `Point` class instance provided:

```python
class Point:
    x: int
    y: int

    def __init__(self, x, y):
        self.x = x
        self.y = y


def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Somewhere else")
        case _:
            print("Not a point")
```

A lot is happening in the preceding example, so let's iterate over all the patterns included here:

- `Point(x=0, y=0)`: This matches if `point` is an instance of the `Point` class and its x and y attributes are equal to 0.
- `Point(x=0, y=y)`: This matches if `point` is an instance of the `Point` class and its x attribute is equal to 0. The y attribute is captured to the y variable, which can be used within the case block.
- `Point(x=x, y=0)`: This matches if `point` is an instance of the `Point` class and its y attribute is equal to 0. The x attribute is captured to the x variable, which can be used within the case block.
- `Point()`: This matches if `point` is an instance of the `Point` class.
- `_`: This always matches.

As you can see, pattern matching can look deep into object attributes. Despite the
`Point(x=0, y=0)` pattern looking like a constructor call, Python does not call an
object constructor when evaluating patterns. It also doesn't inspect arguments and
keyword arguments of `__init__()` methods, so you can access any attribute value in
your match pattern.

Match patterns can also use "positional attribute" syntax, but that requires a bit more
work. You simply need to provide an additional `__match_args__` class attribute that
specifies the natural position order of class instance attributes, as in the following
example:

```python
class Point:
    x: int
    y: int

    __match_args__ = ["x", "y"]

    def __init__(self, x, y):
        self.x = x
        self.y = y


def where_is(point):
    match point:
        case Point(0, 0):
            print("Origin")
        case Point(0, y):
            print(f"Y={y}")
        case Point(x, 0):
            print(f"X={x}")
        case Point():
            print("Somewhere else")
        case _:
            print("Not a point")
```

And that's just the tip of the iceberg. Match statements are actually way more
complex than we could demonstrate in this short section. If we were to consider all
the potential use cases, syntax variants, and corner cases, we could potentially talk
about them throughout the whole chapter. If you want to learn more about them,
you should definitely read though the three "canonical" PEPs: 634, 635, and 636.

Summary

In this chapter, we've covered the most important language syntax and standard library changes that have happened over the last four versions of Python. If you're not actively following Python release notes or haven't yet transitioned to Python 3.9, this should give you enough information to be up to date.

In this chapter, we've also introduced the concept of programming idioms. This is an idea that we will be referring to multiple times throughout the book. In the next chapter, we will take a closer look at many Python idioms by comparing selected features of Python to different programming languages. If you are a seasoned programmer who has just recently transitioned to Python, this will be a great opportunity to learn the "Python way of doing things." It will also be an opportunity to see where Python really shines, and where it might still be behind the competition.

4
Python in Comparison with Other Languages

Many programmers come to Python with prior experience of other programming languages. It happens often that they are already familiar with programming idioms of those languages and try to replicate them in Python. As every programming language is unique, bringing such **foreign idioms** often leads to overly verbose or suboptimal code.

The classic example of a foreign idiom that is often used by inexperienced programmers is iteration over lists. Someone that is familiar with arrays in the C language could write Python code similar to the following example:

```python
for index in range(len(some_list)):
    print(some_list[index])
```

An experienced Pythonic programmer would most probably write:

```python
for item in some_list:
    print(item)
```

Programming languages are often classified by paradigms that can be understood as cohesive sets of features supporting certain "styles of programming." Python is a multiparadigm language and thanks to this, it shares many similarities with a vast amount of other programming languages. As a result, you can write and structure your Python code almost the same way you would do that in Java, C++, or any other mainstream programming language.

Unfortunately, often that won't be as effective as using well-recognized Python patterns. Knowing native idioms allows you to write more readable and efficient code.

This chapter is aimed at programmers experienced with other programming languages. We will review some of the important features of Python together with idiomatic ways of solving common problems. We will also see how these compare to other programming languages and what common pitfalls are lurking for seasoned programmers that are just starting their Python journey. Along the way, we will cover the following topics:

- Class model and object-oriented programming
- Dynamic polymorphism
- Data classes
- Functional programming
- Enumerations

Let's begin by considering the technical requirements.

Technical requirements

The code files for this chapter can be found at `https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition/tree/main/Chapter%204`.

Class model and object-oriented programming

The most prevalent paradigm of Python is **object-oriented programming** (also known as **OOP**). It is centered around objects that encapsulate data (in the form of object attributes) and behavior (in the form of methods). OOP is probably one of the most diverse paradigms. It has many styles, flavors, and implementations that have been developed over many years of programming history. Python takes inspiration from many other languages, so in this section, we will take a look at the implementation of OOP in Python through the prism of different languages.

To facilitate code reuse, extensibility, and modularity, OOP languages usually provide a means for either class composition or inheritance. Python is no different and like many other object-oriented languages supports the subclassing of types.

Python may not have as many object-oriented features as other OOP languages, but it has a pretty flexible data and class model that allows you to implement most OOP patterns with extreme elegance. Also, everything in Python is an object, including functions and class definitions and basic values like integers, floats, Booleans, and strings.

If we would like to find another popular programming language that has similar object-oriented syntax features and a similar data model, one of the closest matches would probably be Kotlin, which is a language that runs (mostly) on **Java Virtual Machine (JVM)**. The following are the similarities between Kotlin and Python:

- A convenient way to call methods of super-classes: Kotlin provides the `super` keyword and Python provides the `super()` function to explicitly reference methods or attributes of super-classes.

- An expression for object self-reference: Kotlin provides the `this` expression, which always references the current object of the class. In Python, the first argument of the method is always an instance reference. By convention, it is named `self`.

- Support for creating data classes: Like Python, Kotlin provides data classes as "syntactic sugar" over classic class definitions to simplify the creation of class-based data structures that are not supposed to convey a lot of behavior.

- The concept of properties: Kotlin allows you to define class property setters and getters as functions. Python provides the `property()` decorator with a similar purpose, together with the concept of descriptors, which allows you to fully customize the attribute access of an object.

What makes Python really stand out in terms of OOP implementation is the approach to inheritance. Python, unlike Kotlin and many other languages, freely permits multiple inheritance (although it often isn't a good idea). Other languages often do not allow this or provide some constraints. Another important Python differentiator is the lack of private/public keywords that would control access to internal object attributes outside of the class definition.

Let's take a closer look at a feature that Python shares with Kotlin and some other JVM-based programming languages, which is access to super-classes through the `super()` call.

Accessing super-classes

There are multiple ways of encapsulating object behavior in OOP languages but one of the most common ones is the usage of classes. Python's OOP implementation is based precisely on the concept of classes and subclassing.

Subclassing is a convenient way of reusing existing classes by enhancing or specializing their behavior. Subclasses often rely on the behavior of their base classes but extend them with additional methods or provide completely new implementations for existing methods by overriding their definitions.

But overriding methods without access to their original implementations within the subclass would not facilitate code reuse at all. That's why Python offers the `super()` function, which returns a proxy object to the method implementations in all base classes. To better understand the potential of the `super()` function, let's imagine we want to subclass a Python dictionary type to allow access to the stored keys through a case-insensitive key lookup. You could use this, for instance, to store HTTP protocol header values as the HTTP protocol specification states that header names are case-insensitive.

The following is a simple example of implementing such a structure in Python through subclassing:

```python
from collections import UserDict
from typing import Any


class CaseInsensitiveDict(UserDict):
    def __setitem__(self, key: str, value: Any):
        return super().__setitem__(key.lower(), value)

    def __getitem__(self, key: str) -> Any:
        return super().__getitem__(key.lower())

    def __delitem__(self, key: str) -> None:
        return super().__delitem__(key.lower())
```

Our implementation of `CaseInsensitiveDict` relies on `collections.UserDict` instead of the built-in `dict` type. Although inheriting from the `dict` type is possible, we would quickly run into inconsistencies as the built-in `dict` type doesn't always call `__setitem__()` to update its state. Most importantly, it won't be used on object initialization and on `update()` method calls. Similar problems can arise when subclassing the `list` type. That's why good practice dictates to use `collections.UserDict` classes for subclassing the `dict` type and `collections.UserList` for subclassing the `list` type.

The core of modified dictionary behavior happens in `__getitem__(self, item: str)` and `__setitem__(self, key: str, value: Any)`. These are methods responsible respectively for accessing dictionary elements using `dictionary[key]` and setting dictionary values using the `dictionary[key] = value` syntax. The typing annotations help us to denote that keys should be strings but values can be any Python type.

`__setitem__()` is responsible for storing and modifying dictionary values. It would not make sense to subclass the base dictionary type and not leverage its internal key-value storage. That's why we use `super().__setitem__()` to invoke the original set-item implementation. But before we allow the value to be stored, we transform the key to lowercase using the `str.lower()` method. That way we ensure that all keys stored in the dictionary will always be lowercase.

The `__getitem__()` method is analogous to the `__setitem__()` implementation. We know that every key is transformed to lowercase before being stored in a dictionary. Thanks to this, when key lookup occurs, we can also transform it to lowercase as well. If the super implementation of the `__getitem__()` method does not return the result, we can be sure that there is no case-insensitive match in the dictionary.

Last but not least, the `__delitem__()` method deletes existing dictionary keys. It uses the same technique to transform a key to lowercase and invoke super-class implementation. Thanks to this, we will be able to remove dictionary keys using the `del dictionary[key]` statement.

The following transcript shows a case-insensitive key lookup of our class in action:

```
>>> headers = CaseInsensitiveDict({
...     "Content-Length": 30,
...     "Content-Type": "application/json",
... })
>>> headers["CONTENT-LENGTH"]
30
>>> headers["content-type"]
'application/json'
```

The above use case for the `super()` function should be simple to follow and understand, but things get a bit more complex when multiple inheritance is involved. Python allows you to use multiple inheritance by introducing the **Method Resolution Order** (**MRO**). We will take a closer look at it in the next section.

Multiple inheritance and Method Resolution Order

Python MRO is based on **C3 linearization**, the deterministic MRO algorithm originally created for the Dylan programming language. The C3 algorithm builds the **linearization** of a class, also called **precedence**, which is an ordered list of the ancestors. This list is used to seek an attribute in a class inheritance tree.

> You can find more information about the Dylan programming language at `http://opendylan.org` and Wikipedia has a great article on C3 linearization that can be found at `https://en.wikipedia.org/wiki/C3_linearization`.

Python didn't have the C3 linearization algorithm as its MRO from the beginning. It was introduced in Python 2.3 together with a common base type for all objects (that is, the `object` type). Before the change to the C3 linearization method, if a class had two ancestors (refer to *Figure 4.1*), the order in which methods were resolved was only easy to compute and track for simple cases that didn't use a multiple inheritance model in a cascading way.

The following is an example of a simple multiple inheritance pattern that would not require any special MRO:

```python
class Base1:
    pass


class Base2:
    def method(self):
        print("Base2.method() called")

class MyClass(Base1, Base2):
    pass
```

Before Python 2.3, that would be a simple depth-first search over a class hierarchy tree. In other words, when `MyClass().method()` is called, the interpreter looks for the method in `MyClass`, then `Base1`, and then eventually finds it in `Base2`.
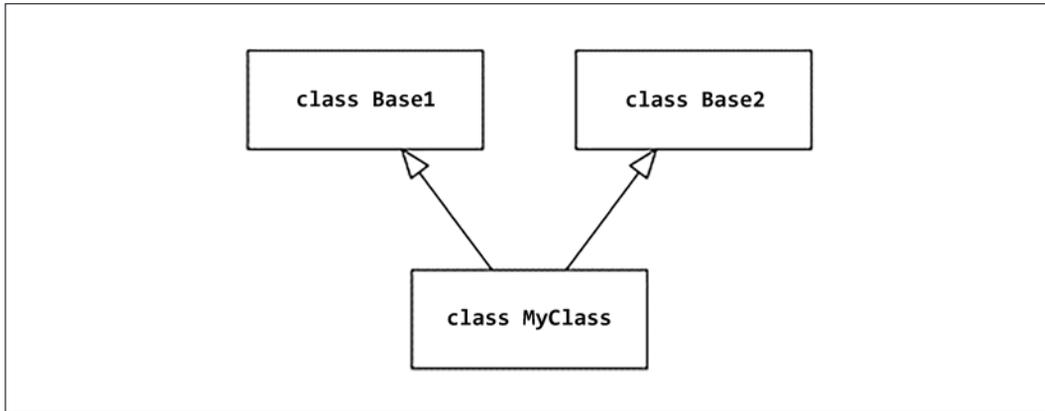
Figure 4.1: Classical hierarchy

When we introduce a `CommonBase` class at the top of our class hierarchy (refer to *Figure 4.2*), things will get more complicated:

```python
class CommonBase:
    pass

class Base1(CommonBase):
    pass


class Base2(CommonBase):
    def method(self):
        print("Base2.method() called")

class MyClass(Base1, Base2):
    pass
```

As a result, the simple resolution order that behaves according to the **left-to-right depth-first** rule is getting back to the top through the `Base1` class before looking into the `Base2` class. This algorithm results in a counterintuitive output. Without the C3 linearization, the method that is executed would not be the one that is the closest in the inheritance tree.
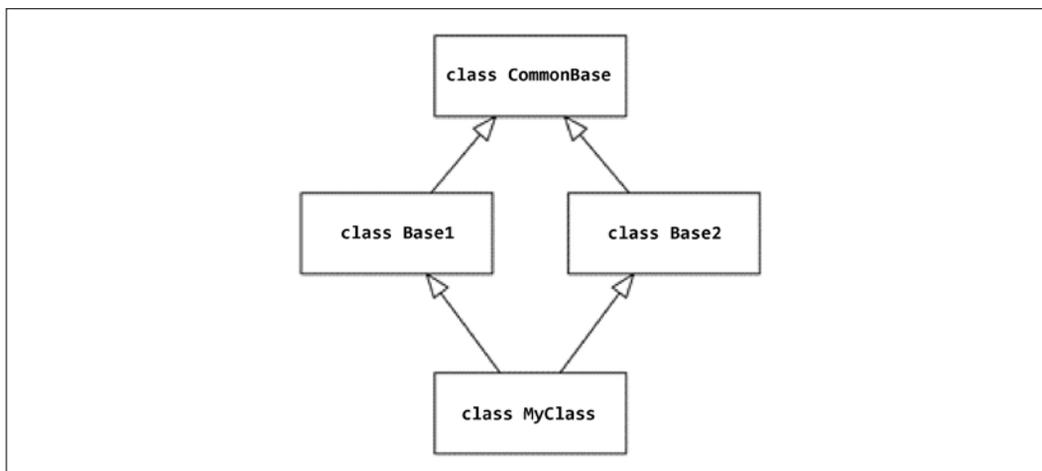
Figure 4.2: The diamond class hierarchy

Such an inheritance scenario (known as the diamond class hierarchy) is rather uncommon for custom-built classes. The standard library typically does not structure the inheritance hierarchies in this way, and many developers think that it is bad practice. It is possible with Python anyway and thus requires a well-defined and clear handling strategy.

Also, starting from Python 2.3, `object` is at the top of the type hierarchy for classes. Essentially, every class is a part of a large diamond class inheritance hierarchy. It became something that has to be resolved on the C side of the language as well. That's why Python now has C3 linearization as the MRO algorithm.

> In Python 2, classes inheriting from the `object` type were called new-style classes. Classes did not inherit implicitly from objects. In Python 3, every class is a new-style class and old-style classes are not available.

The original reference document of the Python MRO written by Michele Simionato describes linearization using the following words:

The linearization of C is the sum of C plus the merge of the linearizations of the parents and the list of the parents.

> The Michele Simionato reference document explaining Python's MRO in great detail can be found at `http://www.python.org/download/releases/2.3/mro`.

The above simply means that C3 is a recursive algorithm. The C3 symbolic notation applied to our earlier inheritance example is as follows:

```
L[MyClass(Base1, Base2)] =
        [MyClass] + merge(L[Base1], L[Base2], [Base1, Base2])
```

Here, `L[MyClass]` is the linearization of `MyClass`, and `merge` is a specific algorithm that merges several linearization results.

The `merge` algorithm is responsible for removing the duplicates and preserving the correct ordering. It uses the concept of list **head** and **tail**. The head is the first element of the list and the tail is the rest of the list following the head. Simionato describes the merge algorithm like this (adapted to our example):

> *Take the head of the first list, that is, L[Base1][0]; if this head is not in the tail of any of the other lists, then add it to the linearization of MyClass and remove it from the lists in the merge, otherwise look at the head of the next list and take it, if it is a good head.*

> *Then, repeat the operation until all the classes are removed or it is impossible to find good heads. In this case, it is impossible to construct the merge; Python 2.3 will refuse to create the MyClass class and will raise an exception.*

In other words, C3 does a recursive depth lookup on each parent to get a sequence of lists. Then, it computes a left-to-right rule to merge all lists with hierarchy disambiguation when a class is involved in several lists.

If we had to calculate the MRO for `MyClass` manually through a series of symbolic steps, we would first have to unfold all `L[class]` linearizations:

```
L[MyClass]
  = [MyClass] + merge(L[Base1], L[Base2], [Base1, Base2])]
  = [MyClass] + merge(
      [Base1 + merge(L[CommonBase], [CommonBase])],
      [Base2 + merge(L[CommonBase], [CommonBase])],
      [Base1, Base2]
    )
  = [MyClass] + merge(
      [Base1] + merge(L[CommonBase], [CommonBase]),
      [Base2] + merge(L[CommonBase], [CommonBase]),
      [Base1, Base2]
    )
  = [MyClass] + merge(
```

```
        [Base1] + merge([CommonBase] + merge(L[object]), [CommonBase]),
        [Base2] + merge([CommonBase] + merge(L[object]), [CommonBase]),
        [Base1, Base2]
    )
```

Essentially, the `object` class has no ancestors so its C3 linearization is just a single element list `[object]`. It means we continue by unfolding `merge([object])` to `[object]`:

```
= [MyClass] + merge(
    [Base1] + merge([CommonBase] + merge([object]), [CommonBase]),
    [Base2] + merge([CommonBase] + merge([object]), [CommonBase]),
    [Base1, Base2]
)
```

`merge([object])` has only a single element list so it immediately unfolds to `[object]`:

```
= [MyClass] + merge(
    [Base1] + merge([CommonBase, object], [CommonBase]),
    [Base2] + merge([CommonBase, object], [CommonBase]),
    [Base1, Base2]
)
```

Now it's time to unfold `merge([CommonBase, object], [CommonBase])`. The head of the first list is `CommonBase`. It is not in the tail of other lists. We can immediately remove it from the merge and add it to the outer linearization result:

```
= [MyClass] + merge(
    [Base1, CommonBase] + merge([object]),
    [Base2, CommonBase] + merge([object]),
    [Base1, Base2]
)
```

We are again left with `merge([object])` and we can continue unfolding:

```
= [MyClass] + merge(
    [Base1, CommonBase, object],
    [Base2, CommonBase, object],
    [Base1, Base2]
)
```

Now we are left with the last merge, which is finally non-trivial. The first head is `Base1`. It is not found in the tails of other lists. We can remove it from the merge and add it to the outer linearization result:

```
  = [MyClass, Base1] + merge(
    [CommonBase, object],
    [Base2, CommonBase, object],
    [Base2]
  )
```

Now the first head is `CommonBase`. It is found in the tail of the second `list` [`Base2`, `CommonBase`, `object`]. It means we can't process it at the moment and have to move to the next head, which is `Base2`. It is not found in the tail of other lists. We can remove it from the merge and add it to the outer linearization result:

```
  = [MyClass, Base1, Base2] + merge(
    [CommonBase, object],
    [CommonBase, object],
    []
  )
```

Now, `CommonBase` is again the first head but this time it is no longer found in other list tails. We can remove it from the merge and add it to the outer linearization result:

```
  = [MyClass, Base1, Base2, CommonBase] + merge(
    [object],
    [object],
    []
  )
```

The last `merge([object], [object], [])` step is trivial. The final linearization result is the following:

```
  [MyClass, Base1, Base2, CommonBase, object]
```

You can easily inspect the results of C3 linearization by verifying the __mro__ attribute of any class. The following transcript presents the computed MRO of `MyClass`:

```
>>> MyClass.__mro__
(<class '__main__.MyClass'>, <class '__main__.Base1'>, <class '__main__.Base2'>, <class '__main__.CommonBase'>, <class 'object'>)
```

The __mro__ attribute of a class (which is read-only) stores the result of the C3 linearization computation. Computation is done when the class definition is loaded. You can also call `MyClass.mro()` to compute and get the result.

Class instance initialization

An object in OOP is an entity that encapsulates data together with behavior. In Python, data is contained as object attributes, which are simply object variables. Behavior, on the other hand, is represented by methods. That is common to almost every OOP language, but the exact nomenclature is sometimes different. For instance, in C++ and Java, object data is said to be stored in fields. In Kotlin, object data is stored behind properties (although they are a bit more than simple object variables).

What makes Python different from statically typed OOP languages is its approach to object attribute declaration and initialization. In short, Python classes do not require you to define attributes in the class body. A variable comes into existence at the time it is initialized. That's why the canonical way to declare object attributes is through assigning their values during object initialization in the __init__() method:

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

That may be confusing for those coming to Python with prior knowledge of statically typed programming languages. In those languages, the declaration of object fields is usually static and lives outside of the object initialization function. That's why programmers with a C++ or Java background often tend to replicate this pattern by assigning some default values as class attributes in the main class body:

```python
class Point:
    x = 0
    y = 0

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

The above code is a classic example of a foreign language idiom replicated in Python. Most of all, it is redundant: class attribute values will always be shadowed by object attributes upon initialization. But it is also a dangerous **code smell**: it can lead to problematic errors if one decides to assign as a class attribute a mutable type like list or dict.

> A code smell is a characteristic of code that may be a sign of a deeper problem. A specific piece of code can be functionally correct and free from errors but can be a stub for future problems. Code smells are usually small architectural deficiencies or unsafe constructs that attract bugs.

The problem comes from the fact that class attributes (attributes assigned outside of the method body) are assigned to type objects and not type instances. When accessing an attribute with `self.attribute`, Python will first look up the name `attribute` value in the class instance namespace. If that lookup fails, it will perform a lookup in the class type namespace. When assigning values through `self.attribute` from within the class method, the behavior is completely different: new values are always assigned in the class instance namespace. This is especially troublesome with mutable types as this may cause an accidental leak of the object state between class instances.

Because using mutable types as class attributes instead of instance attributes is rather a bad practice, it is hard to come up with code examples that would be practical. But it doesn't mean we can't take a quick look at how it actually works. Consider the following class, which is supposed to aggregate values as a list and track the last aggregated value:

```python
class Aggregator:
    all_aggregated = []
    last_aggregated = None

    def aggregate(self, value):
        self.last_aggregated = value
        self.all_aggregated.append(value)
```

To see where the problem lies, let's start an interactive session, create two distinct aggregators, and start aggregating elements:

```python
>>> a1 = Aggregator()
>>> a2 = Aggregator()
>>> a1.aggregate("a1-1")
>>> a1.aggregate("a1-2")
>>> a2.aggregate("a2-1")
```

If we now take a look at the aggregation lists of both instances, we will see very disturbing output:

```
>> a1.all_aggregated
['a1-1', 'a1-2', 'a2-1']
>>> a2.all_aggregated
['a1-1', 'a1-2', 'a2-1']
```

Someone reading the code could think that all `Aggregator` instances are supposed to track the history of their own aggregations. But we see that instead, all `Aggregator` instances share the state of the `all_aggregated` attribute. On the other hand, when looking at the last aggregated values, we see correct values for both aggregators:

```
>>> a1.last_aggregated
'a1-2'
>>> a2.last_aggregated
'a2-1'
```

In situations like these, it is easy to solve the mystery by inspecting the unbound class attribute values:

```
>>> Aggregator.all_aggregated
['a1-1', 'a1-2', 'a2-1']
>>> Aggregator.last_aggregated
>>>
```

As we see from the above transcript, all `Aggregator` instances shared their state through the mutable `Aggregator.all_aggregated` attribute. Something like this could be the intended behavior but very often is just an example of a mistake that is sometimes hard to track down. Due to this fact, all attribute values that are supposed to be unique for every class instance should absolutely be initialized in the `__init__()` method only.

The fixed version of the `Aggregator` class would be as follows:

```python
class Aggregator:
    def __init__(self):
        self.all_aggregated = []
        self.last_aggregated = None

    def aggregate(self, value):
        self.last_aggregated = value
        self.all_aggregated.append(value)
```

We simply moved the initialization of the `all_aggregated` and `last_aggregated` attributes to the `__init__()` method. Now let's repeat the same initialization and aggregation calls as in the previous session:

```
>>> a1 = Aggregator()
>>> a2 = Aggregator()
>>> a1.aggregate("a1-1")
>>> a1.aggregate("a1-2")
>>> a2.aggregate("a2-1")
```

If we now inspect the state of `Aggregator` instances, we will see that they track the history of their aggregations independently:

```
>>> a1.all_aggregated
['a1-1', 'a1-2']
>>> a2.all_aggregated
['a2-1']
```

If you really feel the urge to have some kind of declaration of all attributes at the top of the class definition, you can use type annotations as in the following example:

```python
from typing import Any, List

class Aggregator:
    all_aggregated: List[Any]
    last_aggregated: Any

    def __init__(self):
        self.all_aggregated = []
        self.last_aggregated = None

    def aggregate(self, value: Any):
        self.last_aggregated = value
        self.all_aggregated.append(value)
```

Having class attribute annotations actually isn't a bad practice. They can be used by static type verifiers or IDEs to increase the quality of code and better communicate the intended usage of your class and possible type constraints. Such class attribute annotations are also used to simplify the initialization of data classes, which we will discuss in the *Data classes* section.

Attribute access patterns

Another thing that sets Python apart from other statically typed object-oriented languages is the lack of the notion of public, private, and protected class members. In other languages, these are often used to restrict or open access to object attributes for code outside of the class. The Python feature that is nearest to this concept is **name mangling**. Every time an attribute is prefixed by __ (two underscores) within a class body, it is renamed by the interpreter on the fly:

```python
class MyClass:
    def __init__(self):
        self.__secret_value = 1
```

> Note that the double underscore pattern is referred to as a
> "dunder". Refer to the *Dunder methods (language protocols)* section
> for more information.

Accessing the __secret_value attribute by its initial name outside of the class will raise an AttributeError exception:

```python
>>> instance_of = MyClass()
>>> instance_of.__secret_value
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyClass' object has no attribute '__secret_value'
>>> instance_of._MyClass__secret_value
1
```

One could think that this is synonymous with private/protected fields and methods commonly found in other OOP languages. It indeed makes it harder to access such attributes outside of the class but doesn't make such access impossible. Private and protected fields and methods in many other OOP languages are a means of providing class encapsulation. They are used to restrict access to specific symbols from anyone outside of a specific class (private) or anyone outside the inheritance tree (protected). In Python, name mangling does not restrict attribute access in any way. It only makes it less convenient.

The purpose of name mangling is an implicit way to avoid naming collisions. For instance, it may happen that a specific identifier is a perfect fit for a new internal attribute in some subclass. If that name is already taken somewhere up in the inheritance tree, the name clash may result in unexpected behavior.

In such situations, the programmer may decide to use a different name or use name mangling to resolve the conflict. Name mangling can also help in avoiding name clashes in subclasses. Still, it is not recommended to use name mangling in base classes by default, just to avoid any collisions in advance.

It all boils down to the Python way of doing things. Statically typed languages with private/protected keywords enforce the attribute access restriction. It means that usually there is no way to access such private/protected attributes outside of the class. In Python, it is more common to clearly communicate what the intended use is of each attribute instead of restricting users from doing whatever they want. With or without name mangling, programmers will find a way to access the attribute anyway. So, what's the purpose of making this less convenient for them?

When an attribute is not public, the convention to use is an _ prefix. This does not involve any name mangling algorithm, but just usually documents the attribute as an internal element of the class that is not intended to be used outside of the class context. Many IDEs and style checkers are already aware of this convention and are able to highlight places where such internal members are accessed outside of their class.

Python also has other mechanisms to separate the public part of the class from its private code. Two such mechanisms are descriptors and properties.

Descriptors

A descriptor lets you customize what should be done when you refer to an attribute of an object. Descriptors are the basis of complex attribute access in Python. They are used internally to implement properties, methods, class methods, static methods, and `super`. They are objects that define how attributes of another class can be accessed. In other words, a class can delegate the management of an attribute to another class.

The descriptor classes are based on three special methods that form the **descriptor protocol**:

- `__set__(self, obj, value)`: This is called whenever the attribute is set. In the following examples, we will refer to this as a **setter**.

- `__get__(self, obj, owner=None)`: This is called whenever the attribute is read (referred to as a **getter**).

- `__delete__(self, obj)`: This is called when `del` is invoked on the attribute.

A descriptor that implements __get__() and __set__() is called a **data descriptor**. If it just implements __get__(), then it is called a **non-data descriptor**.

Methods of the descriptor protocol are, in fact, called by the object's special __getattribute__() method on every attribute lookup (do not confuse it with __getattr__(), which has a different purpose). Whenever such a lookup is performed, either by using a dotted notation in the form of instance.attribute or by using the getattr(instance, 'attribute') function call, the __getattribute__() method is implicitly invoked and it looks for an attribute in the following order:

1. It verifies whether the attribute is a data descriptor on the class object of the instance

2. If not, it looks to see whether the attribute can be found in the __dict__ lookup of the instance object

3. Finally, it looks to see whether the attribute is a non-data descriptor on the class object of the instance

In other words, data descriptors take precedence over the __dict__ lookup, and the __dict__ lookup takes precedence over non-data descriptors.

To make it clearer, here is a modified example from the official Python documentation that shows how descriptors work on real code:

```python
class RevealAccess(object):
    """A data descriptor that sets and returns values
       normally and prints a message logging their access.
    """

    def __init__(self, initval=None, name='var'):
        self.val = initval
        self.name = name

    def __get__(self, obj, objtype):
        print('Retrieving', self.name)
        return self.val

    def __set__(self, obj, val):
        print('Updating', self.name)
        self.val = val

    def __delete__(self, obj):
        print('Deleting', self.name)
```

```python
class MyClass(object):
    x = RevealAccess(10, 'var "x"')
    y = 5
```

> The official guide on using descriptors, together with many examples, can be found at `https://docs.python.org/3.9/howto/descriptor.html`.

Note that `x = RevealAccess()` is defined as a class attribute instead of assigning it in the `__init__()` method. Descriptors, in order to work, need to be defined as class attributes. Also, they are closer to methods than normal variable attributes. Here is an example of using the `RevealAccess` descriptor in the interactive session:

```python
>>> m = MyClass()
>>> m.x
Retrieving var "x"
10
>>> m.x = 20
Updating var "x"
>>> m.x
Retrieving var "x"
20
>>> m.y
5
>>> del m.x
Deleting var "x"
```

The preceding example clearly shows that, if a class has the data descriptor for the given attribute, then the descriptor's `__get__()` method is called to return the value every time the instance attribute is retrieved, and `__set__()` is called whenever a value is assigned to such an attribute. The `__del__` method of a descriptor is called whenever an instance attribute is deleted with the `del instance.attribute` statement or the `delattr(instance, 'attribute')` call.

The difference between data and non-data descriptors is important for the reasons highlighted at the beginning of the section. Python already uses the descriptor protocol to bind class functions to instances as methods.

Descriptors also power the mechanism behind the `classmethod` and `staticmethod` decorators. This is because, in fact, the function objects are non-data descriptors too:

```
>>> def function(): pass
>>> hasattr(function, '__get__')
True
>>> hasattr(function, '__set__')
False
```

This is also true for functions created with lambda expressions:

```
>>> hasattr(lambda: None, '__get__')
True
>>> hasattr(lambda: None, '__set__')
False
```

So, without `__dict__` taking precedence over non-data descriptors, we would not be able to dynamically override specific methods on already constructed instances at runtime. Fortunately, thanks to how descriptors work in Python, it is possible; so, developers may use a popular technique called **monkey patching** to change the way in which instances work ad hoc without the need for subclassing.

> Monkey patching is the technique of modifying the class instance dynamically at runtime by adding, modifying, or deleting attributes without touching the class definition or the source code.

Real-life example – lazily evaluated attributes

One example usage of descriptors may be to delay the initialization of the class attribute to the moment when it is accessed from the instance. This may be useful if the initialization of such attributes depends on some context that is not yet available at the time the class is imported. The other case is saving resources when such initialization is simply expensive in terms of computing resources but it is not known whether the attribute will be used anyway at the time the class is imported. Such a descriptor could be implemented as follows:

```
class InitOnAccess:
    def __init__(self, init_func, *args, **kwargs):
        self.klass = init_func
        self.args = args
        self.kwargs = kwargs
        self._initialized = None
```

```python
    def __get__(self, instance, owner):
        if self._initialized is None:
            print('initialized!')
            self._initialized = self.klass(*self.args,
             **self.kwargs)
        else:
            print('cached!')
        return self._initialized
```

The `InitOnAccess` descriptor class includes some `print()` calls that allow us to see whether values were initialized on access or accessed from the cache.

Let's imagine we want to have a class where all instances have access to a shared list of sorted random values. The length of the list could be arbitrarily long, so it makes sense to reuse it for all instances. On the other hand, sorting very long input can be time-consuming. That's why the `InitOnAccess` class will make sure that such a list will be initialized only on first access. Our class definition could be as follows:

```python
import random

class WithSortedRandoms:
    lazily_initialized = InitOnAccess(
        sorted,
        [random.random() for _ in range(5)]
    )
```

Note that we used fairly small input to the `range()` function to make the output readable. Here is an example usage of the `WithSortedRandoms` class in an interactive session:

```
>>> m = WithSortedRandoms()
>>> m.lazily_initialized
initialized!
[0.2592159616928279, 0.32590583255950756, 0.4015520901807743,
0.4148447834912816, 0.4187058605495758, 0.4534290894962043,
0.4796775578337028, 0.6963642650184283, 0.8449725511007807,
0.8808174325885045]
>>> m.lazily_initialized
cached!
[0.2592159616928279, 0.32590583255950756, 0.4015520901807743,
0.4148447834912816, 0.4187058605495758, 0.4534290894962043,
0.4796775578337028, 0.6963642650184283, 0.8449725511007807,
0.8808174325885045]
```

The official OpenGL Python library available on PyPI under the `PyOpenGL` name uses a similar technique to implement a `lazy_property` object that is both a decorator and a data descriptor:

```python
class lazy_property(object):
    def __init__(self, function):
        self.fget = function

    def __get__(self, obj, cls):
        value = self.fget(obj)
        setattr(obj, self.fget.__name__, value)
        return value
```

The `setattr()` function allows you to set the attribute of the object instance by using the attribute from the provided positional argument. Here, it is `self.fget.__name__`. It is constructed like that because the `lazy_property` descriptor is supposed to be used as a decorator of the method acting as a provider of the initialized value as in the following example:

```python
class lazy_property(object):
    def __init__(self, function):
        self.fget = function

    def __get__(self, obj, cls):
        value = self.fget(obj)
        setattr(obj, self.fget.__name__, value)
        return value


class WithSortedRandoms:
    @lazy_property
    def lazily_initialized(self):
        return sorted([[random.random() for _ in range(5)]])
```

Such an implementation is similar to using the `property` decorator described in the next section. The function that is wrapped with it is executed only once and then the instance attribute is replaced with a value returned by that function property. This instance attribute takes precedence over the descriptor (the class attribute) so no more initializations will be performed on the given class instance. This technique is often useful when there's a need to fulfill the following two requirements at the same time:

- An object instance needs to be stored as a class attribute that is shared between its instances (to save resources)

- This object cannot be initialized at the time of import because its creation process depends on some global application state/context

In the case of applications written using OpenGL, you can encounter this kind of situation very often. For example, the creation of shaders in OpenGL is expensive because it requires a compilation of code written in **OpenGL Shading Language** (**GLSL**). It is reasonable to create them only once, and, at the same time, include their definition in close proximity to classes that require them. On the other hand, shader compilations cannot be performed without OpenGL context initialization, so it is hard to define and compile them reliably in a global module namespace at the time of import.

The following example shows the possible usage of the modified version of PyOpenGL's `lazy_property` decorator (here, `lazy_class_attribute`) in some imaginary OpenGL-based application. The highlighted change to the original `lazy_property` decorator was required in order to allow the attribute to be shared between different class instances:

```python
import OpenGL.GL as gl
from OpenGL.GL import shaders


class lazy_class_attribute(object):
    def __init__(self, function):
        self.fget = function

    def __get__(self, obj, cls):
        value = self.fget(cls)
        # note: storing in class object not its instance
        #       no matter if its a class-level or
        #       instance-level access
        setattr(cls, self.fget.__name__, value)
        return value


class ObjectUsingShaderProgram(object):
    # trivial pass-through vertex shader implementation
    VERTEX_CODE = """
        #version 330 core
        layout(location = 0) in vec4 vertexPosition;
        void main(){
```

```
            gl_Position =  vertexPosition;
        }
    """
    # trivial fragment shader that results in everything
    # drawn with white color
    FRAGMENT_CODE = """
        #version 330 core
        out lowp vec4 out_color;
        void main(){
            out_color = vec4(1, 1, 1, 1);
        }
    """


    @lazy_class_attribute
    def shader_program(self):
        print("compiling!")
        return shaders.compileProgram(
            shaders.compileShader(
                self.VERTEX_CODE, gl.GL_VERTEX_SHADER
            ),
            shaders.compileShader(
                self.FRAGMENT_CODE, gl.GL_FRAGMENT_SHADER
            )
        )
```

Like every advanced Python syntax feature, this one should also be used with
caution and documented well in code. Descriptors affect the very basic part of class
behavior. For inexperienced developers, the altered class behavior might be very
confusing and unexpected. Because of that, it is very important to make sure that all
your team members are familiar with descriptors and understand this concept well
if it plays an important role in your project's code base.

Properties

Anyone who has programmed in C++ or Java for a while should probably be
familiar with the term **encapsulation**. It is a means of protecting direct access to class
fields coming from the assumption that all internal data held by a class should be
considered private. In a fully encapsulated class, as few methods as possible should
be exposed as public. Any write or read access to an object's state should be exposed
through setter and getter methods that are able to guard proper usage. In Java, for
instance, this pattern can look as follows:

```java
public class UserAccount {
  private String username;

  public String getUsername() {
    return username;
  }

  public void setUsername(String newUsername) {
    this.username = newUsername;
  }
}
```

The getUsername() method is a username **getter** and the setUsername() method is a username **setter**. The premise is quite good. By hiding access to class members behind getters and setters (also known as accessors and mutators), you are able to guard the right access to internal class values (let's say, perform validation on setters). You are also creating an extension point in the class public API that can be potentially enriched with additional behavior whenever there is such a need without breaking the backward compatibility of the class API.

Let's imagine that you have a class for a user account that, among others, stores the user's password. If you would like to emit audit logs whenever a password is accessed, you could either make sure that every place in your code that accesses user passwords has proper audit log calls or proxy all access to password entry through a set of setter and getter methods that have the logging call added by default.

The problem is that you can never be sure what will require an additional extension in the future. This simple fact often leads to over-encapsulation and a never-ending litany of setter and getter methods for every possible field that could otherwise be public. They are simply tedious to write, and way too often provide little to no benefit and just reduce the signal-to-noise ratio.

Thankfully, Python has a completely different approach to the accessor and mutator pattern through the mechanism of properties. Properties allow you to freely expose public members of classes and simply convert them to getter and setter methods whenever there is such a need. And you can do that completely without breaking the backward compatibility of your class API. Consider the example of an encapsulated UserAccount class that does not use the feature of properties:

```python
class UserAccount:
    def __init__(self, username, password):
        self._username = username
        self._password = password
```

```python
    def get_username(self):
        return self._username

    def set_username(self, username):
        self._username = username

    def get_password(self):
        return self._password

    def set_username(self, password):
        self._password = password
```

Whenever you see code like the above, which can be recognized by the abundance of `get_` and `set_` methods, you can be almost 100% sure that you're dealing with a foreign language idiom. That's something that a C++ or Java programmer could write. A seasoned Python programmer would rather write the following:

```python
class UserAccount:
    def __init__(self, username, password):
        self.username = username
        self.password = password
```

And only when there's an actual need to hide a specific field behind a property, not sooner, an experienced programmer would provide the following modification:

```python
class UserAccount:
    def __init__(self, username, password):
        self.username = username
        self._password = password

    @property
    def password(self):
        return self._password

    @password.setter
    def password(self, value):
        self._password = value
```

The properties provide a built-in descriptor type that knows how to link an attribute to a set of methods. The `property()` function takes four optional arguments: `fget`, `fset`, `fdel`, and `doc`. The last one can be provided to define a `docstring` function that is linked to the attribute as if it were a method. Here is an example of a `Rectangle` class that can be controlled either by direct access to attributes that store two corner points or by using the `width` and `height` properties:

```python
class Rectangle:
    def __init__(self, x1, y1, x2, y2):
        self.x1, self.y1 = x1, y1
        self.x2, self.y2 = x2, y2

    def _width_get(self):
        return self.x2 - self.x1

    def _width_set(self, value):
        self.x2 = self.x1 + value

    def _height_get(self):
        return self.y2 - self.y1

    def _height_set(self, value):
        self.y2 = self.y1 + value

    width = property(
        _width_get, _width_set,
        doc="rectangle width measured from left"
    )
    height = property(
        _height_get, _height_set,
        doc="rectangle height measured from top"
    )

    def __repr__(self):
        return "{}({}, {}, {}, {})".format(
            self.__class__.__name__,
            self.x1, self.y1, self.x2, self.y2
        )
```

The following is an example of such defined properties in an interactive session:

```
>>> rectangle = Rectangle(10, 10, 25, 34)
>>> rectangle.width, rectangle.height
(15, 24)
>>> rectangle.width = 100
>>> rectangle
Rectangle(10, 10, 110, 34)
>>> rectangle.height = 100
>>> rectangle
Rectangle(10, 10, 110, 110)
>>> help(Rectangle)
Help on class Rectangle

class Rectangle(builtins.object)
 |  Methods defined here:
 |
 |  __init__(self, x1, y1, x2, y2)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __repr__(self)
 |      Return repr(self).
 |
 |  ----------------------------------------------------
 |  Data descriptors defined here:
 |  (...)
 |
 |  height
 |      rectangle height measured from top
 |
 |  width
 |      rectangle width measured from left
```

The properties make it easier to write descriptors but must be handled carefully when using inheritance over classes. The attribute created is made on the fly using the methods of the current class and will not use methods that are overridden in the derived classes.

For instance, the following example will fail to override the implementation of the fget method of the parent class's width property:

```
>>> class MetricRectangle(Rectangle):
...     def _width_get(self):
...         return "{} meters".format(self.x2 - self.x1)
...
>>> Rectangle(0, 0, 100, 100).width
100
```

In order to resolve this, the whole property simply needs to be overwritten in the derived class:

```
>>> class MetricRectangle(Rectangle):
...     def _width_get(self):
...         return "{} meters".format(self.x2 - self.x1)
...     width = property(_width_get, Rectangle.width.fset)
...
>>> MetricRectangle(0, 0, 100, 100).width
'100 meters'
```

Unfortunately, the preceding code has some maintainability issues. It can be a source of confusion if the developer decides to change the parent class but forgets to update the `property` call. This is why overriding only parts of the property behavior is not advised. Instead of relying on the parent class's implementation, it is recommended that you rewrite all the property methods in the derived classes if you need to change how they work. In most cases, this is the only option, because usually, the change to the property `setter` behavior implies a change to the behavior of `getter` as well.

Because of this, the best syntax for creating properties is to use `property` as a decorator. This will reduce the number of method signatures inside the class and make the code more readable and maintainable:

```
class Rectangle:
    def __init__(self, x1, y1, x2, y2):
        self.x1, self.y1 = x1, y1
        self.x2, self.y2 = x2, y2

    @property
    def width(self):
        """rectangle width measured from left"""
        return self.x2 - self.x1

    @width.setter
    def width(self, value):
```

```
        self.x2 = self.x1 + value

    @property
    def height(self):
        """rectangle height measured from top"""
        return self.y2 - self.y1

    @height.setter
    def height(self, value):
        self.y2 = self.y1 + value
```

The best thing about the Python property mechanism is that it can be introduced to a class gradually. You can start by exposing public attributes of the class instance and convert them to properties only if there is such a need. Other parts of your code won't notice any change in the class API because properties are accessed as if they were ordinary instance attributes.

We've so far discussed the object-oriented data model of Python in comparison to different programming languages. But the data model is only a part of the OOP landscape. The other important factor of every object-oriented language is the approach to polymorphism. Python provides a few implementations of polymorphism and that will be the topic of the next section.

Dynamic polymorphism

Polymorphism is a mechanism found commonly in OOP languages. Polymorphism abstracts the interface of an object from its type. Different programming languages achieve polymorphism through different means. For statically typed languages, it is usually achieved through:

- **Subtyping**: Subtypes of type A can be used in every interface that expects type A. Interfaces are defined explicitly, and subtypes/subclasses inherit interfaces of their parents. This is a polymorphism mechanism found in C++.

- **Implicit interfaces**: Every type can be used in the interface that expects an interface of type A as long as it implements the same methods (has the same interface) as type A. The declarations of interfaces are still defined explicitly but subclasses/subtypes don't have to explicitly inherit from the base classes/types that define such an interface. This is a polymorphism mechanism found in Go.

Python is a dynamically typed language, so uses a rather lax mechanism of polymorphism that is often referred to as **duck typing**. The duck typing principle says the following:

> *If it walks like a duck and it quacks like a duck, then it must be a duck.*

Application of that principle in Python means that any object can be used within a given context as long as the object works and behaves as the context expects. This typing philosophy is very close to implicit interfaces known in Go, although it does not require any declaration of the expected interfaces of function arguments. Because Python does not enforce types or interfaces of function arguments, it does not matter what types of objects are provided to the function. What matters instead is which methods of those objects are actually used within the function body.

To better understand the concept, consider the following example of a function that is supposed to read a file, print its contents, and close the file afterward:

```python
def printfile(file):
    try:
        contents = file.read()
        print(file)
    finally:
        file.close()
```

From the signature of the `printfile()` function, we can already guess that it expects a file or a file-like object (like `StringIO` from the `io` module). But the truth is this function will consume any object without raising an unexpected exception if we are able to ensure for the input argument that:

- The `file` argument has a `read()` method
- The result of `file.read()` is a valid argument to the `print()` function
- The `file` argument has the `close()` method

The above three points also indicate the three places where polymorphism happens in the above example. Depending on the actual type of the `file` argument, the `printfile()` function will use different implementations of the `read()` and `close()` methods. The type of the `contents` variable can also be different depending on the `file.read()` implementation, in which case the `print()` function will use different implementation of object string representation.

This approach to polymorphism and typing is really powerful and flexible, although it has some downsides. Due to the lack of type and interface enforcement, it is harder to verify the code's correctness before execution. That's why high-quality applications must rely on extensive code testing with rigorous coverage of every path that code execution can take. Python allows you to partially overcome this problem through type hinting annotations that can be verified with additional tools before runtime.

The dynamic type system of Python together with the duck-typing principle creates an implicit and omnipresent form of dynamic polymorphism that makes Python very similar to JavaScript, which also lacks static type enforcement. But there are other forms of polymorphism available to Python developers that are more "classical" and explicit in nature. One of those forms is operator overloading.

Operator overloading

Operator overloading is a specific type of polymorphism that allows the language to have different implementations of specific operators depending on the types of operands.

Operators in many programming languages are already polymorphic. Consider the following expressions that would be valid constructs in Python:

```
7 * 6
3.14 * 2
["a", "b"] * 3
"abba" * 2
```

Those expressions in Python would have four different implementations:

- `7 * 6` is integer multiplication resulting in an integer value of `42`
- `3.14 * 2` is float multiplication resulting in a float value of `6.28`
- `["a", "b"] * 3` is list multiplication resulting in a list value of `['a', 'b', 'a', 'b', 'a', 'b']`
- `"abba" * 2` is string multiplication resulting in a string value of `'abbaabba'`

The semantics and implementation of all Python operators are already different depending on the types of operands. Python provides multiple built-in types together with various implementations of their operators, but it doesn't mean that every operator can be used with any type.

For instance, the + operator is used for the summation or concatenation of operands. It makes sense to concatenate numeric types like integer or floating-point numbers, as well as to concatenate strings and lists. But this operator can't be used with sets or dictionaries as such an operation would not make mathematical sense (sets could be either intersected or joined) and the expected result would be ambiguous (which values of two dictionaries should be used in the event of conflict?).

Operator overloading is just the extension of the built-in polymorphism of operators already included in the programming language. Many programming languages, including Python, allow you to define a new implementation for operand types that didn't have a valid operator implementation or shadow existing implementation through subclassing.

Dunder methods (language protocols)

The Python data model specifies a lot of specially named methods that can be overridden in your custom classes to provide them with additional syntax capabilities. You can recognize these methods by their specific naming conventions that wrap the method name with **double underscores**. Because of this, they are sometimes referred to as **dunder methods**. It is simply shorthand for double underscores.

The most common and obvious example of such dunder methods is __init__(), which is used for class instance initialization:

```python
class CustomUserClass:
    def __init__(self, initiatization_argument):
        ...
```

These methods, either alone or when defined in a specific combination, constitute the so-called **language protocols**. If we say that an object implements a specific language protocol, it means that it is compatible with a specific part of the Python language syntax. The following is a table of the most common protocols within the Python language.

Protocol name	Methods	Description
Callable protocol	__call__()	Allows objects to be called with parentheses: `instance()`
Descriptor protocols	__set__(), __get__(), and __del__()	Allows us to manipulate the attribute access pattern of classes (see the *Descriptors* section)
Container protocol	__contains__()	Allows us to test whether or not an object contains some value using the `in` keyword: `value in instance`

Iterable protocol	`__iter__()`	Allows objects to be iterated using the `for` keyword:
		
		`    for value in instance:`
		
		`        ...`

Sequence protocol	`__getitem__(),`
	
	`__len__()`
		Allows objects to be indexed with square bracket syntax and queried for length using a built-in function:
		
		`item = instance[index]`
		
		`length = len(instance)`

Each operator available in Python has its own protocol and operator overloading happens by implementing the dunder methods of that protocol. Python provides over 50 overloadable operators that can be divided into five main groups:

- Arithmetic operators
- In-place assignment operators
- Comparison operators
- Identity operators
- Bitwise operators

That's a lot of protocols so we won't discuss all of them here. We will instead take a look at a practical example that will allow you to better understand how to implement operator overloading on your own.

> A full list of available dunder methods can be found in the *Data model* section of the official Python documentation available at `https://docs.python.org/3/reference/datamodel.html`.
>
> All operators are also exposed as ordinary functions in the `operators` module. The documentation of that module gives a good overview of Python operators. It can be found at `https://docs.python.org/3.9/library/operator.html`.

Let's assume that we are dealing with a mathematical problem that can be solved through matrix equations. A matrix is a mathematical element of linear algebra with well-defined operations. In the simplest form, it is a two-dimensional array of numbers. Python lacks native support for multi-dimensional arrays other than nesting lists within lists. Because of that, it would be a good idea to provide a custom class that encapsulates matrices and operations between them. Let's start by initializing our class:

```python
class Matrix:
    def __init__(self, rows):
        if len(set(len(row) for row in rows)) > 1:
            raise ValueError("All matrix rows must be the same length")

        self.rows = rows
```

The first dunder method of the `Matrix` class is `__init__()`, which allows us to safely initialize the matrix. It accepts a variable list of matrix rows as input arguments through argument unpacking. As every row needs to have the same number of columns, we iterate over them and verify that they all have the same length.

Now let's add the first operator overloading:

```python
def __add__(self, other):
    if (
        len(self.rows) != len(other.rows) or
        len(self.rows[0]) != len(other.rows[0])
    ):
        raise ValueError("Matrix dimensions don't match")

    return Matrix([
        [a + b for a, b in zip(a_row, b_row)]
        for a_row, b_row in zip(self.rows, other.rows)
    ])
```

The `__add__()` method is responsible for overloading the + (plus sign) operator and here it allows us to add two matrices together. Only matrices of the same dimensions can be added together. This is a fairly simple operation that involves adding all matrix elements one by one to form a new matrix.

The `__sub__()` method is responsible for overloading the – (minus sign) operator that will be responsible for matrix subtraction. To subtract two matrices, we use a similar technique as in the – operator:

```python
def __sub__(self, other):
    if (
        len(self.rows) != len(other.rows) or
        len(self.rows[0]) != len(other.rows[0])
    ):
        raise ValueError("Matrix dimensions don't match")

    return Matrix([
        [a - b for a, b in zip(a_row, b_row)]
```

```
        for a_row, b_row in zip(self.rows, other.rows)
    ])
```

And the following is the last method we add to our class:

```python
def __mul__(self, other):
    if not isinstance(other, Matrix):
        raise TypeError(
            f"Don't know how to multiply {type(other)} with Matrix"
        )

    if len(self.rows[0]) != len(other.rows):
        raise ValueError(
            "Matrix dimensions don't match"
        )

    rows = [[0 for _ in other.rows[0]] for _ in self.rows]

    for i in range(len(self.rows)):
        for j in range(len(other.rows[0])):
            for k in range(len(other.rows)):
                rows[i][j] += self.rows[i][k] * other.rows[k][j]

    return Matrix(rows)
```

The last overloaded operator is the most complex one. This is the * operator, which is implemented through the __mul__() method. In linear algebra, matrices don't have the same multiplication operation as real numbers. Two matrices can be multiplied if the first matrix has a number of columns equal to the number of rows of the second matrix. The result of that operation is a new matrix where each element is a dot product of the corresponding row of the first matrix and the corresponding column of the second matrix.

Here we've built our own implementation of the matrix to present the idea of operators overloading. Although Python lacks a built-in type for matrices, you don't need to build them from scratch. The NumPy package is one of the best Python mathematical packages and among others provides native support for matrix algebra. You can easily obtain the NumPy package from PyPI.

Comparison to C++

One programming language where operator overloading is particularly common is C++. It is a statically typed OOP language that is nothing like Python. Python has OOP elements and some mechanisms that, in essence, are similar to those of C++. These are mainly the existence of classes and class inheritance together with the ability to overload operators. But the way these mechanisms are implemented within the language is completely different. And that's why comparing those two languages is so fascinating.

C++, in contrast to Python, has multiple coexisting polymorphism mechanisms. The main mechanism is through subtyping, which is also available in Python. The second major type of polymorphism in C++ is **ad hoc polymorphism** through **function overloading**. Python lacks a direct counterpart of that feature.

Function overloading in C++ allows you to have multiple implementations of the same function depending on input arguments. It means that you can have two functions or methods sharing the same name but having a different number of and/ or types of arguments. As C++ is a statically typed language, types of arguments are always known in advance and the choice of exact implementation happens at compile time.

To make it even more flexible, function overloading can be used together with operator overloading. The use case for such overloading coexistence can be better understood if we bring back the matrix multiplication use case. We know that two matrices can be multiplied together and we've learned how to do that in the previous section. But linear algebra also allows you to multiply a matrix with a scalar type like a real number. This operation results in a new matrix where every element has been multiplied by the scalar. In code, that would mean essentially another implementation of the multiplication operator.

In C++, you can simply provide multiple coexisting * operator overloading functions. The following is an example of C++ function signatures for overloaded operators that could allow various matrix and scalar multiplication implementations:

```
Matrix operator+(const Matrix& lhs, const Matrix& rhs)
Matrix operator+(const Matrix& lhs, const int& rhs)
Matrix operator+(const Matrix& lhs, const float& rhs)
Matrix operator+(const int& lhs, const Matrix& rhs)
Matrix operator+(const float& lhs, const Matrix& rhs)
```

Python is a dynamically typed language, and that's the main reason why it doesn't have function overloading as in C++. If we want to implement * operator overloading on the `Matrix` class that supports both matrix multiplication and scalar multiplication, we need to verify the operator input type at runtime. This can be done with the built-in `isinstance()` function as in the following example:

```python
def __mul__(self, other):
    if isinstance(other, Matrix):
        ...

    elif isinstance(other, Number):
        return Matrix([
            [item * other for item in row]
            for row in self.rows
        ])
    else:
        raise TypeError(f"Can't subtract {type(other)} from Matrix")
```

Another major difference is that C++ operator overloading is done through free functions instead of class methods, while in Python, the operator is always resolved from one operand's dunder method. This difference can again be displayed using an example of scalar implementation. The previous example allowed us to multiply a matrix by an integer number in the following form:

```python
Matrix([[1, 1], [2, 2]]) * 3
```

This will work because the overloaded operator implementation will be resolved from the left operand. On the other hand, the following expression will result in `TypeError`:

```python
3 * Matrix([1, 1], [2, 2])
```

In C++, you can provide multiple versions of operator overloading that cover all combinations of operand types for the * operator. In Python, the workaround for that problem is providing the __rmul__() method. This method is resolved from the right-side operand if the left-side __mul__() operator raises `TypeError`. Most infix operators have their right-side implementation alternatives. The following is an example of the __rmul__() method for the `Matrix` class that allows you to perform scalar multiplication with a right-hand side number argument:

```python
def __rmul__(self, other):
    if isinstance(other, Number):
        return self * other
```

As you see, it still requires the use of type evaluation through the `isinstance()` function, so operator overloading should be used very cautiously, especially if overloaded operators receive completely new meaning that is not in line with their original purpose.

The need to provide alternative overloaded implementations of the operator depending on the single operand type is usually a sign that the operator has lost its clear meaning. For instance, matrix multiplication and scalar multiplication are mathematically two distinct operations. They have different properties. For instance, scalar multiplication is cumulative while matrix multiplication isn't. Providing an overloaded operator for a custom class that has multiple internal implementations can quickly lead to confusion, especially in code that deals with math problems.

We were deliberately silent about the fact that Python actually has a dedicated matrix multiplication operator despite the fact that it doesn't have the built-in matrix type. That was just to better showcase the danger and complexities of overusing operator overloading. The dedicated operator for matrix multiplication is @ and actually, the potential confusion between scalar and matrix multiplication was one of the main reasons this operator was introduced.

In many programming languages, operator overloading can be considered a special case of function and method overloading and these usually come in a pair. Surprisingly, Python has operator overloading but doesn't offer real function and method overloading. It offers different patterns to fill that gap. We will discuss them in the next section.

Function and method overloading

A common feature of many programming languages is function and method overloading. It is another type of polymorphism mechanism. Overloading allows you to have multiple implementations of a single function by using different call signatures. Either a language compiler or interpreter is able to select a matching implementation based on the set of function call arguments provided. Function overloading is usually resolved based on:

- Function arity (number of parameters): Two function definitions can share a function name if their signatures expect a different number of parameters.

- Types of parameters: Two function definitions can share a function name if their signatures expect different types of parameters.

As already stated in the *Operator overloading* section, Python lacks an overloading mechanism for functions and methods other than operator overloading. If you define multiple functions in a single module that share the same name, the latter definition will always shadow all previous definitions.

If there is a need to provide several function implementations that behave differently depending on the type or number of arguments provided, Python offers several alternatives:

- Using methods and/or subclassing: Instead of relying on a function to distinguish the parameter type, you can bind it to a specific type by defining it as a method of that type.

- Using argument and keyword argument unpacking: Python allows for some flexibility regarding function signatures to support a variable number of arguments via *args and **kwargs patterns (also known as **variadic functions**).

- Using type checking: The isinstance() function allows us to test input arguments against specific types and base classes to decide how to handle them.

Of course, each of the above options has some limitations. Pushing function implementation directly to class definitions as methods will not make any sense if said method doesn't constitute unique object behavior. Argument and keyword argument unpacking can make function signatures vague and hard to maintain.

Very often the most reliable and readable substitute for function overloading in Python is simply type checking. We've already seen this technique in action when discussing operator overloading. Let's recall the __mul__() method that was able to distinguish between matrix and scalar multiplication:

```python
def __mul__(self, other):
    if isinstance(other, Matrix):
        ...

    elif isinstance(other, Number):
        ...

    else:
        raise TypeError(f"Can't subtract {type(other)} from Matrix")
```

As you can see, something that in a statically typed language would have to be done through function overloading, in Python can be resolved with a simple isinstance() call. That can be understood as an upside rather than a downside of Python. Still, this technique is convenient only for a small number of call signatures. When the number of supported types grows, it is often better to use more modular patterns. Such patterns rely on **single-dispatch functions**.

Single-dispatch functions

In situations when an alternative to function overloading is required and the number of alternative function implementations is really large, using multiple `if isinstance(...)` clauses can quickly get out of hand. Good design practice dictates writing small, single-purpose functions. One large function that branches over several types to handle input arguments differently is rarely a good design.

The Python Standard Library provides a convenient alternative. The `functools.singledispatch()` decorator allows you to register multiple implementations of a function. Those implementations can take any number of arguments but implementations will be dispatched depending on the type of the first argument. Single dispatch starts with a definition of a function that will be used by default for any non-registered type. Let's assume that we need a function that can output various variables in human-readable format for the purpose of a larger report being displayed in the console output. By default, we could use the f-string to denote a raw value in string format:

```python
from functools import singledispatch


@singledispatch
def report(value):
    return f"raw: {value}"
```

From there, we can start registering different implementations for various types using the `report.register()` decorator. That decorator is able to read function argument type annotations to register specific type handlers. Let's say we want datetime objects to be reported in ISO format:

```python
from datetime import datetime


@report.register
def _(value: datetime):
    return f"dt: {value.isoformat()}"
```

Note that we used the _ token as the actual function name. That serves two purposes. First, it is a convention for names of objects that are not supposed to be used explicitly. Second, if we used the `report` name instead, we would shadow the original function, thus losing the ability to access it and register new types.

Let's define a couple more type handlers:

```python
from numbers import Real

@report.register
def _(value: complex):
    return f"complex: {value.real}{value.imag:+}j"

@report.register
def _(value: Real):
    return f"real: {value:f}"
```

Note that typing annotations aren't necessary but we've used them as an element of good practice. If you don't want to use typing annotations, you can specify the registered type as the `register()` method argument as in the following example:

```python
@report.register(complex)
def _(value):
    return f"complex: {value.real}{value.imag:+}j"

@report.register(real)
def _(value):
    return f"real: {value:f}"
```

If we tried to verify the behavior of our collection of single-dispatch implementations in an interactive session, we would get an output like the following:

```python
>>> report(datetime.now())
'dt: 2020-12-12T00:22:31.690377'
>>> report(100-30j)
'complex: 100.0-30.0j'
>>> report(9001)
'real: 9001.000000'
>>> report("January")
'raw: January'
>>> for key, value in report.registry.items():
...     print(f"{key} -> {value}")
...
<class 'object'> -> <function report at 0x7fdfd6929a60>
<class 'datetime.datetime'> -> <function _ at 0x7fdfd69a5af0>
<class 'complex'> -> <function _ at 0x7fdfd6993d30>
<class 'float'> -> <function _ at 0x7fdfd6d7ab80>
<class 'int'> -> <function _ at 0x7fdfd6d7ab80>
```

As we see, the `report()` function is now an entry point to a collection of registered functions. Whenever it is called with an argument, it looks in the registry mapping stored in `report.registry`. There's always at least one key that maps the `object` type to the default implementation of the function.

Additionally, there is a variation of the single-dispatch mechanism dedicated to class methods. Methods always receive the current object instance as their first argument. That means the `functools.singledispatch()` decorator would not be effective as the first argument of methods is always the same type. The `functools.singledispatchmethod()` decorator keeps that calling convention in mind and allows you to register multiple type-specific implementations on methods as well. It works by resolving the first non-`self`, non-`cls` argument:

```python
from functools import singledispatchmethod

class Example:
    @singledispatchmethod
    def method(self, argument):
        pass

    @method.register
    def _(self, argument: float):
        pass
```

Remember that while the single-dispatch mechanism is a form of polymorphism that resembles function overloading, it isn't exactly the same. You cannot use it to provide several implementations of a function on multiple argument types and the Python Standard Library currently lacks such a multiple-dispatch utility.

Data classes

As we learned from the *Class instance initialization* section, the canonical way to declare class instance attributes is through assigning them in the `__init__()` method as in the following example:

```python
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Let's assume we are building a program that does some geometric computation and `Vector` is a class that allows us to hold information about two-dimensional vectors. We will display the data of the vectors on the screen and perform common mathematical operations, such as addition, subtraction, and equality comparison. We already know that we can use special methods and operator overloading to achieve that goal in a convenient way. We can implement our `Vector` class as follows:

```python
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        """Add two vectors using + operator"""
        return Vector(
            self.x + other.x,
            self.y + other.y,
        )

    def __sub__(self, other):
        """Subtract two vectors using - operator"""
        return Vector(
            self.x - other.x,
            self.y - other.y,
        )

    def __repr__(self):
        """Return textual representation of vector"""
        return f"<Vector: x={self.x}, y={self.y}>"

    def __eq__(self, other):
        """Compare two vectors for equality"""
        return self.x == other.x and self.y == other.y
```

The following is the interactive session example that shows how it behaves when used with common operators:

```python
>>> Vector(2, 3)
<Vector: x=2, y=3>
>>> Vector(5, 3) + Vector(1, 2)
<Vector: x=6, y=5>
>>> Vector(5, 3) - Vector(1, 2)
<Vector: x=4, y=1>
```

```
>>> Vector(1, 1) == Vector(2, 2)
False
>>> Vector(2, 2) == Vector(2, 2)
True
```

The preceding vector implementation is quite simple, but it involves a lot of code that could be avoided. Our Vector class is focused on data. Most of the behavior it provides is centered around creating new Vector instances through mathematic operations. It doesn't provide complex initialization nor custom attribute access patterns. Things like equality comparison, string representation, and attribute initialization will look very similar and repetitive for various classes focused on data.

If your program uses many similar simple classes focused on data that do not require complex initialization, you'll end up writing a lot of boilerplate code just for the __init__(), __repr__(), and __eq__() methods.

With the dataclasses module, we can make our Vector class code a lot shorter:

```python
from dataclasses import dataclass


@dataclass
class Vector:
    x: int
    y: int

    def __add__(self, other):
        """Add two vectors using + operator"""
        return Vector(
            self.x + other.x,
            self.y + other.y,
        )

    def __sub__(self, other):
        """Subtract two vectors using - operator"""
        return Vector(
            self.x - other.x,
            self.y - other.y,
        )
```

The dataclass class decorator reads attribute annotations of the Vector class and automatically creates the __init__(), __repr__(), and __eq__() methods. The default equality comparison assumes that the two instances are equal if all their respective attributes are equal to each other.

But that's not all. Data classes offer many useful features. They can easily be made compatible with other Python protocols, too. Let's assume we want our `Vector` class instances to be immutable. Thanks to this, they could be used as dictionary keys and as content values in sets. You can do this by simply adding a `frozen=True` argument to the `dataclass` decorator, as in the following example:

```python
from dataclasses import dataclass


@dataclass(frozen=True)
class FrozenVector:
    x: int
    y: int
```

Such a frozen `Vector` data class becomes completely immutable, so you won't be able to modify any of its attributes. You can still add and subtract two `Vector` instances as in our example; these operations simply create new `Vector` objects.

We've learned already about the dangers of assigning default values to class attributes in the main class body instead of the `__init__()` function. The `dataclass` module offers a useful alternative through the `field()` constructor. This constructor allows you to specify both mutable and immutable default values for data class attributes in a sane and secure way without risking leaking the state between class instances. Static and immutable default values are provided using the `field(default=value)` call. The mutable values should always be passed by providing a type constructor using the `field(default_factory=constructor)` call. The following is an example of a data class with two attributes that have their default values assigned through the `field()` constructor:

```python
from dataclasses import dataclass, field


@dataclass
class DataClassWithDefaults:
    immutable: str = field(default="this is static default value")
    mutable: list = field(default_factory=list)
```

Once a data class attribute has its default assigned, the corresponding initialization argument for that field becomes optional. The following transcript presents various ways of initializing `DataClassWithDefaults` class instances:

```
>>> DataClassWithDefaults()
DataClassWithDefaults(immutable='this is static default value',
mutable=[])
>>> DataClassWithDefaults("This is immutable")
DataClassWithDefaults(immutable='This is immutable', mutable=[])
>>> DataClassWithDefaults(None, ["this", "is", "list"])
DataClassWithDefaults(immutable=None, mutable=['this', 'is', 'list'])
```

Data classes are similar in nature to structs in C or Go. Their main purpose is to hold data and provide shortcuts for the otherwise tedious initialization of instance attributes. But they should not be used as a basis for every possible custom class. If your class isn't meant to represent the data, and/or requires custom or complex state initialization, you should rather use the default way of initialization: through the __init__() method.

Python is not only about OOP. It supports other programming paradigms as well. One of those paradigms is functional programming, which concentrates on the evaluation of functions. Pure functional programming languages are usually drastically different than their OOP counterparts. But multiparadigm programming languages try to take the best of many programming styles. That's also true for Python. In the next section, we will review a few elements of Python that support functional programming. You will soon notice that this paradigm in Python is actually built over the foundation laid by OOP.

Functional programming

One of the great things about programming in Python is that you are never constrained to a single way of thinking about your programs. There are always various ways to solve a given problem, and sometimes the best one requires an approach that is slightly different from the one that would be the most obvious. Sometimes, this approach requires the use of declarative programming. Fortunately, Python, with its rich syntax and large standard library, offers features of functional programming, and functional programming is one of the main paradigms of declarative programming.

Functional programming is a paradigm where the program flow is achieved mainly through the evaluation of (mathematical) functions rather than through a series of steps that change the state of the program. Purely functional programs avoid the changing of state (side effects) and the use of mutable data structures.

One of the best ways to better understand the general concept of functional programming is by familiarizing yourself with the basic terms of functional programming:

- **Side effects**: A function is said to have a side effect if it modifies the state outside of its local environment. In other words, a side effect is any observable change outside of the function scope that happens as a result of a function call. An example of such side effects could be the modification of a global variable, the modification of an attribute of an object that is available outside of the function scope, or saving data to some external service. Side effects are the core of the concept of OOP, where class instances are objects that are used to encapsulate the state of an application, and methods are functions bound to those objects that are supposed to manipulate the state of these objects. Procedural programming also heavily relies on side effects.

- **Referential transparency**: When a function or expression is referentially transparent, it can be replaced with the value that corresponds to its output without changing the behavior of the program. So, a lack of side effects is a requirement for referential transparency, but not every function that lacks side effects is a referentially transparent function. For instance, Python's built-in `pow(x, y)` function is referentially transparent, because it lacks side effects, and for every x and y argument, it can be replaced with the value of x^y. On the other hand, the `datetime.now()` constructor method of the `datetime` type does not seem to have any observable side effects but will return a different value every time it is called. So, it is referentially opaque.

- **Pure functions**: A pure function is a function that does not have any side effects and that always returns the same value for the same set of input arguments. In other words, it is a function that is referentially transparent. Every mathematical function is, by definition, a pure function. Analogously, a function that leaves a trace of its execution for the outside world (for instance, by modifying received objects) is not a pure function.

- **First-class functions**: Language is said to contain first-class functions if functions in this language can be treated as any other value or entity. First-class functions can be passed as arguments to other functions, returned as function return values, and assigned to variables. In other words, a language that has first-class functions is a language that treats functions as first-class citizens. Functions in Python are first-class functions.

Using these concepts, we could describe a purely functional language as a language that:

- Has first-class functions
- Is concerned only with pure functions
- Avoids any state modification and side effects

Python, of course, is not a purely functional programming language, and it would be really hard to imagine a useful Python program that uses only pure functions without any side-effects. On the other hand, Python offers a large variety of features that, for years, were only accessible in purely functional languages, like:

- Lambda functions and first-class functions
- `map()`, `filter()`, and `reduce()` functions
- Partial objects and functions
- Generators and generator expressions

Those features make it possible to write substantial amounts of Python code in a functional way, even though Python isn't purely functional.

Lambda functions

Lambda functions are a very popular programming concept that is especially profound in functional programming. In other programming languages, lambda functions are sometimes known as anonymous functions, lambda expressions, or function literals. Lambda functions are anonymous functions that don't have to be bound to any identifier (variable).

> At some point in Python 3's development, there was a heated discussion about removing the feature of lambda functions together with the `map()`, `filter()`, and `reduce()` functions. You can learn more about Guido van Rossum's article about reasons for removing those features at `https://www.artima.com/weblogs/viewpost.jsp?thread=98196`.

Lambda functions in Python can be defined only using expressions. The syntax for lambda functions is as follows:

```
lambda <arguments>: <expression>
```

The best way to present the syntax of lambda functions is by comparing a "normal" function definition with its anonymous counterpart. The following is a simple function that returns the area of a circle of a given radius:

```python
import math

def circle_area(radius):
    return math.pi * radius ** 2
```

The same function expressed as a lambda function would take the following form:

```python
lambda radius: math.pi * radius ** 2
```

Lambda functions are anonymous, but it doesn't mean they cannot be referred to using an identifier. Functions in Python are first-class objects, so whenever you use a function name, you're actually using a variable that is a reference to the function object. As with any other function, lambda functions are first-class citizens, so they can also be assigned to a new variable. Once assigned to a variable, they are seemingly indistinguishable from other functions, except for some metadata attributes. The following transcripts from interactive interpreter sessions illustrate this:

```python
>>> import math
>>> def circle_area(radius):
...     return math.pi * radius ** 2
...
>>> circle_area(42)
5541.769440932395
>>> circle_area
<function circle_area at 0x10ea39048>
>>> circle_area.__class__
<class 'function'>
>>> circle_area.__name__
'circle_area'

>>> circle_area = lambda radius: math.pi * radius ** 2
>>> circle_area(42)
5541.769440932395
>>> circle_area
```

```
<function <lambda> at 0x10ea39488>
>>> circle_area.__class__
<class 'function'>
>>> circle_area.__name__
'<lambda>'
```

The main use for lambda expressions is to define contextual one-off functions that won't have to be reused elsewhere. To better understand their potential, let's imagine that we have an application that stores information about people. To represent a record of a person's data, we could use the following data class:

```
from dataclasses import dataclass


@dataclass
class Person:
    age: int
    weight: int
    name: str
```

Now let's imagine that we have a set of such records and we want to sort them by different fields. Python provides a sorted() function that is able to sort any list as long as elements can be compared with at least "less than" comparison (the < operator). We could define custom operator overloading on the Person class, but we would have to know in advance what field our records will be sorted on.

Thankfully, the sorted() function accepts the key keyword argument, which allows you to specify a function that will transform every element of the input into a value that can be naturally sorted by the function. Lambda expressions allow you to define such sorting keys on demand. For instance, sorting people by age can be done using the following call:

```
sorted(people, key=lambda person: person.age)
```

The above behavior of the sorted() function presents a common pattern of allowing code to accept a callable argument that resolves some injected behavior. Lambda expressions are often a convenient way of defining such behaviors.

The map(), filter(), and reduce() functions

The map(), filter(), and reduce() functions are three built-in functions that are most often used in conjunction with lambda functions. They are commonly used in functional-style Python programming because they allow us to declare transformations of any complexity, while simultaneously avoiding side effects.

In Python 2, all three functions were available as default built-in functions that did not require additional imports. In Python 3, the `reduce()` function was moved to the `functools` module, so it requires an additional import.

`map(func, iterable, ...)` applies the `func` function argument to every item of `iterable`. You can pass more iterables to the `map()` function. If you do so, `map()` will consume elements from each iterable simultaneously. The `func` function will receive as many arguments as there are iterables on every map step. If iterables are of different sizes, `map()` will stop when the shortest one is exhausted. It is worth remembering that `map()` does not evaluate the whole result at once, but returns an iterator so that every result item can be evaluated only when it is necessary.

The following is an example of `map()` being used to calculate the squares of the first 10 integers staring from `0`:

```
>>> map(lambda x: x**2, range(10))
<map object at 0x10ea09cf8>
>>> list(map(lambda x: x**2, range(10)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

The following is an example of the `map()` function being used over multiple iterables of different sizes:

```
>>> mapped = list(map(print, range(5), range(4), range(5)))
0 0 0
1 1 1
2 2 2
3 3 3
>>> mapped
[None, None, None, None]
```

`filter(func, iterable)` works similarly to `map()` by evaluating input elements one by one. Unlike `map()`, the `filter()` function does not transform input elements into new values, but allows us to filter out those input values that do not meet the predicate defined by the `func` argument. The following are examples of the `filter()` function's usage:

```
>>> evens = filter(lambda number: number % 2 == 0, range(10))
>>> odds = filter(lambda number: number % 2 == 1, range(10))
>>> print(f"Even numbers in range from 0 to 9 are: {list(evens)}")
Even numbers in range from 0 to 9 are: [0, 2, 4, 6, 8]
>>> print(f"Odd numbers in range from 0 to 9 are: {list(odds)}")
Odd numbers in range from 0 to 9 are: [1, 3, 5, 7, 9]
```

```
>>> animals = ["giraffe", "snake", "lion", "squirrel"]
>>> animals_s = filter(lambda animal: animal.startswith('s'), animals)
>>> print(f"Animals that start with letter 's' are: {list(animals_s)}")
Animals that start with letter 's' are: ['snake', 'squirrel']]
```

The `reduce(func, iterable)` function works in completely the opposite way to `map()`. As the name suggests, this function can be used to reduce an iterable to a single value. Instead of taking items of `iterable` and mapping them to the `func` return values in one-by-one fashion, it cumulatively performs the operation specified by `func` over all `iterable` items. So, for the following inputs of `reduce()`:

```
reduce(func, [a, b, c, d])
```

The return value would be equal to:

```
func(func(func(a, b), c), d)
```

Let's consider the following example of `reduce()` calls being used to sum values of elements contained in various iterable objects:

```
>>> from functools import reduce
>>> reduce(lambda a, b: a + b, [2, 2])
4
>>> reduce(lambda a, b: a + b, [2, 2, 2])
6
>>> reduce(lambda a, b: a + b, range(100))
4950
```

One interesting aspect of `map()` and `filter()` is that they can work on infinite sequences. Of course, evaluating an infinite sequence to a `list` type or trying to ordinarily loop over such a sequence will result in a program that never ends. The `count()` function from `itertools` is an example of a function that returns infinite iterables. It simply counts from 0 to infinity. If you try to loop over it as in the following example, your program will never stop:

```
from itertools import count

for i in count():
    print(i)
```

However, the return values of `map()` and `filter()` are iterators. Instead of using a `for` loop, you can consume consecutive elements of the iterator using the `next()` function. Let's take a look again at our previous `map()` call that generated consecutive integer squares starting from 0:

```
map(lambda x: x**2, range(n))
```

The `range()` function returns a bounded iterable of n items. If we don't know how many items we want to generate, we can simply replace it with `count()`:

```
map(lambda x: x**2, count())
```

From now on we can start consuming consecutive squares. We can't use a `for` loop because that would never end. But we can use `next()` numerous times and consume items one at a time:

```
sequence = map(lambda x: x**2, count())
next(sequence)
next(sequence)
next(sequence)
...
```

Unlike the `map()` and `filter()` functions, the `reduce()` function needs to evaluate all input items in order to return its value, as it does not yield intermediary results. This means that it cannot be used on infinite sequences.

Partial objects and partial functions

Partial objects are loosely related to the concept of partial functions in mathematics. A partial function is a generalization of a mathematical function in a way that isn't forced to map every possible input value range (domain) to its results. In Python, partial objects can be used to slice the possible input range of a given function by setting some of its arguments to a fixed value.

In the previous sections, we used the `x ** 2` expression to get the square value of x. Python provides a built-in function called `pow(x, y)` that can calculate any power of any number. So, our `lambda x: x ** 2` expression is a partial function of the `pow(x, y)` function, because we have limited the domain values for y to a single value, 2. The `partial()` function from the `functools` module provides an alternative way to easily define such partial functions without the need for lambda expressions, which can sometimes become unwieldy.

Let's say that we now want to create a slightly different partial function out of `pow()`. Last time, we generated squares of consecutive numbers. Now, let's narrow the domain of other input arguments and say we want to generate consecutive powers of the number two—so, 1, 2, 4, 8, 16, and so on.

The signature of a partial object constructor is `partial(func, *args, **keywords)`. The partial object will behave exactly like `func`, but its input arguments will be pre-populated with `*args` (starting from the leftmost) and `**keywords`. The `pow(x, y)` function does not support keyword arguments, so we have to pre-populate the leftmost `x` argument as follows:

```
>>> from functools import partial
>>> powers_of_2 = partial(pow, 2)
>>> powers_of_2(2)
4
>>> powers_of_2(5)
32
>>> powers_of_2(10)
1024
```

Note that you don't need to assign your partial object to any identifier if you don't want to reuse it. You can successfully use it to define one-off functions in the same way that you would use lambda expressions.

> The `itertools` module is a treasury of helpers and utilities for iterating over any type of iterable objects in various ways. It provides various functions that, among other things, allow us to cycle containers, group their contents, split iterables into chunks, and chain multiple iterables into one. Every function in that module returns iterators. If you are interested in functional-style programming in Python, you should definitely familiarize yourself with this module. You can find the documentation of the `itertools` module at https://docs.python.org/3/library/itertools.html.

Generators

Generators provide an elegant way to write simple and efficient code for functions that return a sequence of elements. Based on the `yield` statement, they allow you to pause a function and return an intermediate result. The function saves its execution context and can be resumed later, if necessary.

For instance, the function that returns consecutive numbers of the Fibonacci sequence can be written using a generator syntax. The following code is an example that was taken from the *PEP 255 (Simple Generators)* document:

```python
def fibonacci():
    a, b = 0, 1
    while True:
        yield b
        a, b = b, a + b
```

You can retrieve new values from generators as if they were iterators, so using the next() function or for loops:

```python
>>> fib = fibonacci()
>>> next(fib)
1
>>> next(fib)
1
>>> next(fib)
2
>>> for item in fibonacci():
...     print(item)
...     if item > 10:
...         break
...
1
1
2
3
5
8
13
```

Our fibonacci() function returns a generator object, a special iterator that knows how to save the execution context. It can be called indefinitely, yielding the next element of the sequence each time. The syntax is concise, and the infinite nature of the algorithm does not disturb the readability of the code. It does not have to provide a way to make the function stoppable. In fact, it looks similar to how the sequence generating function would be designed in pseudo code.

In many cases, the resources required to process one element are less than the resources required to store whole sequences. Therefore, they can be kept low, making the program more efficient. For instance, the Fibonacci sequence is infinite, and yet the generator that generates it does not require an infinite amount of memory to provide the values one by one and, theoretically, could work *ad infinitum*. A common use case is to stream data buffers with generators (for example, from files). They can be paused, resumed, and stopped whenever necessary at any stage of the data processing pipeline without any need to load whole datasets into the program's memory.

In functional programming, generators can be used to provide a stateful function that otherwise would require saving intermediary results as side effects as if it were a stateless function.

Generator expressions

Generator expressions are another syntax element that allows you to write code in a more functional way. Its syntax is similar to comprehensions that are used with dictionary, set, and list literals. A generator expression is denoted by parentheses, like in the following example:

```
(item for item in iterable_expression)
```

Generator expressions can be used as input arguments in any function that accepts iterables. They also allow `if` clauses to filter specific elements the same way as list, dictionary, and set comprehensions. This means that you can often replace complex `map()` and `filter()` constructions with more readable and compact generator expressions.

Syntactically, generator expressions are no different from any other comprehension expressions. Their main advantage is that they evaluate only one item at a time. So, if you process an arbitrarily long iterable expression, a generator expression may be a good fit as it doesn't need to fit the whole collection of intermediary results into program memory.

Lambdas, map, reduce, filter, partial functions, and generators are focused on presenting program logic as an evaluation of function call expressions. Another important element of functional programming is having first-class functions. In Python, all functions are objects and like any other object, they can be inspected and modified at runtime. It allows for a useful syntax feature called **function decorators.**

Decorators

The decorator is generally a callable expression that accepts a single argument when called (it will be the decorated function) and returns another callable object.

> Prior to Python 3.9, only named expressions could be used with a dedicated decorator syntax. Starting from Python 3.9, any expression is a valid target for a dedicated decorator syntax, including lambda expressions.

While decorators are often discussed in the scope of methods and functions, they are not limited to them. In fact, anything that is callable (any object that implements the `__call__` method is considered callable) can be used as a decorator, and often, objects returned by them are not simple functions but are instances of more complex classes that are implementing their own `__call__` method.

The decorator syntax is simply syntactic sugar. Consider the following decorator usage:

```python
@some_decorator
def decorated_function():
    pass
```

This can always be replaced by an explicit decorator call and function reassignment:

```python
def decorated_function():
    pass
decorated_function = some_decorator(decorated_function)
```

However, the latter is less readable and also very hard to understand if multiple decorators are used on a single function.

> A decorator does not even need to return a callable!
>
> As a matter of fact, any function can be used as a decorator, because Python does not enforce the return type of decorators. So, using some function as a decorator that accepts a single argument but does not return a callable object, let's say `str`, is completely valid in terms of syntax. This will eventually fail if you try to call an object that's been decorated this way. This part of the decorator syntax creates a field for some interesting experimentation.

Decorators are elements of the programming language inspired by aspect-oriented programming and the decorator design pattern. The main use case is to conveniently enhance an existing function implementation with extra behavior coming from other aspects of your application.

Consider the following example, taken from the Flask framework documentation:

```python
@app.route('/secret_page')
@login_required
def secret_page():
    pass
```

`secret_page()` is a view function that presumably is supposed to return a secret page. It is decorated with two decorators. `app.route()` assigns a URI route to the view function and `login_required()` enforces user authentication.

According to the single-responsibility principle, functions should be rather small and single-purpose. In our Flask application, the `secret_page()` view function would be responsible for preparing the HTTP response that can be later rendered in a web browser. It probably shouldn't deal with things like parsing HTTP requests, verifying user credentials, and so on.

As the names suggests, the `secret_page()` function returns something that is secret, and shouldn't be visible to anyone. Verifying user credentials isn't part of the view function's responsibility but it is part of the general idea of "a secret page." The `@login_required` decorator allows you to bring the aspect of user authentication close to the view function. It makes the application more concise and the intent of the programmer more readable.

Let's look further at the actual example of the `@login_required` decorator from the Flask framework documentation:

```python
from functools import wraps
from flask import g, request, redirect, url_for


def login_required(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if g.user is None:
            return redirect(url_for('login', next=request.url))
        return f(*args, **kwargs)
    return decorated_function
```

> The `@wraps` decorator allows you to copy decorated function metadata like name and type annotations. It is a good practice to use the `@wraps` decorator in your own decorators as it eases debugging and gives access to original function type annotations.

As we can see, this decorator returns a new `decorated_function()` function that at first verifies if the global `g` object has a valid user assigned. That's a common way of testing whether the user has been authenticated in Flask. If the test succeeds, the decorated function calls the original function by returning `f(*args, **kwargs)`. If the login test fails, the decorated function will redirect the browser to the login page.

As we can see, the `login_required()` decorator conveys a little bit more than simple check-or-fail behavior. That makes decorators a great mechanism of code reuse. The login requirement may be a common aspect of applications, but the implementation of that aspect can change over time. Decorators offer a convenient way to pack such aspects into portable behaviors that can be easily added on top of existing functions.

We will use and explain decorators in more details in *Chapter 8*, *Elements of Metaprogramming*, where we will discuss decorators as a metaprogramming technique.

Enumerations

There are common programming features that are found in many programming languages regardless of the dominant programming paradigm. One such feature is enumerated types that have a finite number of named values. They are especially useful for encoding a closed set of values for variables or function arguments.

One of the special handy types found in the Python Standard Library is the `Enum` class from the `enum` module. This is a base class that allows you to define symbolic enumerations, similar in concept to the enumerated types found in many other programming languages (C, C++, C#, Java, and many more) that are often denoted with the `enum` keyword.

In order to define your own enumeration in Python, you will need to subclass the `Enum` class and define all enumeration members as class attributes. The following is an example of a simple Python `enum`:

```python
from enum import Enum

class Weekday(Enum):
    MONDAY = 0
    TUESDAY = 1
    WEDNESDAY = 2
```

```
    THURSDAY = 3
    FRIDAY = 4
    SATURDAY = 5
    SUNDAY = 6
```

The Python documentation defines the following nomenclature for enum:

- **enumeration** or **enum**: This is the subclass of the Enum base class. Here, it would be Weekday.

- **member**: This is the attribute you define in the Enum subclass. Here, it would be Weekday.MONDAY, Weekday.TUESDAY, and so on.

- **name**: This is the name of the Enum subclass attribute that defines the member. Here, it would be MONDAY for Weekday.MONDAY, TUESDAY for Weekday.TUESDAY, and so on.

- **value**: This is the value assigned to the Enum subclass attribute that defines the member. Here, for Weekday.MONDAY it would be one, for Weekday.TUESDAY it would be two, and so on.

You can use any type as the enum member value. If the member value is not important in your code, you can even use the auto() type, which will be replaced with automatically generated values. Here is a similar example written with the use of auto:

```python
from enum import Enum, auto

class Weekday(Enum):
    MONDAY = auto()
    TUESDAY = auto()
    WEDNESDAY = auto()
    THURSDAY = auto()
    FRIDAY = auto()
    SATURDAY = auto()
    SUNDAY = auto()
```

Enumerations in Python are really useful in every place where some variable can take only a finite number of values/choices. For instance, they can be used to define the status of objects, as shown in the following example:

```python
from enum import Enum, auto

class OrderStatus(Enum):
    PENDING = auto()
    PROCESSING = auto()
```

```
        PROCESSED = auto()

class Order:
    def __init__(self):
        self.status = OrderStatus.PENDING

    def process(self):
        if self.status == OrderStatus.PROCESSED:
            raise ValueError(
                ""Can't process order that has ""
                ""been already processed""
            )

        self.status = OrderStatus.PROCESSING
        ...
        self.status = OrderStatus.PROCESSED
```

Another use case for enumerations is storing selections of non-exclusive choices. This is something that is often implemented using bit flags and bit masks in languages where the bit manipulation of numbers is very common, like C. In Python, this can be done in a more expressive and convenient way using the `Flag` base enumeration class:

```
from enum import Flag, auto

class Side(Flag):
    GUACAMOLE = auto()
    TORTILLA = auto()
    FRIES = auto()
    BEER = auto()
    POTATO_SALAD = auto()
```

You can combine such flags using bitwise operators (the | and & operators) and test for flag membership with the `in` keyword. Here are some examples of a `Side` enumeration:

```
>>> mexican_sides = Side.GUACAMOLE | Side.BEER | Side.TORTILLA
>>> bavarian_sides = Side.BEER | Side.POTATO_SALAD
>>> common_sides = mexican_sides & bavarian_sides
>>> Side.GUACAMOLE in mexican_sides
True
>>> Side.TORTILLA in bavarian_sides
False
>>> common_sides
<Side.BEER: 8>
```

Symbolic enumerations share some similarity with dictionaries and named tuples because they all map names/keys to values. The main difference is that the `Enum` definition is immutable and global. It should be used whenever there is a closed set of possible values that can't change dynamically during program runtime, and especially if that set should be defined only once and globally. Dictionaries and named tuples are data containers. You can create as many instances of them as you like.

Summary

In this chapter, we've looked at the Python language through the prism of different programming paradigms. Whenever it was sensible, we've tried to see how it compares to other programming languages that share similar features to see both strengths and weaknesses of Python.

We went pretty deep into the details of object-oriented programming concepts and extended our knowledge of supplementary paradigms like functional programming, so we are now fully prepared to start discussing topics on structuring and architecting whole applications.

The next chapter will cover that pretty extensively as it will be fully dedicated to various design patterns and methodologies.

5

Interfaces, Patterns, and Modularity

In this chapter, we will dive deep into the realm of design patterns through the lens of interfaces, patterns, and modularity. We've already neared this realm when introducing the concept of programming idioms. Idioms can be understood as small and well-recognized programming patterns for solving small problems. The key characteristic of a programming idiom is that it is specific to a single programming language. While idioms can often be ported to a different language, it is not guaranteed that the resulting code will feel natural to "native" users of that programming language.

Idioms generally are concerned with small programming constructs—usually a few lines of code. Design patterns, on the other hand, deal with much larger code structures—functions and classes. They are also definitely more ubiquitous. Design patterns are reusable solutions to many common design problems appearing in software engineering. They are often language-agnostic and thus can be expressed using many programming languages.

In this chapter, we will look at a quite unusual take on the topic of design patterns. Many programming books start by going back to the unofficial origin of software design patterns—the *Design Patterns: Elements of Reusable Object-Oriented Software* book by Gamma, Vlissides, Helm, and Johnson. What usually follows is a lengthy catalog of classic design patterns with more or less idiomatic examples of their Python implementation. Singletons, factories, adapters, flyweights, bridges, visitors, strategies, and so on and so forth.

There are also countless web articles and blogs doing exactly the same, so if you are interested in learning the classic design patterns, you shouldn't have any problems finding resources online.

> If you are interested in learning about the implementation of "classic" design patterns in Python, you can visit the `https://python-patterns.guide` site. It provides a comprehensive catalog of design patterns together with Python code examples.

Instead, we will focus on two key "design pattern enablers":

- Interfaces
- Inversion of control and dependency injectors

These two concepts are "enablers" because without them we wouldn't even have proper language terms to talk about design patterns. By discussing the topic of interfaces and inversion of control, we will be able to better understand what the challenges are for building modular applications. And only by deeply understanding those challenges will we be able to figure out why we actually need patterns.

We will of course use numerous classic design patterns on the way, but we won't focus on any specific pattern.

Technical requirements

The following are Python packages that are mentioned in this chapter that you can download from PyPI:

- `zope.interface`
- `mypy`
- `redis`
- `flask`
- `injector`
- `flask-injector`

Information on how to install packages is included in *Chapter 2*, *Modern Python Development Environments*.

The code files for this chapter can be found at `https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition/tree/main/Chapter%205`.

Interfaces

Broadly speaking, an **interface** is an intermediary that takes part in the interaction between two entities. For instance, the interface of a car consists mainly of the steering wheel, pedals, gear stick, dashboard, knobs, and so on. The interface of a computer traditionally consists of a mouse, keyboard, and display.

In programming, interface may mean two things:

- The overall shape of the interaction plane that code can have
- The abstract definition of possible interactions with the code that is intentionally separated from its implementation

In the spirit of the first meaning, the interface is a specific combination of symbols used to interact with the unit of code. The interface of a function, for instance, will be the name of that function, its input arguments, and the output it returns. The interface of an object will be all of its methods that can be invoked and all the attributes that can be accessed.

Collections of units of code (functions, objects, classes) are often grouped into libraries. In Python, libraries take the form of modules and packages (collections of modules). They also have interfaces. Contents of modules and packages usually can be used in various combinations and you don't have to interact with all of their contents. That makes them programmable applications, and that's why interfaces of libraries are often referred to as **Application Programming Interfaces** (**APIs**).

This meaning of interface can be expanded to other elements of the computing world. Operating systems have interfaces in the form of filesystems and system calls. Web and remote services have interfaces in the form of communication protocols.

The second meaning of interface can be understood as the formalization of the former. Here interface is understood as a contract that a specific element of the code declares to fulfill. Such a formal interface can be extracted from the implementation and can live as a standalone entity. This gives the possibility to build applications that depend on a specific interface but don't care about the actual implementation, as long as it exists and fulfills the contract.

This formal meaning of interface can also be expanded to larger programming concepts:

- **Libraries:** The C programming language defines the API of its standard library, also known as the **ISO C Library**. Unlike Python, the C standard library has numerous implementations. For Linux, the most common is probably the **GNU C Library** (**glibc**), but it has alternatives like **dietlibc** or **musl**. Other operating systems come with their own ISO C Library implementations.

- **Operating System:** The **Portable Operating System Interface** (**POSIX**) is a collection of standards that define a common interface for operating systems. There are many systems that are certified to be compliant with that standard (macOS and Solaris to name a couple). There are also operating systems that are mostly compliant (Linux, Android, OpenBSD, and many more). Instead of using the term "POSIX compliance," we can say that those systems implement the POSIX interface.

- **Web services: OpenID Connect** (**OIDC**) is an open standard for authentication and an authorization framework based on the **OAuth 2.0** protocol. Services that want to implement the OIDC standard must provide specific well-defined interfaces described in this standard.

Formal interfaces are an extremely important concept in object-oriented programming languages. In this context, the interface abstracts either the form or purpose of the modeled object. It usually describes a collection of methods and attributes that a class should have to implement with the desired behavior.

In a purist approach, the definition of interface does not provide any usable implementation of methods. It just defines an explicit contract for any class that wishes to implement the interface. Interfaces are often composable. This means that a single class can implement multiple interfaces at once. In this way, interfaces are the key building block of design patterns. A single design pattern can be understood as a composition of specific interfaces. Similar to interfaces, design patterns do not have an inherent implementation. They are just reusable scaffolding for developers to solve common problems.

Python developers prefer **duck typing** over explicit interface definitions but having well-defined interaction contracts between classes can often improve the overall quality of the software and reduce the area of potential errors. For instance, creators of a new interface implementation get a clear list of methods and attributes that a given class needs to expose. With proper implementation, it is impossible to forget about a method that is required by a given interface.

Support for an abstract interface is the cornerstone of many statically typed languages. Java, for instance, has traits that are explicit declarations that a class implements a specific interface. This allows Java programmers to achieve polymorphism without type inheritance, which sometimes can become problematic. Go, on the other hand, doesn't have classes and doesn't offer type inheritance, but interfaces in Go allow for selected object-oriented patterns and polymorphism without type inheritance. For both those languages, interfaces are like an explicit version of duck typing behavior—Java and Go use interfaces to verify type safety at compile time, rather than using duck typing to tie things together at runtime.

Python has a completely different typing philosophy than these languages, so it does not have native support for interfaces verified at compile time. Anyway, if you would like to have more explicit control of application interfaces, there is a handful of solutions to choose from:

- Using a third-party framework like **zope.interface** that adds a notion of interfaces
- Using **Abstract Base Classes** (**ABCs**)
- Leveraging typing annotation, `typing.Protocol`, and static type analyzers.

We will carefully review each of those solutions in the following sections.

A bit of history: zope.interface

There are a few frameworks that allow you to build explicit interfaces in Python. The most notable one is a part of the **Zope** project. It is the `zope.interface` package. Although, nowadays, Zope is not as popular as it used to be a decade ago, the `zope.interface` package is still one of the main components of the still popular **Twisted** framework. `zope.interface` is also one of the oldest and still active interface frameworks commonly used in Python. It predates mainstream Python features like ABCs, so we will start from it and later see how it compares to other interface solutions.

> The `zope.interface` package was created by Jim Fulton to mimic the features of Java interfaces at the time of its inception.

The interface concept works best for areas where a single abstraction can have multiple implementations or can be applied to different objects that probably shouldn't be tangled with inheritance structure. To better present this idea, we will take the example of a problem that can deal with different entities that share some common traits but aren't exactly the same thing.

We will try to build a simple collider system that can detect collisions between multiple overlapping objects. This is something that could be used in a simple game or simulation. Our solution will be rather trivial and inefficient. Remember that the goal here is to explore the concept of interfaces and not to build a bulletproof collision engine for a blockbuster game.

The algorithm we will use is called **Axis-Aligned Bounding Box** (**AABB**). It is a simple way to detect a collision between two axis-aligned (no rotation) rectangles. It assumes that all elements that will be tested can be constrained with a rectangular bounding box. The algorithm is fairly simple and needs to compare only four rectangle coordinates:
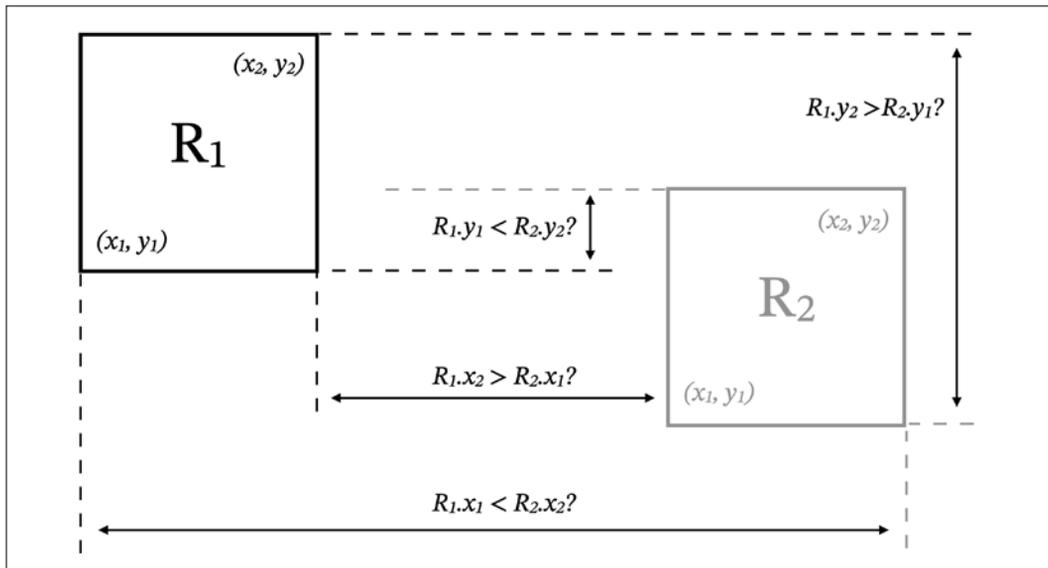

Figure 5.1: Rectangle coordinate comparisons in the AABB algorithm

We will start with a simple function that checks whether two rectangles overlap:

```python
def rects_collide(rect1, rect2):
    """Check collision between rectangles

    Rectangle coordinates:
            ┌──────(x2, y2)
            │       │
        (x1, y1) ──┘
    """
    return (
        rect1.x1 < rect2.x2 and
        rect1.x2 > rect2.x1 and
        rect1.y1 < rect2.y2 and
        rect1.y2 > rect2.y1
    )
```

We haven't defined any typing annotations but from the above code, it should be clearly visible that we expect both arguments of the `rects_collide()` function to have four attributes: `x1, y1, x2, y2`. These correspond to the coordinates of the lower-left and upper-right corners of the bounding box.

Having the `rects_collide()` function, we can define another function that will detect all collisions within a batch of objects. It can be as simple as follows:

```python
import itertools

def find_collisions(objects):
    return [
        (item1, item2)
        for item1, item2
        in itertools.combinations(objects, 2)
        if rects_collide(
            item1.bounding_box,
            item2.bounding_box
        )
    ]
```

What is left is to define some classes of objects that can be tested together against collisions. We will model a few different shapes: a square, a rectangle, and a circle. Each shape is different so will have a different internal structure. There is no sensible class that we could make a common ancestor. To keep things simple, we will use dataclasses and properties. The following are all initial definitions:

```python
from dataclasses import dataclass


@dataclass
class Square:
    x: float
    y: float
    size: float

    @property
    def bounding_box(self):
        return Box(
            self.x,
            self.y,
            self.x + self.size,
            self.y + self.size
        )


@dataclass
class Rect:
    x: float
    y: float
    width: float
    height: float

    @property
    def bounding_box(self):
        return Box(
            self.x,
            self.y,
            self.x + self.width,
            self.y + self.height
        )


@dataclass
class Circle:
    x: float
```

```
    y: float
    radius: float

    @property
    def bounding_box(self):
        return Box(
            self.x - self.radius,
            self.y - self.radius,
            self.x + self.radius,
            self.y + self.radius
        )
```

The only common thing about those classes (apart from being dataclasses) is the bounding_box property that returns the Box class instance. The Box class is also a dataclass:

```
@dataclass
class Box:
    x1: float
    y1: float
    x2: float
    y2: float
```

Definitions of dataclasses are quite simple and don't require explanation. We can test if our system works by passing a bunch of instances to the find_collisions() function as in the following example:

```
for collision in find_collisions([
    Square(0, 0, 10),
    Rect(5, 5, 20, 20),
    Square(15, 20, 5),
    Circle(1, 1, 2),
]):
    print(collision)
```

If we did everything right, the above code should yield the following output with three collisions:

```
(Square(x=0, y=0, size=10), Rect(x=5, y=5, width=20, height=20))
(Square(x=0, y=0, size=10), Circle(x=1, y=1, radius=2))
(Rect(x=5, y=5, width=20, height=20), Square(x=15, y=20, size=5))
```

Everything is fine, but let's do a thought experiment. Imagine that our application grew a little bit and was extended with additional elements. If it's a game, someone could include objects representing sprites, actors, or effect particles. Let's say that someone defined the following `Point` class:

```
@dataclass
class Point:
    x: float
    y: float
```

What would happen if the instance of that class was put on the list of possible colliders? You would probably see an exception traceback similar to the following:

```
Traceback (most recent call last):
  File "/.../simple_colliders.py", line 115, in <module>
    for collision in find_collisions([
  File "/.../simple_colliders.py", line 24, in find_collisions
    return [
  File "/.../simple_colliders.py", line 30, in <listcomp>
    item2.bounding_box
AttributeError: 'Point' object has no attribute 'bounding_box'
```

That provides some clue about what the issue is. The question is if we could do better and catch such problems earlier? We could at least verify all input objects' `find_collisions()` functions to check if they are collidable. But how to do that?

Because none of the collidable classes share a common ancestor, we cannot easily use the `isinstance()` function to see if their types match. We can check for the `bounding_box` attribute using the `hasattr()` function, but doing that deeply enough to see whether that attribute has the correct structure would lead us to ugly code.

Here is where `zope.interface` comes in handy. The core class of the `zope.interface` package is the `Interface` class. It allows you to explicitly define a new interface. Let's define an `ICollidable` class that will be our declaration of anything that can be used in our collision system:

```
from zope.interface import Interface, Attribute


class ICollidable(Interface):
    bounding_box = Attribute("Object's bounding box")
```

The common convention for Zope is to prefix interface classes with `I`. The `Attribute` constructor denotes the desired attribute of the objects implementing the interface. Any method defined in the interface class will be used as an interface method declaration. Those methods should be empty. The common convention is to use only the docstring of the method body.

When you have such an interface defined, you must denote which of your concrete classes implement that interface. This style of interface implementation is called explicit interfaces and is similar in nature to traits in Java. In order to denote the implementation of a specific interface, you need to use the `implementer()` class decorator. In our case, this will look as follows:

```python
from zope.interface import implementer


@implementer(ICollidable)
@dataclass
class Square:
    ...


@implementer(ICollidable)
@dataclass
class Rect:
    ...


@implementer(ICollidable)
@dataclass
class Circle:
    ...
```

> The bodies of the dataclasses in the above example have been truncated for the sake of brevity.

It is common to say that the interface defines a contract that a concrete implementation needs to fulfill. The main benefit of this design pattern is being able to verify consistency between contract and implementation before the object is used. With the ordinary duck-typing approach, you only find inconsistencies when there is a missing attribute or method at runtime.

With `zope.interface`, you can introspect the actual implementation using two methods from the `zope.interface.verify` module to find inconsistencies early on:

- `verifyClass(interface, class_object)`: This verifies the class object for the existence of methods and correctness of their signatures without looking for attributes.
- `verifyObject(interface, instance)`: This verifies the methods, their signatures, and also attributes of the actual object instance.

It means that we can extend the `find_collisions()` function to perform initial verification of object interfaces before further processing. We can do that as follows:

```python
from zope.interface.verify import verifyObject

def find_collisions(objects):
    for item in objects:
        verifyObject(ICollidable, item)

    ...
```

Now, if someone passes to the `find_collisions()` function an instance of the class that does not have the `@implementer(ICollidable)` decorator, they will receive an exception traceback similar to this one:

```
Traceback (most recent call last):
  File "/.../colliders_interfaces.py", line 120, in <module>
    for collision in find_collisions([
  File "/.../colliders_interfaces.py", line 26, in find_collisions
    verifyObject(ICollidable, item)
  File "/.../site-packages/zope/interface/verify.py", line 172, in
verifyObject
    return _verify(iface, candidate, tentative, vtype='o')
  File "/.../site-packages/zope/interface/verify.py", line 92, in _
verify
    raise MultipleInvalid(iface, candidate, excs)
zope.interface.exceptions.MultipleInvalid: The object Point(x=100,
y=200) has failed to implement interface <InterfaceClass __main__.
ICollidable>:
    Does not declaratively implement the interface
    The __main__.ICollidable.bounding_box attribute was not provided
```

The last two lines tell us about two errors:

- **Declaration error**: Invalid item isn't explicitly declared to implement the interface and that's an error.
- **Structural error**: Invalid item doesn't have all elements that the interface requires.

The latter error guards us from incomplete interfaces. If the `Point` class had the `@implementer(ICollidable)` decorator but didn't include the `bounding_box()` property, we would still receive the exception.

The `verifyClass()` and `verifyObject()` methods only verify the surface area of the interface and aren't able to traverse into attribute types. You optionally do a more in-depth verification using the `validateInvariants()` method that every interface class of `zope.interface` provides. It allows hook-in functions to validate the values of interfaces. So if we would like to be extra safe, we could use the following pattern of interfaces and their validation:

```python
from zope.interface import Interface, Attribute, invariant
from zope.interface.verify import verifyObject


class IBBox(Interface):
    x1 = Attribute("lower-left x coordinate")
    y1 = Attribute("lower-left y coordinate")
    x2 = Attribute("upper-right x coordinate")
    y2 = Attribute("upper-right y coordinate")


class ICollidable(Interface):
    bounding_box = Attribute("Object's bounding box")
    invariant(lambda self: verifyObject(IBBox, self.bounding_box))

def find_collisions(objects):
    for item in objects:
        verifyObject(ICollidable, item)
        ICollidable.validateInvariants(item)

    ...
```

Thanks to using the `validateInvariants()` method, we are able to check if input items have all attributes necessary to satisfy the `ICollidable` interface, and also verify whether the structure of those attributes (here `bounding_box`) satisfies deeper constraints. In our case, we use `invariant()` to verify the nested interface.

Using `zope.interface` is an interesting way to decouple your application. It allows you to enforce proper object interfaces without the need for the overblown complexity of multiple inheritance, and also allows you to catch inconsistencies early.

The biggest downside of `zope.interface` is the requirement to explicitly declare interface implementors. This is especially troublesome if you need to verify instances coming from the external classes of built-in libraries. The library provides some solutions for that problem, although they make code eventually overly verbose. You can, of course, handle such issues on your own by using the adapter pattern, or even monkey-patching external classes. Anyway, the simplicity of such solutions is at least debatable.

Using function annotations and abstract base classes

Formal interfaces are meant to enable loose coupling in large applications, and not to provide you with more layers of complexity. `zope.interface` is a great concept and may greatly fit some projects, but it is not a silver bullet. By using it, you may shortly find yourself spending more time on fixing issues with incompatible interfaces for third-party classes and providing never-ending layers of adapters instead of writing the actual implementation.

If you feel that way, then this is a sign that something went wrong. Fortunately, Python supports building a lightweight alternative to the explicit interfaces. It's not a full-fledged solution such as `zope.interface` or its alternatives but generally provides more flexible applications. You may need to write a bit more code, but in the end, you will have something that is more extensible, better handles external types, and maybe more *future-proof.*

Note that Python, at its core, does not have an explicit notion of interfaces, and probably never will have, but it has some of the features that allow building something that resembles the functionality of interfaces. The features are as follows:

- **ABCs**
- **Function annotations**
- **Type annotations**

The core of our solution is abstract base classes, so we will feature them first.

As you probably know, direct type comparison is considered harmful and not Pythonic. You should always avoid comparisons as in the following example:

```
assert type(instance) == list
```

Comparing types in functions or methods this way completely breaks the ability to pass a class subtype as an argument to the function. A slightly better approach is to use the `isinstance()` function, which will take the inheritance into account:

```
assert isinstance(instance, list)
```

The additional advantage of `isinstance()` is that you can use a larger range of types to check the type compatibility. For instance, if your function expects to receive some sort of sequence as the argument, you can compare it against the list of basic types:

```
assert isinstance(instance, (list, tuple, range))
```

And such type compatibility checking is OK in some situations but is still not perfect. It will work with any subclass of `list`, `tuple`, or `range`, but will fail if the user passes something that behaves exactly the same as one of these sequence types but does not inherit from any of them. For instance, let's relax our requirements and say that you want to accept any kind of iterable as an argument. What would you do?

The list of basic types that are iterable is actually pretty long. You need to cover `list`, `tuple`, `range`, `str`, `bytes`, `dict`, `set`, `generators`, and a lot more. The list of applicable built-in types is long, and even if you cover all of them, it will still not allow checking against the custom class that defines the `__iter__()` method but inherits directly from `object`.

And this is the kind of situation where ABCs are the proper solution. ABC is a class that does not need to provide a concrete implementation, but instead defines a blueprint of a class that may be used to check against type compatibility. This concept is very similar to the concept of abstract classes and virtual methods known in the C++ language.

Abstract base classes are used for two purposes:

- Checking for implementation completeness
- Checking for implicit interface compatibility

The usage of ABCs is quite simple. You start by defining a new class that either inherits from the `abc.ABC` base class or has `abc.ABCMeta` as its metaclass. We won't be discussing metaclasses until *Chapter 8, Elements of Metaprogramming*, so in this chapter, we'll be using only classic inheritance.

The following is an example of a basic abstract class that defines an interface that doesn't do anything particularly special:

```python
from abc import ABC, abstractmethod


class DummyInterface(ABC):

    @abstractmethod
    def dummy_method(self): ...

    @property
    @abstractmethod
    def dummy_property(self): ...
```

The `@abstractmethod` decorator denotes a part of the interface that must be implemented (by overriding) in classes that will subclass our ABC. If a class will have a nonoverridden method or property, you won't be able to instantiate it. Any attempt to do so will result in a `TypeError` exception.

This approach is a great way to ensure implementation completeness and is as explicit as the `zope.interface` alternative. If we would like to use ABCs instead of `zope.interface` in the example from the previous section, we could do the following modification of class definitions:

```python
from abc import ABC, abstractmethod
from dataclasses import dataclass


class ColliderABC(ABC):
    @property
    @abstractmethod
    def bounding_box(self): ...


@dataclass
class Square(ColliderABC):
    ...


@dataclass
class Rect(ColliderABC):
```

```
    ...

@dataclass
class Circle(ColliderABC):
    ...
```

The bodies and properties of the `Square`, `Rect`, and `Circle` classes don't change as the essence of our interface doesn't change at all. What has changed is the way explicit interface declaration is done. We now use inheritance instead of the `zope.interface.implementer()` class decorator. If we still want to verify if the input of `find_collisions()` conforms to the interface, we need to use the `isinstance()` function. That will be a fairly simple modification:

```
def find_collisions(objects):
    for item in objects:
        if not isinstance(item, ColliderABC):
            raise TypeError(f"{item} is not a collider")
    ...
```

We had to use subclassing so coupling between components is a bit more tight but still comparable to that of `zope.interface`. As far as we rely on interfaces and not on concrete implementations (so, `ColliderABC` instead of `Square`, `Rect`, or `Circle`), coupling is still considered loose.

But things could be more flexible. This is Python and we have full introspection power. Duck typing in Python allows us to use any object that "quacks like a duck" as if it was a duck. Unfortunately, usually it is in the spirit of "try and see." We assume that the object in the given context matches the expected interface. And the whole purpose of formal interfaces was to actually have a contract that we can validate against. Is there a way to check whether an object matches the interface without actually trying to use it first?

Yes. To some extent. Abstract base classes provide the special `__subclasshook__(cls)` method. It allows you to inject your own logic into the procedure that determines whether the object is an instance of a given class. Unfortunately, you need to provide the logic all by yourself, as the `abc` creators did not want to constrain the developers in overriding the whole `isinstance()` mechanism. We have full power over it, but we are forced to write some boilerplate code.

Although you can do whatever you want to, usually the only reasonable thing to do in the `__subclasshook__()` method is to follow the common pattern. In order to verify whether the given class is implicitly compatible with the given abstract base class, we will have to check if it has all the methods of the abstract base class.

The standard procedure is to check whether the set of defined methods are available somewhere in the **Method Resolution Order** (**MRO**) of the given class. If we would like to extend our `ColliderABC` interface with a subclass hook, we could do the following:

```python
class ColliderABC(ABC):
    @property
    @abstractmethod
    def bounding_box(self): ...

    @classmethod
    def __subclasshook__(cls, C):
        if cls is ColliderABC:
            if any("bounding_box" in B.__dict__ for B in C.__mro__):
                return True
        return NotImplemented
```

With the `__subclasshook__()` method defined that way, `ColliderABC` becomes an **implicit interface**. This means that any object will be considered an instance of `ColliderABC` as long as it has the structure that passes the subclass hook check. Thanks to this, we can add new components compatible with the `ColliderABC` interface without explicitly inheriting from it. The following is an example of the `Line` class that would be considered a valid subclass of `ColliderABC`:

```python
@dataclass
class Line:
    p1: Point
    p2: Point

    @property
    def bounding_box(self):
        return Box(
            self.p1.x,
            self.p1.y,
            self.p2.x,
            self.p2.y,
        )
```

As you can see, the `Line` dataclass does not mention `ColliderABC` anywhere in its code. But you can verify the implicit interface compatibility of `Line` instances by comparing them against `ColliderABC` using the `isinstance()` function as in the following example:

```
>>> line = Line(Point(0, 0), Point(100, 100))
>>> line.bounding_box
Box(x1=0, y1=0, x2=100, y2=100)
>>> isinstance(line, ColliderABC)
True
```

We worked with properties, but the same approach may be used for methods as well. Unfortunately, this approach to the verification of type compatibility and implementation completeness does not take into account the signatures of class methods. So, if the number of expected arguments is different in the implementation, it will still be considered compatible. In most cases, this is not an issue, but if you need such fine-grained control over interfaces, the `zope.interface` package allows for that. As already said, the `__subclasshook__()` method does not constrain you in adding much more complexity to the `isinstance()` function's logic to achieve a similar level of control.

Using collections.abc

ABCs are like small building blocks for creating a higher level of abstraction. They allow you to implement really usable interfaces, but are very generic and designed to handle a lot more than this single design pattern. You can unleash your creativity and do magical things, but building something generic and really usable may require a lot of work that may never pay off. Python's Standard Library and Python's built-in types fully embrace the abstract base classes.

The `collections.abc` module provides a lot of predefined ABCs that allow checking for the compatibility of types with common Python interfaces. With the base classes provided in this module, you can check, for example, whether a given object is callable, mapping, or whether it supports iteration. Using them with the `isinstance()` function is way better than comparing against the base Python types. You should definitely know how to use these base classes even if you don't want to define your own custom interfaces with `abc.ABC`.

The most common abstract base classes from `collections.abc` that you will use quite often are:

- `Container`: This interface means that the object supports the `in` operator and implements the `__contains__()` method.
- `Iterable`: This interface means that the object supports iteration and implements the `__iter__()` method.
- `Callable`: This interface means that it can be called like a function and implements the `__call__()` method.

- `Hashable`: This interface means that the object is **hashable** (that is, it can be included in sets and as a key in dictionaries) and implements the __hash__ method.

- `Sized`: This interface means that the object has a size (that is, it can be a subject of the `len()` function) and implements the __len__() method.

> A full list of the available abstract base classes from the `collections.abc` module is available in the official Python documentation under `https://docs.python.org/3/library/collections.abc.html`.

The `collections.abc` module shows pretty well where ABCs work best: creating contracts for small and simple protocols of objects. They won't be good tools to conveniently ensure the fine-grained structure of a large interface. They also don't come with utilities that would allow you to easily verify attributes or perform in-depth validation of function arguments and return types.

Fortunately, there is a completely different solution available for this problem: static type analysis and the `typing.Protocol` type.

Interfaces through type annotations

Type annotations in Python proved to be extremely useful in increasing the quality of software. More and more professional programmers use `mypy` or other static type analysis tools by default, leaving conventional type-less programming for prototypes and quick throwaway scripts.

Support for typing in the standard library and community projects grew greatly in recent years. Thanks to this, the flexibility of typing annotations increases with every Python release. It also allows you to use typing annotations in completely new contexts.

One such context is using type annotations to perform structural subtyping (or static duck-typing). That's simply another approach to the concept of implicit interfaces. It also offers minimal simple-minded runtime check possibilities in the spirit of ABC subclass hooks.

The core of structural subtyping is the `typing.Protocol` type. By subclassing this type, you can create a definition of your interface. The following is an example of base `Protocol` interfaces we could use in our previous examples of the collision detection system:

```python
from typing import Protocol, runtime_checkable


@runtime_checkable
class IBox(Protocol):
    x1: float
    y1: float
    x2: float
    y2: float


@runtime_checkable
class ICollider(Protocol):
    @property
    def bounding_box(self) -> IBox: ...
```

This time we have used two interfaces. Tools like mypy will be able to perform deep type verification so we can use additional interfaces to increase the type safety. The @runtime_checkable decorator extends the protocol class with isinstance() checks. It is something we had to perform manually for ABCs using subclass hooks in the previous section. Here it comes almost for free.

> We will learn more about the usage of static type analysis tools in
> *Chapter 10, Testing and Quality Automation.*

To take full advantage of static type analysis, we also must annotate the rest of the code with proper annotations. The following is the full collision checking code with runtime interface validation based on protocol classes:

```python
import itertools
from dataclasses import dataclass
from typing import Iterable, Protocol, runtime_checkable


@runtime_checkable
class IBox(Protocol):
    x1: float
    y1: float
    x2: float
    y2: float
```

```python
@runtime_checkable
class ICollider(Protocol):
    @property
    def bounding_box(self) -> IBox: ...


def rects_collide(rect1: IBox, rect2: IBox):
    """Check collision between rectangles

    Rectangle coordinates:
        ┌───(x2, y2)
        │      │
     (x1, y1)─┘
    """
    return (
        rect1.x1 < rect2.x2 and
        rect1.x2 > rect2.x1 and
        rect1.y1 < rect2.y2 and
        rect1.y2 > rect2.y1
    )


def find_collisions(objects: Iterable[ICollider]):
    for item in objects:
        if not isinstance(item, ICollider):
            raise TypeError(f"{item} is not a collider")

    return [
        (item1, item2)
        for item1, item2
        in itertools.combinations(objects, 2)
        if rects_collide(
            item1.bounding_box,
            item2.bounding_box
        )
    ]
```

We haven't included the code of the `Rect`, `Square`, and `Circle` classes, because their implementation doesn't have to change. And that's the real beauty of implicit interfaces: there is no explicit interface declaration in a concrete class beyond the inherent interface that comes from the actual implementation.

In the end, we could use any of the previous `Rect`, `Square`, and `Circle` class iterations (plain dataclasses, zope-declared classes, or ABC-descendants). They all would work with structural subtyping through the `typing.Protocol` class.

As you can see, despite the fact that Python lacks native support for interfaces (in the same way as, for instance, Java or the Go language do), we have plenty of ways to standardize contracts of classes, methods, and functions. This ability becomes really useful when implementing various design patterns to solve commonly occurring programming problems. Design patterns are all about reusability and the use of interfaces can help in structuring them into design templates that can be reused over and over again.

But the use of interfaces (and analogous solutions) doesn't end with design patterns. The ability to create a well-defined and verifiable contract for a single unit of code (function, class, or method) is also a crucial element of specific programming paradigms and techniques. Notable examples are **inversion of control** and **dependency injection**. These two concepts are tightly coupled so we will discuss them in the next section together.

Inversion of control and dependency injection

Inversion of Control (**IoC**) is a simple property of some software designs. According to Wiktionary, if a design exhibits IoC, it means that:

> (…) *the flow of control in a system is inverted in comparison to the traditional architecture.*

But what is the traditional architecture? IoC isn't a new idea, and we can trace it back to at least David D. Clark's paper from 1985 titled *The structuring of systems using of upcalls*. It means that traditional design probably refers to the design of software that was common or thought to be traditional in the 1980s.

> You can access Clark's full paper in a digitalized form at `https://groups.csail.mit.edu/ana/Publications/PubPDFs/The%20Structuring%20of%20Systems%20Using%20Upcalls.pdf`.

Clark describes the traditional architecture of a program as a layered structure of procedures where control always goes from top to bottom. Higher-level layers invoke procedures from lower layers.

Those invoked procedures gain control and can invoke even deeper-layered procedures before returning control upward. In practice, control is traditionally passed from application to library functions. Library functions may pass it deeper to even lower-level libraries but, eventually, return it back to the application.

IoC happens when a library passes control up to the application so that the application can take part in the library behavior. To better understand this concept, consider the following trivial example of sorting a list of integer numbers:

```
sorted([1,2,3,4,5,6])
```

The built-in `sorted()` function takes an iterable of items and returns a list of sorted items. Control goes from the caller (your application) directly to the `sorted()` function. When the `sorted()` function is done with sorting, it simply returns the sorted result and gives control back to the caller. Nothing special.

Now let's say we want to sort our numbers in a quite unusual way. That could be, for instance, sorting them by the absolute distance from number 3. Integers closest to 3 should be at the beginning of the result list and the farthest should be at the end. We can do that by defining a simple key function that will specify the order key of our elements:

```
def distance_from_3(item):
    return abs(item - 3)
```

Now we can pass that function as the callback `key` argument to the `sorted()` function:

```
sorted([1,2,3,4,5,6], key=distance_from_3)
```

What will happen now is the `sorted()` function will invoke the `key` function on every element of the iterable argument. Instead of comparing item values, it will now compare the return values of the `key` function. Here is where IoC happens. The `sorted()` function "upcalls" back to the `distance_from_3()` function provided by the application as an argument. Now it is a library that calls the functions from the application, and thus the flow of control is reversed.

> Callback-based IoC is also humorously referred to as the **Hollywood principle** in reference to the "don't call us, we'll call you" phrase.

Note that IoC is just a property of a design and not a design pattern by itself. An example with the `sorted()` function is the simplest example of callback-based IoC but it can take many different forms. For instance:

- **Polymorphism**: When a custom class inherits from a base class and base methods are supposed to call custom methods
- **Argument passing**: When the receiving function is supposed to call methods of the supplied object
- **Decorators**: When a decorator function calls a decorated function
- **Closures**: When a nested function calls a function outside of its scope

As you see, IoC is a rather common aspect of object-oriented or functional programming paradigms. And it also happens quite often without you even realizing it. While it isn't a design pattern by itself, it is a key ingredient of many actual design patterns, paradigms, and methodologies. The most notable one is **dependency injection**, which we will discuss later in this chapter.

Clark's traditional flow of control in procedural programming also happens in object-oriented programming. In object-oriented programs, objects themselves are receivers of control. We can say that control is passed to the object whenever a method of that object is invoked. So the traditional flow of control would require objects to hold full ownership of all dependent objects that are required to fulfill the object's behavior.

Inversion of control in applications

To better illustrate the differences between various flows of control, we will build a small but practical application. It will initially start with a traditional flow of control and later on, we will see if it can benefit from IoC in selected places.

Our use case will be pretty simple and common. We will build a service that can track web page views using so-called **tracking pixels** and serve page view statistics over an HTTP endpoint. This technique is commonly used in tracking advertisement views or email openings. It can also be useful in situations when you make extensive use of HTTP caching and want to make sure that caching does not affect page view statistics.

Our application will have to track counts of page views in some persistent storage. That will also give us the opportunity to explore application modularity—a characteristic that cannot be implemented without IoC.

What we need to build is a small web backend application that will have two endpoints:

- `/track`: This endpoint will return an HTTP response with a 1x1 pixel GIF image. Upon request, it will store the `Referer` header and increase the number of requests associated with that value.

- `/stats`: This endpoint will read the top 10 most common `Referer` values received on the `track/` endpoint and return an HTTP response containing a summary of the results in JSON format.

> The `Referer` header is an optional HTTP header that web browsers will use to tell the web server what is the URL of the origin web page from which the resource is being requested. Take note of the misspelling of the word *referrer*. The header was first standardized in *RFC 1945, Hypertext Transfer Protocol – HTTP/1.0* (see `https://tools.ietf.org/html/rfc1945`). When the misspelling was discovered, it was already too late to fix it.

We've already introduced Flask as a simple web microframework in *Chapter 2, Modern Python Development Environments*, so we will use it here as well. Let's start by importing some modules and setting up module variables that we will use on the way:

```python
from collections import Counter
from http import HTTPStatus

from flask import Flask, request, Response

app = Flask(__name__)
storage = Counter()

PIXEL = (
    b'GIF89a\x01\x00\x01\x00\x80\x00\x00\x00'
    b'\x00\x00\xff\xff\xff!\xf9\x04\x01\x00'
    b'\x00\x00\x00,\x00\x00\x00\x00\x01\x00'
    b'\x01\x00\x00\x02\x01D\x00;'
)
```

The `app` variable is the core object of the Flask framework. It represents a Flask web application. We will use it later to register endpoint routes and also run the application development server.

The storage variable holds a Counter instance. It is a convenient data structure from the Standard Library that allows you to track counters of any immutable values. Our ultimate goal is to store page view statistics in a persistent way, but it will be a lot easier to start off with something simpler. That's why we will initially use this variable as our in-memory storage of page view statistics.

Last but not least, is the PIXEL variable. It holds a byte representation of a 1x1 transparent GIF image. The actual visual appearance of the tracking pixel does not matter and probably will never change. It is also so small that there's no need to bother with loading it from the filesystem. That's why we are **inlining** it in our module to fit the whole application in a single Python module.

Once we're set, we can write code for the /track endpoint handler:

```python
@app.route('/track')
def track():
    try:
        referer = request.headers["Referer"]
    except KeyError:
        return Response(status=HTTPStatus.BAD_REQUEST)

    storage[referer] += 1

    return Response(
        PIXEL, headers={
            "Content-Type": "image/gif",
            "Expires": "Mon, 01 Jan 1990 00:00:00 GMT",
            "Cache-Control": "no-cache, no-store, must-revalidate",
            "Pragma": "no-cache",
        }
    )
```

> We use extra Expires, Cache-Control, and Pragma headers to control the HTTP caching mechanism. We set them so that they would disable any form of caching on most web browser implementations. We also do it in a way that should disable caching by potential proxies. Take careful note of the Expires header value that is way in the past. This is the lowest possible **epoch time** and in practice means that resource is always considered expired.

Flask request handlers typically start with the `@app.route(route)` decorator that registers the following handler function for the given HTTP route. Request handlers are also known as views. Here we have registered the `track()` view as a handler of the `/track` route endpoint. This is the first occurrence of IoC in our application: we register our own handler implementation within Flask frameworks. It is a framework that will call back our handlers on incoming requests that match associated routes.

After the signature, we have simple code for handling the request. We check if the incoming request has the expected `Referer` header. That's the value which the browser uses to tell what URI the requested resource was included on (for instance, the HTML page we want to track). If there's no such header, we will return an error response with a `400 Bad Request` HTTP status code.

If the incoming request has the `Referer` header, we will increase the counter value in the `storage` variable. The `Counter` structure has a `dict`-like interface and allows you to easily modify counter values for keys that haven't been registered yet. In such a case, it will assume that the initial value for the given key was 0. That way we don't need to check whether a specific `Referer` value was already seen and that greatly simplifies the code. After increasing the counter value, we return a pixel response that can be finally displayed by the browser.

Note that although the `storage` variable is defined outside the `track()` function, it is not yet an example of IoC. That's because whoever calls the `stats()` function can't replace the implementation of the storage. We will try to change that in the next iterations of our application.

The code for the `/stats` endpoint is even simpler:

```python
@app.route('/stats')
def stats():
    return dict(storage.most_common(10))
```

In the `stats()` view, we again take advantage of the convenient interface of the `Counter` object. It provides the `most_common(n)` method, which returns up to n most common key-value pairs stored in the structure. We immediately convert that to a dictionary. We don't use the `Response` class, as Flask by default serializes the non-`Response` class return values to JSON and assumes a `200 OK` status for the HTTP response.

In order to test our application easily, we finish our script with the simple invocation of the built-in development server:

```python
if __name__ == '__main__':
    app.run(host="0.0.0.0", port=8000)
```

If you store the application in the `tracking.py` file, you will be able to start the server using the `python tracking.py` command. It will start listening on port `8000`. If you would like to test the application in your own browser, you can extend it with the following endpoint handler:

```python
@app.route('/test')
def test():
    return """
    <html>
    <head></head>
    <body><img src="/track"></body>
    </html>
    """
```

If you open the address `http://localhost:8000/test` several times in your web browser and then go to `http://localhost:8000/stats`, you will see output similar to the following:

```
{"http://localhost:8000/test":6}
```

The problem with the current implementation is that it stores request counters in memory. Whenever the application is restarted, the existing counters will be reset and we'll lose important data. In order to keep the data between restarts, we will have to replace our storage implementation.

The options to provide data persistency are many. We could, for instance, use:

- A simple text file
- The built-in `shelve` module
- A **relational database management system** (**RDBMS**) like MySQL, MariaDB, or PostgreSQL
- An in-memory key-value or data struct storage service like Memcached or Redis

Depending on the context and scale of the workload our application needs to handle, the best solution will be different. If we don't know yet what is the best solution, we can also make the storage pluggable so we can switch storage backends depending on the actual user needs. To do so, we will have to invert the flow of control in our `track()` and `stats()` functions.

Good design dictates the preparation of some sort of definition of the interface of the object that is responsible for the IoC. The interface of the `Counter` class seems like a good starting point. It is convenient to use. The only problem is that the `+=` operation can be implemented through either the `__add__()` or `__iadd__()` special method. We definitely want to avoid such ambiguity. Also, the `Counter` class has way too many extra methods and we need only two:

- A method that allows you to increase the counter value by one
- A method that allows you to retrieve the 10 most often requested keys

To keep things simple, and readable, we will define our views storage interface as an abstract base class of the following form:

```python
from abc import ABC, abstractmethod
from typing import Dict


class ViewsStorageBackend(ABC):
    @abstractmethod
    def increment(self, key: str): ...

    @abstractmethod
    def most_common(self, n: int): Dict[str, int] ...
```

From now on, we can provide various implementations of the views storage backend. The following will be the implementation that adapts the previously used `Counter` class into the `ViewsStorageBackend` interface:

```python
from collections import Counter
from typing import Dict

from .tracking_abc import ViewsStorageBackend


class CounterBackend(ViewsStorageBackend):
    def __init__(self):
        self._counter = Counter()

    def increment(self, key: str):
        self._counter[key] += 1

    def most_common(self, n: int) -> Dict[str, int]:
        return dict(self._counter.most_common(n))
```

If we would like to provide persistency through the Redis in-memory storage service, we could do so by implementing a new storage backend as follows:

```python
from typing import Dict
from redis import Redis


class RedisBackend(ViewsStorageBackend):
    def __init__(
        self,
        redis_client: Redis,
        set_name: str
    ):
        self._client = redis_client
        self._set_name = set_name

    def increment(self, key: str):
        self._client.zincrby(self._set_name, 1, key)

    def most_common(self, n: int) -> Dict[str, int]:
        return {
            key.decode(): int(value)
            for key, value in
            self._client.zrange(
                self._set_name, 0, n-1,
                desc=True,
                withscores=True,
            )
        }
```

> Redis is an in-memory data store. This means that by default, data is stored only in memory. Redis will persist data on disk during restart but may lose data in an unexpected crash (for instance, due to a power outage). Still, this is only a default behavior. Redis offers various modes for data persistence, some of which are comparable to other databases. This means Redis is a completely viable storage solution for our simple use case. You can read more about Redis persistence at `https://redis.io/topics/persistence`.

Both backends have the same interface loosely enforced with an abstract base class. It means instances of both classes can be used interchangeably. The question is, how will we invert control of our `track()` and `stats()` functions in a way that will allow us to plug in a different views storage implementation?

Let's recall the signatures of our functions:

```python
@app.route('/stats')
def stats():
    ...


@app.route('/track')
def track():
    ...
```

In the Flask framework, the `app.route()` decorator registers a function as a specific route handler. You can think of it as a callback for HTTP request paths. You don't call that function manually anymore and Flask is in full control of the arguments passed to it. But we want to be able to easily replace the storage implementation. One way to do that would be through postponing the handler registration and letting our functions receive an extra `storage` argument. Consider the following example:

```python
def track(storage: ViewsStorageBackend):
    try:
        referer = request.headers["Referer"]
    except KeyError:
        return Response(status=HTTPStatus.BAD_REQUEST)

    storage.increment(referer)

    return Response(
        PIXEL, headers={
            "Content-Type": "image/gif",
            "Expires": "Mon, 01 Jan 1990 00:00:00 GMT",
            "Cache-Control": "no-cache, no-store, must-revalidate",
            "Pragma": "no-cache",
        }
    )


def stats(storage: ViewsStorageBackend):
    return storage.most_common(10)
```

Our extra argument is annotated with the `ViewsStorageBackend` type so the type can be easily verified with an IDE or additional tools. Thanks to this we have inverted control of those functions and also achieved better modularity. Now you can easily switch the implementation of storage for different classes with a compatible interface. The extra benefit of IoC is that we can easily unit-test `stats()` and `track()` methods in isolation from storage implementations.

> We will discuss the topic of unit-tests together with detailed examples of tests that leverage IoC in *Chapter 10*, *Testing and Quality Automation.*

The only part that is missing is actual route registration. We can no longer use the `app.route()` decorator directly on our functions. That's because Flask won't be able to resolve the `storage` argument on its own. We can overcome that problem by "pre-injecting" desired storage implementations into handler functions and create new functions that can be easily registered with the `app.route()` call.

The simple way to do that would be using the `partial()` function from the `functools` module. It takes a single function together with a set of arguments and keyword arguments and returns a new function that has selected arguments preconfigured. We can use that approach to prepare various configurations of our service. Here, for instance, is an application configuration that uses Redis as a storage backend:

```python
from functools import partial

if __name__ == '__main__':
    views_storage = RedisBackend(Redis(host="redis"), "my-stats")

    app.route("/track", endpoint="track")(
        partial(track, storage=views_storage))
    app.route("/stats", endpoint="stats")(
        partial(stats, storage=views_storage))

    app.run(host="0.0.0.0", port=8000)
```

The presented approach can be applied to many other web frameworks as the majority of them have the same route-to-handler structure. It will work especially well for small services with only a handful of endpoints. Unfortunately, it may not scale well in large applications. It is simple to write but definitely not the easiest to read. Seasoned Flask programmers will for sure feel this approach is unnatural and needlessly repetitive. Here, it simply breaks the common convention of writing Flask handler functions.

The ultimate solution would be one that allows you to write and register view functions without the need to manually inject dependent objects. So, for instance:

```python
@app.route('/track')
def track(storage: ViewsStorageBackend):
    ...
```

In order to do that, from the Flask framework we would need to:

- Recognize extra arguments as dependencies of views.
- Allow the definition of a default implementation for said dependencies.
- Automatically resolve dependencies and inject them into views at runtime.

Such a mechanism is referred to as **dependency injection**, which we mentioned previously. Some web frameworks offer a built-in dependency injection mechanism, but in the Python ecosystem, it is a rather rare occurrence. Fortunately, there are plenty of lightweight dependency injection libraries that can be added on top of any Python framework. We will explore such a possibility in the next section.

Using dependency injection frameworks

When IoC is used at a great scale, it can easily become overwhelming. The example from the previous section was quite simple so it didn't require a lot of setup. Unfortunately, we have sacrificed a bit of readability and expressiveness for better modularity and responsibility isolation. For larger applications, this can be a serious problem.

Dedicated dependency injection libraries come to the rescue by combining a simple way to mark function or object dependencies with a runtime dependency resolution. All of that usually can be achieved with minimal impact on the overall code structure.

There are plenty of dependency injection libraries for Python, so definitely there is no need to build your own from scratch. They are often similar in implementation and functionality, so we will simply pick one and see how it could be applied in our view tracking application.

Our library of choice will be the injector library, which is freely available on PyPI. We will pick it up for several reasons:

- **Reasonably active and mature**: Developed over more than 10 years with releases every few months.

- **Framework support**: It has community support for various frameworks including Flask through the `flask-injector` package.

- **Typing annotation support**: It allows writing unobtrusive dependency annotations and leveraging static typing analysis.

- **Simple**: `injector` has a Pythonic API. It makes code easy to read and to reason about.

> You can install `injector` in your environment using `pip` as follows:
>
> ```
> $ pip install injector
> ```
>
> You can find more information about `injector` at `https://github.com/alecthomas/injector`.

In our example, we will use the `flask-injector` package as it provides some initial boilerplate to integrate `injector` with Flask seamlessly. But before we do that, we will first separate our application into several modules that would better simulate a larger application. After all, dependency injection really shines in applications that have multiple components.

We will create the following Python modules:

- `interfaces`: This will be the module holding our interfaces. It will contain `ViewsStorageBackend` from the previous section without any changes.

- `backends`: This will be the module holding specific implementations of storage backends. It will contain `CounterBackend` and `RedisBackend` from the previous section without any changes.

- `tracking`: This will be the module holding the application setup together with view functions.

- `di`: This will be the module holding definitions for the `injector` library, which will allow it to automatically resolve dependencies.

The core of the `injector` library is a `Module` class. It defines a so-called **dependency injection container**—an atomic block of mapping between dependency interfaces and their actual implementation instances. The minimal `Module` subclass may look as follows:

```python
from injector import Module, provider

def MyModule(Module):
```

```
    @provider
    def provide_dependency(self, *args) -> Type:
        return ...
```

The `@provider` decorator marks a `Module` method as a method providing the implementation for a particular `Type` interface. The creation of some objects may be complex, so `injector` allows modules to have additional nondecorated helper methods.

The method that provides dependency may also have its own dependencies. They are defined as method arguments with type annotations. This allows for cascading dependency resolution. `injector` supports composing dependency injection context from multiple modules so there's no need to define all dependencies in a single module.

Using the above template, we can create our first injector module in the `di.py` file. It will be `CounterModule`, which provides a `CounterBackend` implementation for the `ViewsStorageBackend` interface. The definition will be as follows:

```
from injector import Module, provider, singleton

from interfaces import ViewsStorageBackend
from backends import CounterBackend


class CounterModule(Module):
    @provider
    @singleton
    def provide_storage(self) -> ViewsStorageBackend:
        return CounterBackend()
```

`CounterStorage` doesn't take any arguments, so we don't have to define extra dependencies. The only difference from the general module template is the `@singleton` decorator. It is an explicit implementation of the **singleton design pattern**. A singleton is simply a class that can have only a single instance. In this context, it means that every time this dependency is resolved, `injector` will always return the same object. We need that because `CounterStorage` stores view counters under the internal `_counter` attribute. Without the `@singleton` decorator, every request for the `ViewsStorageBackend` implementation would return a completely new object and thus we would constantly lose track of view numbers.

The implementation of `RedisModule` will be only slightly more complex:

```python
from injector import Module, provider, singleton
from redis import Redis

from interfaces import ViewsStorageBackend
from backends import RedisBackend

class RedisModule(Module):
    @provider
    def provide_storage(self, client: Redis) -> ViewsStorageBackend:
        return RedisBackend(client, "my-set")

    @provider
    @singleton
    def provide_redis_client(self) -> Redis:
        return Redis(host="redis")
```

> The code files for this chapter provide a complete `docker-compose` environment with a preconfigured Redis Docker image so you don't have to install Redis on your own host.

In the `RedisStorage` module, we take advantage of the `injector` library's ability to resolve cascading dependencies. The `RedisBackend` constructor requires a Redis client instance so we can treat it as another `provide_storage()` method argument. `injector` will recognize typing annotation and automatically match the method that provides the `Redis` class instance. We could go even further and extract a host argument to separate configuration dependency. We won't do that for the sake of simplicity.

Now we have to tie everything up in the `tracking` module. We will be relying on `injector` to resolve dependencies on views. This means that we can finally define `track()` and `stats()` handlers with extra `storage` arguments and register them with the `@app.route()` decorator as if they were normal Flask views. Updated signatures will be the following:

```python
@app.route('/stats')
def stats(storage: ViewsStorageBackend):
    ...


@app.route('/track')
def track(storage: ViewsStorageBackend):
    ...
```

What is left is the final configuration of the app that designates which modules should be used to provide interface implementations. If we would like to use `RedisBackend`, we would finish our `tracking` module with the following code:

```python
import di


if __name__ == '__main__':
    FlaskInjector(app=app, modules=[di.RedisModule()])
    app.run(host="0.0.0.0", port=8000)
```

The following is the complete code of the `tracking` module:

```python
from http import HTTPStatus

from flask import Flask, request, Response
from flask_injector import FlaskInjector

from interfaces import ViewsStorageBackend
import di


app = Flask(__name__)

PIXEL = (
    b'GIF89a\x01\x00\x01\x00\x80\x00\x00\x00'
    b'\x00\x00\xff\xff\xff!\xf9\x04\x01\x00'
    b'\x00\x00\x00,\x00\x00\x00\x00\x01\x00'
    b'\x01\x00\x00\x02\x01D\x00;'
)


@app.route('/track')
def track(storage: ViewsStorageBackend):
    try:
        referer = request.headers["Referer"]
    except KeyError:
        return Response(status=HTTPStatus.BAD_REQUEST)

    storage.increment(referer)

    return Response(
        PIXEL, headers={
            "Content-Type": "image/gif",
            "Expires": "Mon, 01 Jan 1990 00:00:00 GMT",
```

```python
            "Cache-Control": "no-cache, no-store, must-revalidate",
            "Pragma": "no-cache",
        }
    )


@app.route('/stats')
def stats(storage: ViewsStorageBackend):
    return storage.most_common(10)


@app.route("/test")
def test():
    return """
    <html>
    <head></head>
    <body><img src="/track"></body>
    </html>
    """


if __name__ == '__main__':
    FlaskInjector(app=app, modules=[di.RedisModule()])
    app.run(host="0.0.0.0", port=8000)
```

As you can see, the introduction of the dependency injection mechanism didn't change the core of our application a lot. The preceding code closely resembles the first and simplest iteration, which didn't have the IoC mechanism. At the cost of a few interface and injector module definitions, we've got scaffolding for a modular application that could easily grow into something much bigger. We could, for instance, extend it with additional storage that would serve more analytical purposes or provide a dashboard that allows you to view the data at different angles.

Another advantage of dependency injection is loose coupling. In our example, views never create instances of storage backends nor their underlying service clients (in the case of `RedisBackend`). They depend on shared interfaces but are independent of implementations. Loose coupling is usually a good foundation for a well-architected application.

It is of course hard to show the utility of IoC and dependency injection in a really concise example like the one we've just seen. That's because these techniques really shine in big applications. Anyway, we will revisit the use case of the pixel tracking application in *Chapter 10*, *Testing and Quality Automation*, where we will show that IoC greatly improves the testability of your code.

Summary

This chapter was a journey through time. Python is considered a modern language but in order to better understand its patterns, we had to make some historical trips.

We started with interfaces—a concept almost as old as object-oriented programming (the first OOP language—Simula—dates to 1967!). We took a look at `zope.interface`, something that is probably one of the oldest actively maintained interface libraries in the Python ecosystem. We learned some of its advantages and disadvantages. That allowed us to really embrace two mainstream Python alternatives: abstract base classes and structural subtyping through advanced type annotations.

After familiarizing ourselves with interfaces, we looked into inversion of control. Internet sources about this topic can be really confusing and this concept is often confused with dependency injection. To settle any disputes, we traced the origin of the term to the 80s, when no one had yet ever dreamed about dependency injection containers. We learned how to recognize inversion of control in various forms and saw how it can improve the modularity of applications. We tried to invert control in a simple application manually. We saw that sometimes it can cost us readability and expressiveness. Thanks to this, we are now able to fully recognize the value that comes from the simplicity of ready-made dependency injection libraries.

The next chapter should be refreshing. We will completely move away from the topics of object-oriented programming, language features, design patterns, and paradigms. It will be all about concurrency. We will learn how to write code that does a lot, in parallel, and—hopefully—does it fast.

6
Concurrency

Concurrency and one of its manifestations, parallel processing, are among the broadest topics in the area of software engineering. Concurrency is such a huge topic that dozens of books could be written and we would still not be able to discuss all of its important aspects and models. The purpose of this chapter is to show you why concurrency may be required in your application, when to use it, and what Python's most important concurrency models are.

We will discuss some of the language features, built-in modules, and third-party packages that allow you to implement these models in your code. But we won't cover them in much detail. Treat the content of this chapter as an entry point for your own research and reading. We will try to guide you through the basic ideas and help in deciding if you really need concurrency. Hopefully, after reading this chapter you will be able to tell which approach suits your needs best.

In this chapter, we will cover the following topics:

- What is concurrency?
- Multithreading
- Multiprocessing
- Asynchronous programming

Before we get into the basic concepts of concurrency, let's begin by considering the technical requirements.

Technical requirements

The following are the Python packages that are used in this chapter, which you can download from PyPI:

- requests
- aiohttp

Information on how to install packages is included in *Chapter 2*, *Modern Python Development Environments*.

The code files for this chapter can be found at `https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition/tree/main/Chapter%206`.

Before we delve into various implementations of concurrency available to Python programmers, let's discuss what concurrency actually is.

What is concurrency?

Concurrency is often confused with actual methods of implementing it. Some programmers also think that it is a synonym for parallel processing. This is the reason why we need to start by properly defining concurrency. Only then will we be able to properly understand various concurrency models and their key differences.

First and foremost, concurrency is not the same as parallelism. Concurrency is also not a matter of application implementation. Concurrency is a property of a program, algorithm, or problem, whereas parallelism is just one of the possible approaches to problems that are concurrent.

In Leslie Lamport's 1976 paper *Time, Clocks, and the Ordering of Events in Distributed Systems*, he defines the concept of concurrency as follows:

> *"Two events are concurrent if neither can causally affect the other."*

By extrapolating events to programs, algorithms, or problems, we can say that something is concurrent if it can be fully or partially decomposed into components (units) that are order-independent. Such units may be processed independently from each other, and the order of processing does not affect the final result. This means that they can also be processed simultaneously or in parallel. If we process information this way (that is, in parallel), then we are indeed dealing with parallel processing. But this is still not obligatory.

Doing work in a distributed manner, preferably using the capabilities of multicore processors or computing clusters, is a natural consequence of concurrent problems. Anyway, it does not mean that this is the only way of efficiently dealing with concurrency. There are a lot of use cases where concurrent problems can be approached in ways other than synchronous ways, but without the need for parallel execution. In other words, when a problem is concurrent, it gives you the opportunity to deal with it in a special, preferably more efficient, way.

We often get used to solving problems in a classical way: by performing a sequence of steps. This is how most of us think and process information—using synchronous algorithms that do one thing at a time, step by step. But this way of processing information is not well suited to solving large-scale problems or when you need to satisfy the demands of multiple users or software agents simultaneously:

- When the time to process the job is limited by the performance of the single processing unit (a single machine, CPU core, and so on)
- When you are not able to accept and process new inputs until your program has finished processing the previous one

These problems create three common application scenarios where concurrent processing is a viable approach to satisfy user needs:

- **Processing distribution**: The scale of the problem is so big that the only way to process it in an acceptable time frame (with constrained resources) is to distribute execution on multiple processing units that can handle the work in parallel.
- **Application responsiveness**: Your application needs to maintain responsiveness (accept new inputs), even if it did not finish processing previous inputs.
- **Background processing**: Not every task needs to be performed in a synchronous way. If there is no need to immediately access the results of a specific action, it may be reasonable to defer execution in time.

The processing distribution scenario directly maps to parallel processing. That's why it is usually solved with multithreading and multiprocessing models. The application responsiveness scenario often doesn't require parallel processing, so the actual solution really depends on the problem details. The problem of application responsiveness also covers the case when the application needs to serve multiple clients (users or software agents) independently, without the need to wait for others to be successfully served.

It is an interesting observation that these groups of problems are not exclusive. Often, you will have to maintain application responsiveness and at the same time won't be able to handle all the inputs on a single processing unit. This is the reason why different and seemingly alternative or conflicting approaches to concurrency may often be used at the same time. This is especially common in the development of web servers, where it may be necessary to use asynchronous event loops, or threads in conjunction with multiple processes, in order to utilize all the available resources and still maintain low latencies under the high load.

Python provides several ways to deal with concurrency. These are mainly:

- **Multithreading**: This is characterized by running multiple threads of execution that share the memory context of the parent process. It is one of the most popular (and oldest) concurrency models and works best in applications that do a lot of **I/O** (**Input/Output**) operations or need to maintain user interface responsiveness. It is fairly lightweight but comes with a lot of caveats and memory safety risks.

- **Multiprocessing**: This is characterized by running multiple independent processes to perform work in a distributed manner. It is similar to threads in operation, although it does not rely on a shared memory context. Due to the nature of Python, it is better suited for CPU-intensive applications. It is more heavyweight than multithreading and requires implementing inter-process communication patterns to orchestrate work between processes.

- **Asynchronous programming**: This is characterized by running multiple cooperative tasks within a single application process. Cooperative tasks work like threads, although switching between them is facilitated by the application itself instead of the operating system kernel. It is well suited to I/O-bound applications, especially for programs that need to handle multiple simultaneous network connections. The downside of asynchronous programming is the need to use dedicated asynchronous libraries.

The first model we will discuss in detail is multithreading.

Multithreading

Developers often consider multithreading to be a very complex topic. While this statement is totally true, Python provides high-level classes and functions that greatly help in using threads. CPython has some inconvenient implementation details that make threads less effective than in other programming languages like C or Java. But that doesn't mean that they are completely useless in Python.

There is still quite a large range of problems that can be solved effectively and conveniently with Python threads.

In this section, we will discuss those limitations of multithreading in CPython, as well as the common concurrent problems for which Python threads are still a viable solution.

What is multithreading?

Thread is short for a thread of execution. A programmer can split their work into threads that run simultaneously. Threads are still bound to the parent process and can easily communicate because they share the same memory context. The execution of threads is coordinated by the OS kernel.

Multithreading will benefit from a multiprocessor or multicore machines, where each thread can be executed on a separate CPU core, thus making the program run faster. This is a general rule that should hold true for most programming languages. In Python, the performance benefit from multithreading on multicore CPUs has some limits, which we will discuss later. For the sake of simplicity, let's assume for now that this statement is also true for Python.

The simplest way to start a new thread of execution using Python is to use the `threading.Thread()` class as in the following example:

```python
def my_function():
    print("printing from thread")


if __name__ == "__main__":
    thread = Thread(target=my_function)
    thread.start()
    thread.join()
```

The `my_function()` function is the function we want to execute in the new thread. We pass it to the `Thread` class constructor as the `target` keyword argument. Instances of this class are used to encapsulate and control application threads.

Creating a new `Thread` class instance is not enough to start a new thread. In order to do this, you need to call the `start()` method. Once the new thread is started, it will be running next to the main thread until the `target` function finishes. In the above example, we explicitly wait for the extra thread to finish using the `join()` method.

> We say that the `join()` method is a **blocking operation**. This means that the thread isn't doing anything in particular (it doesn't consume CPU time) and simply waits for a specific event to happen.

The `start()` and `join()` methods allow you to create and start multiple threads at once. The following is a simple modification of the previous example that starts and joins multiple threads in bulk:

```python
from threading import Thread


def my_function():
    print("printing from thread")


if __name__ == "__main__":
    threads = [Thread(target=my_function) for _ in range(10)]
    for thread in threads:
        thread.start()

    for thread in threads:
        thread.join()
```

All threads share the same memory context. This means that you must be extremely wary about how your threads access the same data structures. If two parallel threads update the same variable without any protection, there might be a situation where a subtle timing variation in thread execution can alter the final result in an unexpected way. To better understand this problem, let's consider a small program that runs multiple threads reading and updating the same value:

```python
from threading import Thread

thread_visits = 0


def visit_counter():
    global thread_visits
    for i in range(100_000):
        value = thread_visits
        thread_visits = value + 1
```

```python
if __name__ == "__main__":
    thread_count = 100
    threads = [
        Thread(target=visit_counter)
        for _ in range(thread_count)
    ]
    for thread in threads:
        thread.start()

    for thread in threads:
        thread.join()

    print(f"{thread_count=}, {thread_visits=}")
```

The above program starts 100 threads and each one tries to read and increment the `thread_visits` variable 100,000 times. If we were to run the tasks sequentially, the final value of the `thread_visits` variable should be 10,000,000. But threads can interweave and lead to unexpected results. Let's save the above code example in the `threaded_visits.py` file and run it a few times to see the actual results:

```
$ python3 thread_visits.py
thread_count=100, thread_visits=6859624
$ python3 thread_visits.py
thread_count=100, thread_visits=7234223
$ python3 thread_visits.py
thread_count=100, thread_visits=7194665
```

On each run, we got a completely different number, and it was always very far from the expected 10,000,000 thread visits. But that doesn't mean that the actual number of thread visits was that small. With such a large number of threads, they started interweaving and affecting our results.

Such a situation is called a **race hazard** or **race condition**. It is one of the most hated culprits of software bugs for multithreaded applications. Obviously, there is a slice of time between the read and write operations on the `thread_visits` variable where another thread can step in and manipulate the result.

One might think that the problem could be fixed using the += operator, which looks like a single atomic operation:

```python
def visit_counter():
    global thread_visits
    for i in range(100_000):
        thread_visits += 1
```

But that won't help us either! The += operator is just a shorthand for incrementing a variable, but it will actually take a few operations in the Python interpreter. Between those operations, there's still time for threads to interweave.

The proper way around race conditions is to use thread locking primitives. Python has a few lock classes in the threading module. Here we can use the simplest one—threading.Lock. The following is an example of a thread-safe visit_counter() function:

```python
from threading import Lock

thread_visits = 0
thread_visits_lock = Lock()

def visit_counter():
    global thread_visits
    for i in range(100_000):
        with thread_visits_lock:
            thread_visits += 1
```

If you run the modified version of the code, you will notice that thread visits with locks are counted properly. But that will be at the expense of performance. The threading Lock() will make sure that only one thread at a time can process a single block of code. This means that the protected block cannot run in parallel. Moreover, acquiring and releasing the lock are operations that require some additional effort. With a lot of threads trying to access the lock, a performance drop will be noticeable. We will see other examples of using locks to secure parallel data access later in the chapter.

Multithreading is usually supported at the OS kernel level. When a machine has a single processor with a single core, the system uses a time slicing mechanism to allow threads to run seemingly in parallel. With time slicing, the CPU switches from one thread to another so fast that there is an illusion of threads running simultaneously.

> Single-core CPUs are pretty uncommon these days in desktop computers but can still be a concern in other areas. Small and cheap instances in many cloud compute platforms, as well as low-cost embedded systems, often have only single-core CPUs or virtual CPUs.

Parallelism without multiple processing units is obviously virtual, and the application performance gain on such hardware is harder to evaluate. Anyway, sometimes, it is still useful to implement code with threads, even if it means having to execute on a single core. We will review such use cases later.

Everything changes when your execution environment has multiple processors or multiple processor cores. In such cases, threads can be distributed among CPUs or their cores by the OS kernel. This thus provides the opportunity to run your program substantially faster. This is true for many programming languages but not necessarily for Python. To understand why that is so, let's take a closer look at how Python deals with threads.

How Python deals with threads

Unlike some other languages, Python uses multiple kernel-level threads that can run any of the interpreter-level threads. Kernel-level threads are operated and scheduled by the OS kernel. CPython uses OS-specific system calls to create threads and join threads. It doesn't have full control over when threads run and on which CPU core they will execute. These responsibilities are left to the sole discretion of the system kernel. Moreover, the kernel can preempt a running thread at any time, for instance, to run a thread with a higher priority.

Unfortunately, the standard implementation of the Python (the CPython interpreter) language comes with a major limitation that renders threads less useful in many contexts. All operations accessing Python objects are **serialized** by one global lock. This is done because many of the interpreter's internal structures are not thread-safe and need to be protected. Not every operation requires locking, and there are certain situations when threads release the lock.

> In the context of parallel processing, if we say that something is **serialized**, we mean that actions are taken in a serial fashion (one after another). Unintended serialization in concurrent programs is usually something that we want to avoid.

This mechanism of the CPython interpreter is known as the **Global Interpreter Lock** (**GIL**). The removal of the GIL is a topic that occasionally appears on the Python-dev emailing list and was postulated by Python developers multiple times. Sadly, at the time of writing, no one has ever managed to provide a reasonable and simple solution that would allow you to get rid of this limitation. It is highly improbable that we will see any progress in this area anytime soon. It is safer to assume that the GIL will stay in CPython, and so we need to learn how to live with it.

So, what is the point of multithreading in Python? When threads contain only pure Python code and don't do any I/O operations (like communicating through sockets), there is little point in using threads to speed up the program. That's because the GIL will most likely globally serialize the execution of all threads. But remember that the GIL cares only about protecting Python objects. In practice, the GIL is released on a number of blocking system calls like socket calls. It can be also released in sections of C extensions that do not use any Python/C API functions. This means that multiple threads can do I/O operations or execute specifically crafted C extension code completely in parallel.

> We will discuss the details of interacting with the GIL in Python C extensions in *Chapter 9*, *Bridging Python with C and C++*.

Multithreading allows you to efficiently utilize time when your program is waiting for an external resource. This is because a sleeping thread that has released the GIL (this happens internally in CPython) can wait on "standby" and "wake up" when the results are back. Last, whenever a program needs to provide a responsive interface, multithreading can be an answer, even in single-core environments where the OS needs to use time slicing. With multithreading, the program can easily interact with the user while doing some heavy computing in the so-called background.

> Note that the GIL does not exist in every implementation of the Python language. It is a limitation of CPython, Stackless Python, and PyPy, but does not exist in Jython (Python for JVM) and IronPython (Python for .NET). There has also been some development of a GIL-free version of PyPy. It is based on software transactional memory and is called **PyPy-STM**.

In the next section, we will discuss more specific examples of situations where threading can be useful.

When should we use multithreading?

Despite the GIL limitation, threads can be really useful in some of the following cases:

- **Application responsiveness**: Applications that can accept new input and respond within a given time frame (be responsive) even if they did not finish processing previous inputs.
- **Multiuser applications and network communication**: Applications that are supposed to accept inputs of multiple users simultaneously often communicate with users over the network. This means that they can heavily reduce the impact of locking by leveraging those parts of CPython where the GIL is released.
- **Work delegation and background processing**: Applications where much of the heavy lifting is done by external applications or services and your code acts as a gateway to those resources.

Let's start with responsive applications, as those are the ones that tend to prefer multithreading over other concurrency models.

Application responsiveness

Let's say you ask your OS to copy a large file from one folder to another through its graphical user interface. The task will possibly be pushed into the background and the interface window will display a constantly refreshed progress status. This way, you get live feedback on the progress of the whole process. You will also be able to cancel the operation. You can also carry out other work like browsing the web or editing your documents while your OS is still copying the file. The graphical user interface of your system will stay responsive to your actions. This is less irritating than a raw `cp` or `copy` shell command that does not provide any feedback until the entirety of the work is finished.

A responsive interface also allows a user to work on several tasks at the same time. For instance, **Gimp** (a popular open-source image editing application) will let you play around with a picture while another one is being filtered, since the two tasks are independent.

When trying to achieve such responsive interfaces, a good approach is to try to push long-running tasks into the background, or at least try to provide constant feedback to the user. The easiest way to achieve that is to use threads. In such a scenario, threads are used to make sure that the user can still operate the interface, even if the application needs to process its tasks for a longer period of time.

This approach is often used together with event-driven programming where the main application thread pushes events to be processed by background worker threads (see *Chapter 7*, *Event-Driven Programming*). Web browsers are good examples of applications that often use this architectural pattern.

> Do not confuse **application responsiveness** with **Responsive Web Design** (**RDW**). The latter is a popular design approach of web applications that allows you to display the same web application well on a variety of mediums (such as desktop browsers, mobiles, or tablets).

Multiuser applications

Serving multiple users simultaneously may be understood as a special case of application responsiveness. The key difference is that here the application has to satisfy the parallel inputs of many users and each one of them may have some expectations about how quickly the application should respond. Simply put, one user should not have to wait for other user inputs to be processed in order to be served.

Threading is a popular concurrency model for multiuser applications and is extremely common in web applications. For instance, the main thread of a web server may accept all incoming connections but dispatch the processing of every single request to a separate dedicated thread. This usually allows us to handle multiple connections and requests at the same time. The number of connections and requests the application will be able to handle at the same time is only constrained by the ability of the main thread to quickly accept connections and dispatch requests to new threads. A limitation of this approach is that applications using it can quickly consume many resources. Threads are not free: memory is shared but each thread will have at least its own stack allocated. If the number of threads is too large, the total memory consumption can quickly get out of hand.

Another model of threaded multiuser applications assumes that there is always a limited pool of threads acting as workers that are able to process incoming user inputs. The main thread is then only responsible for allocating and managing the pool of workers. Web applications often utilize this pattern too. A web server, for instance, can create a limited number of threads and each of those threads will be able to accept connections on its own and handle all requests incoming on that connection. This approach usually allows you to serve fewer users at the same time (compared to one thread per request) but gives more control over resource usage. Two very popular Python WSGI-compliant web servers—Gunicorn and uWSGI— allow serving HTTP requests with threaded workers in a way that generally follows this principle.

> **WSGI** stands for **Web Server Gateway Interface**. It is a common Python standard (defined in PEP 3333, accessible at `https://www.python.org/dev/peps/pep-3333/`) for communication between web servers and applications that promotes the portability of web applications between web servers. Most modern Python web frameworks and web servers are based on the WSGI.

Using multithreading to enable concurrency in multiuser applications is generally less expensive in terms of resources than using multiprocessing. Separate Python processes will use more memory than threads since a new interpreter needs to be loaded for each one of them. On the other hand, having too many threads is expensive too. We know that the GIL isn't such a problem for I/O-intensive applications, but there will always be a time when you will need to execute Python code. Since you cannot parallelize all of the application parts with bare threads, you will never be able to utilize all of the resources on machines with multicore CPUs and a single Python process. This is why the optimal solution is sometimes a hybrid of multiprocessing and multithreading—multiple workers (processes) running with multiple threads. Fortunately, some WSGI-compliant web servers allow for such setup (for instance, Gunicorn with the `gthread` worker type).

Multiuser applications often utilize the delegation of work to threads as a means of ensuring proper responsiveness for multiple users. But work delegation alone can also be understood as a standalone use case for multithreading too.

Work delegation and background processing

If your application depends on many external resources, threads may really help in speeding it up. Let's consider the case of a function that indexes files in a folder and pushes the built indexes into a database. Depending on the type of file, the function executes a different processing routine. For example, one is specialized in PDFs and another one in OpenOffice files.

Instead of processing all files in a sequence, your function can set up a single thread for each converter and push jobs to be done to each one of them through a queue. The overall time taken by the function will be closer to the processing time of the slowest converter than to the total sum of the work.

The other common use case for threads is performing multiple network requests to an external service. For instance, if you want to fetch multiple results from a remote web API, it could take a lot of time to do that synchronously, especially if the remote server is located in a distant location.

If you wait for every previous response before making new requests, you will spend a lot of time just waiting for the external service to respond. Additional round-trip time delays will be added to every such request.

If you are communicating with some efficient service (the Google Maps API, for instance), it is highly probable that it can serve most of your requests concurrently without affecting the response times of individual requests. It is then reasonable to perform multiple queries in separate threads. Here, when doing an HTTP request, the application will most likely spend most of its time reading from the TCP socket. Delegating such work to threads allows for a great improvement of your application's performance.

An example of a multithreaded application

To see how Python threading works in practice, let's construct an example application that could benefit from the usage of threads. We will consider a simple problem that was already highlighted in the previous section as a common use case for multithreading: making parallel HTTP requests to some remote service.

Let's say we need to fetch information from some web service using multiple queries that cannot be batched into a single bulk HTTP request. As a realistic example, we will use the foreign exchange reference rates endpoint from a free API, available at `https://www.vatcomply.com`. The reasons for this choice are as follows:

- This service is open and does not require any authentication keys.

- The interface of the service is very simple and can be easily queried using the popular `requests` library.

- This API uses a currency data format that is common across many similar APIs. If this service goes down (or stops being free), you will be able to easily switch the base URL of the API to the URL of a different service.

> Free API services come and go. It is possible that after some time the URLs in this book won't work or the API will require a paid subscription. In such cases, running your own service may be a good option.
>
> At `https://github.com/exchangeratesapi/ exchangeratesapi`, you can find code for a currency exchange API service that uses the same data format as the API used in this chapter.

In our examples, we will try to obtain exchange rates for selected currencies using multiple currencies as reference points. We will then present the results as an exchange rate currency matrix, similar to the following:

```
1 USD =    1.0 USD, 0.887 EUR,   3.8 PLN,  8.53 NOK,  22.7 CZK
1 EUR =   1.13 USD,   1.0 EUR,  4.29 PLN,  9.62 NOK,  25.6 CZK
1 PLN =  0.263 USD, 0.233 EUR,   1.0 PLN,  2.24 NOK,  5.98 CZK
1 NOK =  0.117 USD, 0.104 EUR, 0.446 PLN,   1.0 NOK,  2.66 CZK
1 CZK =  0.044 USD, 0.039 EUR, 0.167 PLN, 0.375 NOK,   1.0 CZK
```

The API we've chosen offers several ways to query for multiple data points within single requests, but unfortunately it does not allow you to query for data using multiple base currencies at once. Obtaining the rate for a single base is as simple as doing the following:

```
>>> import requests
>>> response = requests.get("https://api.vatcomply.com/rates?base=USD")
>>> response.json()
{'base': 'USD', 'rates': {'BGN': 1.7343265053, 'NZD': 1.4824864769,
'ILS': 3.5777245721, 'RUB': 64.7361000266, 'CAD': 1.3287221779, 'USD':
1.0, 'PHP': 52.0368892436, 'CHF': 0.9993792675, 'AUD': 1.3993970027,
'JPY': 111.2973308504, 'TRY': 5.6802341048, 'HKD': 7.8425113062,
'MYR': 4.0986077858, 'HRK': 6.5923561231, 'CZK': 22.7170346723,
'IDR': 14132.9963642813, 'DKK': 6.6196683515, 'NOK': 8.5297508203,
'HUF': 285.09355325, 'GBP': 0.7655848187, 'MXN': 18.930477964, 'THB':
31.7495787887, 'ISK': 118.6485767491, 'ZAR': 14.0298838344, 'BRL':
3.8548372794, 'SGD': 1.3527533919, 'PLN': 3.8015429636, 'INR':
69.3340427419, 'KRW': 1139.4519819101, 'RON': 4.221867518, 'CNY':
6.7117141084, 'SEK': 9.2444799149, 'EUR': 0.8867606633}, 'date': '2019-
04-09'}
```

> In order to keep our examples concise, we will use the `requests` package to perform HTTP requests. It is not a part of the standard library but can be easily obtained from PyPI using `pip`.
>
> You can read more about `requests` at `https://requests.readthedocs.io/`.

Since our goal is to show how a multithreaded solution of concurrent problems compares to a classical synchronous solution, we will start with an implementation that doesn't use threads at all. Here is the code of a program that loops over the list of base currencies, queries the foreign exchange rates API, and displays the results on standard output as a text-formatted table:

```python
import time

import requests

SYMBOLS = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')
BASES = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')

def fetch_rates(base):
    response = requests.get(
        f"https://api.vatcomply.com/rates?base={base}"
    )
    response.raise_for_status()
    rates = response.json()["rates"]

    # note: same currency exchanges to itself 1:1
    rates[base] = 1.

    rates_line = ", ".join(
        [f"{rates[symbol]:7.03} {symbol}" for symbol in SYMBOLS]
    )
    print(f"1 {base} = {rates_line}")

def main():
    for base in BASES:
        fetch_rates(base)

if __name__ == "__main__":
    started = time.time()
    main()
    elapsed = time.time() - started

    print()
    print("time elapsed: {:.2f}s".format(elapsed))
```

The `main()` function iterates over a list of base currencies and calls the `fetch_rates()` function to obtain exchange rates for the base currencies. Inside `fetch_rates()`, we make a single HTTP request using the `requests.get()` function. The `response.raise_for_status()` method will raise an exception if the server returns a response with a status code denoting a server or client error. For now, we don't expect any exceptions and simply assume that after receiving the request, we can successfully read the response payload using the `response.json()` method. We will discuss how to properly handle exceptions raised within threads in the *Dealing with errors in threads* section.

We've added a few statements around the execution of the `main()` function that are intended to measure how much time it took to finish the job. Let's save that code in a file named `synchronous.py` and execute it to see how it works:

```
$ python3 synchronous.py
```

On my computer, it can take a couple of seconds to complete that task:

```
1 USD =     1.0 USD,   0.823 EUR,    3.73 PLN,     8.5 NOK,    21.5 CZK
1 EUR =    1.22 USD,     1.0 EUR,    4.54 PLN,    10.3 NOK,    26.2 CZK
1 PLN =   0.268 USD,    0.22 EUR,     1.0 PLN,    2.28 NOK,    5.76 CZK
1 NOK =   0.118 USD,  0.0968 EUR,   0.439 PLN,     1.0 NOK,    2.53 CZK
1 CZK =  0.0465 USD,  0.0382 EUR,   0.174 PLN,   0.395 NOK,     1.0 CZK

time elapsed: 4.08s
```

Every run of our script will always take a different amount of time. This is because the processing time mostly depends on a remote service that's accessible through a network connection. There are many non-deterministic factors affecting the final result. If we wanted to be really methodical, we would make longer tests, repeat them multiple times, and calculate an average from the measurements. But for the sake of simplicity, we won't do that. You will see later that this simplified approach is just enough for illustration purposes.

We have some baseline implementation. Now it is time to introduce threads. In the next section, we will try to introduce one thread per call of the `fetch_rates()` function.

Using one thread per item

Now, it is time for improvement. We don't do a lot of processing in Python, and long execution times are caused by communication with the external service. We send an HTTP request to the remote server, it calculates the answer, and then we wait until the response is transferred back.

There is a lot of I/O involved, so multithreading seems like a viable option. We can start all the requests at once in separate threads and then just wait until we receive data from all of them. If the service that we are communicating with is able to process our requests concurrently, we should definitely see a performance improvement.

So, let's start with the easiest approach. Python provides clean and easy-to-use abstraction over system threads with the `threading` module. The core of this standard library is the `Thread` class, which represents a single thread instance. Here is a modified version of the `main()` function that creates and starts a new thread for every base currency to process and then waits until all the threads finish:

```python
from threading import Thread

def main():
    threads = []
    for base in BASES:
        thread = Thread(target=fetch_rates, args=[base])
        thread.start()
        threads.append(thread)

    while threads:
        threads.pop().join()
```

It is a quick and dirty solution that approaches the problem in a bit of a frivolous way. It has some serious issues that we will have to address later. But hey, it works. Let's save the modified script in the `threads_one_per_item.py` file and run it to see if there is some performance improvement:

```
$ python3 one_thread_per_item.py
```

On my computer, I see substantial improvement in total processing time:

```
1 EUR =    1.22 USD,    1.0 EUR,    4.54 PLN,    10.3 NOK,    26.2 CZK
1 NOK =   0.118 USD, 0.0968 EUR,   0.439 PLN,     1.0 NOK,    2.53 CZK
1 CZK =  0.0465 USD, 0.0382 EUR,   0.174 PLN,   0.395 NOK,     1.0 CZK
1 USD =     1.0 USD,  0.823 EUR,    3.73 PLN,     8.5 NOK,    21.5 CZK
1 PLN =   0.268 USD,   0.22 EUR,     1.0 PLN,    2.28 NOK,    5.76 CZK

time elapsed: 1.34s
```

> Due to using `print()` inside of a thread, the output you will see may be slightly malformed. This is one of the multithreading problems that we will take care of later in this section.

Once we know that threads have a beneficial effect on our application, it is time to use them in a more logical way. First, we need to identify the following issues in the preceding code:

- We start a new thread for every parameter. Thread initialization also takes some time, but this minor overhead is not the only problem. Threads also consume other resources, like memory or file descriptors. Our example input has a strictly defined number of items, but what if it did not have a limit? You definitely don't want to run an unbound number of threads that depend on the arbitrary size of data input.

- The `fetch_rates()` function that's executed in threads calls the built-in `print()` function, and in practice it is very unlikely that you would want to do that outside of the main application thread. This is mainly due to the way the standard output is buffered in Python. You can experience malformed output when multiple calls to this function interweave between threads. Also, the `print()` function is considered slow. If used recklessly in multiple threads, it can lead to serialization that will waste all your benefits of multithreading.

- Last but not least, by delegating every function call to a separate thread, we make it extremely hard to control the rate at which our input is processed. Yes, we want to do the job as fast as possible, but very often, external services enforce hard limits on the rate of requests from a single client that they can process. Sometimes, it is reasonable to design a program in a way that enables you to throttle the rate of processing, so your application won't be blacklisted by external APIs for abusing their usage limits.

In the next section, we will see how to use a thread pool to solve the problem of an unbounded number of threads.

Using a thread pool

The first issue we will try to solve is the unbounded number of threads that are run by our program. A good solution would be to build a pool of threaded workers with a strictly defined size that will handle all the parallel work and communicate with main thread through some thread-safe data structure. By using this thread pool approach, we will also make it easier to solve two other problems that we've mentioned in the previous section.

The general idea is to start a predefined number of threads that will consume the work items from a queue until it becomes empty. When there is no other work to do, the threads will quit, and we will be able to exit from the program. A good candidate for our communication data structure is the `Queue` class from the built-in `queue` module. It is a **First-In First-Out** (**FIFO**) queue implementation that is very similar to the `deque` collection from the `collections` module and was specifically designed to handle inter-thread communication. Here is a modified version of the `main()` function that starts only a limited number of worker threads with a new `worker()` function as a target and communicates with them using a thread-safe queue:

```python
from queue import Queue
from threading import Thread

THREAD_POOL_SIZE = 4

def main():
    work_queue = Queue()

    for base in BASES:
        work_queue.put(base)

    threads = [
        Thread(target=worker, args=(work_queue,))
        for _ in range(THREAD_POOL_SIZE)
    ]

    for thread in threads:
        thread.start()

    work_queue.join()

    while threads:
        threads.pop().join()
```

> Python has some built-in thread pooling utilities. We will cover them later in the *Using multiprocessing.dummy as the multithreading interface* section.

The `main` function initializes the `Queue` instance as the `worker_queue` variable and puts all the base currencies in the queue as items of work to be processed by worker threads. It then initializes the `THREAD_POOL_SIZE` number of threads with the `worker()` function as a thread target and `work_queue` as their input argument. It then waits until all items have been processed using `work_queue.join()` and then waits for all threads to finish by calling the `join` method of every `Thread` instance.

The processing of work items from the queue happens in the `worker` function. Here is its code:

```python
from queue import Empty


def worker(work_queue):
    while not work_queue.empty():
        try:
            item = work_queue.get_nowait()
        except Empty:
            break
        else:
            fetch_rates(item)
            work_queue.task_done()
```

The `worker()` function runs in a `while` loop until `work_queue.empty()` returns `True`. In every iteration, it tries to obtain a new item in a non-blocking fashion using the `work_queue.get_nowait()` method. If the queue is already empty, it will raise an `Empty` exception, and our function will break the loop and finish. If there is an item to pick from the queue, the `worker()` function will pass it to `fetch_rates(item)` and mark the item as processed using `work_queue.task_done()`. When all items from the queue have been marked as done, the `work_queue.join()` function from the main thread will return.

The rest of the script, namely the `fetch_rates()` function and the code under the `if __name__ == "__main__"` clause, stays the same. The following is the full script that we can save in the `thread_pool.py` file:

```python
import time
from queue import Queue, Empty
from threading import Thread

import requests

THREAD_POOL_SIZE = 4
SYMBOLS = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')
```

```python
BASES = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')


def fetch_rates(base):
    response = requests.get(
        f"https://api.vatcomply.com/rates?base={base}"
    )

    response.raise_for_status()
    rates = response.json()["rates"]
    # note: same currency exchanges to itself 1:1
    rates[base] = 1.

    rates_line = ", ".join(
        [f"{rates[symbol]:7.03} {symbol}" for symbol in SYMBOLS]
    )
    print(f"1 {base} = {rates_line}")


def worker(work_queue):
    while not work_queue.empty():
        try:
            item = work_queue.get_nowait()
        except Empty:
            break
        else:
            fetch_rates(item)
            work_queue.task_done()


def main():
    work_queue = Queue()

    for base in BASES:
        work_queue.put(base)

    threads = [
        Thread(target=worker, args=(work_queue,))
        for _ in range(THREAD_POOL_SIZE)
    ]

    for thread in threads:
        thread.start()
```

```
    work_queue.join()

    while threads:
        threads.pop().join()

if __name__ == "__main__":
    started = time.time()
    main()
    elapsed = time.time() - started

    print()
    print("time elapsed: {:.2f}s".format(elapsed))
```

We can now execute the script to see if there is any performance difference between the previous attempt:

```
$ python thread_pool.py
```

On my computer, I can see the following output:

```
1 NOK =   0.118 USD,  0.0968 EUR,   0.439 PLN,      1.0 NOK,    2.53 CZK
1 PLN =   0.268 USD,    0.22 EUR,    1.0 PLN,     2.28 NOK,    5.76 CZK
1 USD =     1.0 USD,   0.823 EUR,   3.73 PLN,      8.5 NOK,    21.5 CZK
1 EUR =    1.22 USD,     1.0 EUR,   4.54 PLN,     10.3 NOK,    26.2 CZK
1 CZK =  0.0465 USD,  0.0382 EUR,   0.174 PLN,   0.395 NOK,     1.0 CZK

time elapsed: 1.90s
```

The overall runtime may be slower than when using one thread per argument, but at least now it is not possible to exhaust all the computing resources with an arbitrarily long input. Also, we can tweak the THREAD_POOL_SIZE parameter for a better resource/time balance.

In this attempt, we used an unmodified version of the fetch_rates() function that outputs the API result on the standard output directly from within the thread. In some cases, this may lead to malformed output when two threads attempt to print results at the same time. In the next section, we will try to improve it by introducing two-way queues.

Using two-way queues

The issue that we are now able to solve is the potentially problematic printing of the output in threads. It would be much better to leave such a responsibility to the main thread that started the worker threads. We can handle that by providing another queue that will be responsible for collecting results from our workers. Here is the complete code that puts everything together, with the main changes highlighted:

```python
import time
from queue import Queue, Empty
from threading import Thread

import requests

SYMBOLS = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')
BASES = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')

THREAD_POOL_SIZE = 4

def fetch_rates(base):
    response = requests.get(
        f"https://api.vatcomply.com/rates?base={base}"
    )
    response.raise_for_status()
    rates = response.json()["rates"]

    # note: same currency exchanges to itself 1:1
    rates[base] = 1.
    return base, rates

def present_result(base, rates):
    rates_line = ", ".join([
        f"{rates[symbol]:7.03} {symbol}"
        for symbol in SYMBOLS
    ])
    print(f"1 {base} = {rates_line}")

def worker(work_queue, results_queue):
    while not work_queue.empty():
        try:
            item = work_queue.get_nowait()
        except Empty:
            break
```

```python
        else:
            results_queue.put(fetch_rates(item))
            work_queue.task_done()

def main():
    work_queue = Queue()
    results_queue = Queue()

    for base in BASES:
        work_queue.put(base)

    threads = [
        Thread(
            target=worker,
            args=(work_queue, results_queue)
        ) for _ in range(THREAD_POOL_SIZE)
    ]

    for thread in threads:
        thread.start()

    work_queue.join()

    while threads:
        threads.pop().join()

    while not results_queue.empty():
        present_result(*results_queue.get())

if __name__ == "__main__":
    started = time.time()
    main()
    elapsed = time.time() - started

    print()
    print("time elapsed: {:.2f}s".format(elapsed))
```

The main difference is the introduction of the `results_queue` instance of the `Queue` class and the `presents_results()` function. The `fetch_rates()` function no longer prints its results to standard output. It instead returns processed API results straight to the `worker()` function. Worker threads pass those results unmodified through a new `results_queue` output queue.

Now only the main thread is responsible for printing the results on standard output. After all the work has been marked as done, the `main()` function consumes results from `results_queue` and passes them to the `present_results()` function.

This eliminates the risk of malformed inputs that we could encounter if the `present_result()` function would do more `print()`. We don't expect any performance improvement from this approach with small inputs, but in fact we also reduced the risk of thread serialization due to slow `print()` execution.

In all the previous examples, we've assumed that the API we use will always respond with a meaningful and valid answer. We've didn't cover any failure scenarios to keep things simple, but in real applications, it could be a problem. In the next section, we will see what happens when an exception is raised within a thread and how it affects communication over queues.

Dealing with errors in threads

The `raise_for_status()` method of the `requests.Response` object will raise an exception if the HTTP response has a status code indicating the error condition. We have used that method in all the previous iterations of the `fetch_rates()` function but we haven't handled any potential exceptions yet.

If the service we are calling with the `requests.get()` method responds with a status code indicating an error, the exception will be raised in a separate thread and will not crash the entire program. The worker thread will, of course, exit immediately. But the main thread will wait for all tasks stored on `work_queue` to finish (with the `work_queue.join()` call). Without further improvement, we may end up in a situation where some of the worker threads crashed and the program will never exit. To avoid this we should ensure that our worker threads gracefully handle possible exceptions and make sure that all items from the queue are processed.

Let's make some minor changes to our code in order to be prepared for any issues that may occur. In case there are exceptions in the worker thread, we may put an error instance on the `results_queue` queue so the main thread will be able to tell which of the tasks have failed to process. We can also mark the current task as done, the same as we would do if there was no error. That way, we make sure that the main thread won't lock indefinitely while waiting on the `work_queue.join()` method call.

The main thread might then inspect the results and re-raise any of the exceptions found on the results queue. Here are the improved versions of the `worker()` and `main()` functions that can deal with exceptions in a safer way (the changes are highlighted):

```python
def worker(work_queue, results_queue):
    while not work_queue.empty():
        try:
            item = work_queue.get_nowait()
        except Empty:
            break

        try:
            result = fetch_rates(item)
        except Exception as err:
            results_queue.put(err)
        else:
            results_queue.put(result)
        finally:
            work_queue.task_done()


def main():
    work_queue = Queue()
    results_queue = Queue()

    for base in BASES:
        work_queue.put(base)

    threads = [
        Thread(target=worker, args=(work_queue, results_queue))
        for _ in range(THREAD_POOL_SIZE)
    ]

    for thread in threads:
        thread.start()

    work_queue.join()

    while threads:
        threads.pop().join()

    while not results_queue.empty():
        result = results_queue.get()
        if isinstance(result, Exception):
            raise result

        present_result(*result)
```

To see how error handling works in action we will try to simulate a convincing error scenario. Since we don't have full control over the API we use, we will randomly inject error responses to the `fetch_rates()` function. The following is the modified version of that function:

```python
import random

def fetch_rates(base):
    response = requests.get(
        f"https://api.vatcomply.com/rates?base={base}"
    )

    if random.randint(0, 5) < 1:
        # simulate error by overriding status code
        response.status_code = 500

    response.raise_for_status()
    rates = response.json()["rates"]
    # note: same currency exchanges to itself 1:1
    rates[base] = 1.
    return base, rates
```

By modifying `response.status_code` to `500`, we will simulate the situation of our API returning a response indicating a server error. This is a common status code for issues occurring on the server side. In such situations, details of the error are not always disclosed. This status code is just enough for the `response.raise_for_status()` method to raise an exception.

Let's save a modified version of the code in the `error_handling.py` file and run it to see how it handles exceptions:

```
$ python3 error_handling.py
```

Errors are injected randomly, so this may need to be executed a few times. After a couple of tries, you should see an output similar to the following:

```
1 PLN =   0.268 USD,   0.22 EUR,   1.0 PLN,   2.28 NOK,   5.76 CZK
Traceback (most recent call last):
  File ".../error_handling.py", line 92, in <module>
    main()
  File ".../error_handling.py", line 85, in main
    raise result
  File ".../error_handling.py", line 53, in worker
```

```
    result = fetch_rates(item)
  File ".../error_handling.py", line 30, in fetch_rates
    response.raise_for_status()
  File ".../.venv/lib/python3.9/site-packages/requests/models.py", line
943, in raise_for_status
    raise HTTPError(http_error_msg, response=self)
requests.exceptions.HTTPError: 500 Server Error: OK for url: https://
api.vatcomply.com/rates?base=NOK
```

Our code did not succeed in obtaining all the items, but we at least got clear information about the error cause, which was in this case a `500 Server Error` response status.

In the next section, we will make the last improvement to our multithreaded program. We will introduce a throttling mechanism to protect our program from rate limiting and avoid the accidental abuse of the free service we use.

Throttling

The last of the issues mentioned in the *Using one thread per item* section that we haven't tackled yet is potential rate limits that may be imposed by external service providers. In the case of the foreign exchange rates API, the service maintainer did not inform us about any rate limits or throttling mechanisms. But many services (even paid ones) often do impose rate limits.

Usually, when a service has rate limits implemented, it will start returning responses indicating errors after a certain number of requests are made, surpassing the allocated quota. We've already prepared for error responses in the previous section, but that is often not enough to properly handle rate limits. That's because many services often count requests made beyond the limit, and if you go beyond the limit consistently, you may never get back to the allocated quota.

When using multiple threads, it is very easy to exhaust any rate limit or simply—if the service does not throttle incoming requests—saturate the service to the level that it will not be able to respond to anyone. If done on purpose, this is known as a **Denial-of-Service** (**DoS**) attack.

In order to not go over the rate limits or cause accidental DoS, we need to limit the pace at which we make requests to the remote service. Limiting the pace of work is often called throttling. There are a few packages in PyPI that allow you to limit the rate of any kind of work that are really easy to use. But we won't use any external code here. Throttling is a good opportunity to introduce some locking primitives for threading, so we will try to build a throttling solution from scratch.

The algorithm we will use is sometimes called a **token bucket** and is very simple. It includes the following functionality:

- There is a bucket with a predefined number of tokens
- Each token corresponds to a single permission to process one item of work
- Each time the worker asks for one or more tokens (permissions), we do the following:
    1. We check how much time has passed since the last time we refilled the bucket
    2. If the time difference allows for it, we refill the bucket with the number of tokens that correspond to the time difference
    3. If the number of stored tokens is bigger than or equal to the amount requested, we decrease the number of stored tokens and return that value
    4. If the number of stored tokens is less than requested, we return zero

The two important things are to always initialize the token bucket with zero tokens and to never allow it to overfill. This may be counter-intuitive, but if we don't follow these precautions, we can release the tokens in bursts that exceed the rate limit. Because, in our situation, the rate limit is expressed in requests per second, we don't need to deal with arbitrary amount of time. We assume that the base for our measurement is one second, so we will never store more tokens than the number of requests allowed for that amount of time. Here is an example implementation of the class that allows for throttling with the token bucket algorithm:

```python
from threading import Lock

class Throttle:
    def __init__(self, rate):
        self._consume_lock = Lock()
        self.rate = rate
        self.tokens = 0
        self.last = None

    def consume(self, amount=1):
        with self._consume_lock:
            now = time.time()

            # time measurement is initialized on first
            # token request to avoid initial bursts
            if self.last is None:
```

```
        self.last = now

    elapsed = now - self.last

    # make sure that quant of passed time is big
    # enough to add new tokens
    if elapsed * self.rate > 1:
        self.tokens += elapsed * self.rate
        self.last = now

    # never over-fill the bucket
    self.tokens = min(self.rate, self.tokens)

    # finally dispatch tokens if available
    if self.tokens >= amount:
        self.tokens -= amount
        return amount

    return 0
```

The usage of this class is very simple. We have to create only one instance of `Throttle` (for example, `Throttle(10)`) in the main thread and pass it to every worker thread as a positional argument:

```
def main():
    work_queue = Queue()
    results_queue = Queue()
    throttle = Throttle(10)

    for base in BASES:
        work_queue.put(base)

    threads = [
        Thread(
            target=worker,
            args=(work_queue, results_queue, throttle)
        ) for _ in range(THREAD_POOL_SIZE)
    ]
    ...
```

This throttle instance will be shared across threads, but it is safe to use because we guarded the manipulation of its internal state with the instance of the `Lock` class from the `threading` module. We can now update the `worker()` function implementation to wait with every item until the `throttle` object releases a new token, as follows:

```python
import time

def worker(work_queue, results_queue, throttle):
    while True:
        try:
            item = work_queue.get_nowait()
        except Empty:
            break

        while not throttle.consume():
            time.sleep(0.1)

        try:
            result = fetch_rates(item)
        except Exception as err:
            results_queue.put(err)
        else:
            results_queue.put(result)
            finally:
                work_queue.task_done()
```

The `while not throttle.consume()` block prevents us from processing work queue items if a `throttle` object does not release any tokens (zero evaluates to `False`). We've put a short sleep to add some pacing for the threads in the event of an empty bucket. There's probably a more elegant way to do that, but this simple technique does the job fairly well.

When `throttle.consume()` returns a non-zero value, we consider the token consumed. The thread can exit the `while` loop and proceed with processing the work queue item. When the processing is done, it will read another item from the work queue and again try to consume the token. This whole process will continue until the work queue is empty.

This was a very brief introduction to threads. We haven't covered every possible aspect of multithreaded applications, but we already know enough to take a look at other concurrency models and see how they compare to threads. The next concurrency model will be multiprocessing.

Multiprocessing

Let's be honest, multithreading is challenging. Dealing with threads in a sane and safe manner required a tremendous amount of code when compared to the synchronous approach. We had to set up a thread pool and communication queues, gracefully handle exceptions from threads, and also worry about thread safety when trying to provide a rate limiting capability. Dozens of lines of code are needed just to execute one function from some external library in parallel! And we rely on the promise from the external package creator that their library is thread-safe. Sounds like a high price for a solution that is practically applicable only for doing I/O-bound tasks.

An alternative approach that allows you to achieve parallelism is multiprocessing. Separate Python processes that do not constrain each other with the GIL allow for better resource utilization. This is especially important for applications running on multicore processors that are performing really CPU-intensive tasks. Right now, this is the only built-in concurrent solution available for Python developers (using CPython interpreter) that allows you to take benefit from multiple processor cores in every situation.

The other advantage of using multiple processes over threads is the fact that they do not share a memory context. Thus, it is harder to corrupt data and introduce deadlocks or race conditions in your application. Not sharing the memory context means that you need some additional effort to pass the data between separate processes, but fortunately there are many good ways to implement reliable inter-process communication. In fact, Python provides some primitives that make communication between processes almost as easy as it is between threads.

The most basic way to start new processes in any programming language is usually by forking the program at some point. On POSIX and POSIX-like systems (like UNIX, macOS, and Linux), a fork is a system call that will create a new child process. In Python it is exposed through the `os.fork()` function. The two processes continue the program in their own right after the forking. Here is an example script that forks itself exactly once:

```python
import os

pid_list = []

def main():
    pid_list.append(os.getpid())
    child_pid = os.fork()

    if child_pid == 0:
```

```
            pid_list.append(os.getpid())
            print()
            print("CHLD: hey, I am the child process")
            print("CHLD: all the pids I know %s" % pid_list)

        else:
            pid_list.append(os.getpid())
            print()
            print("PRNT: hey, I am the parent process")
            print("PRNT: the child pid is %d" % child_pid)
            print("PRNT: all the pids I know %s" % pid_list)


if __name__ == "__main__":
    main()
```

The os.fork() spawns a new process. Both processes will have the same memory state till the moment of the fork() call, but after that call their memories diverge, hence the fork name. os.fork() returns an integer value. If it is 0, we know that the current process is a child process. The parent process will receive the **Process ID (PID)** number of its child process.

Let's save the script in the forks.py file and run it in a shell session:

```
$ python3 forks.py
```

On my computer, I've got the following output:

```
PRNT: hey, I am the parent process
PRNT: the child pid is 9304
PRNT: all the pids I know [9303, 9303]

CHLD: hey, I am the child process
CHLD: all the pids I know [9303, 9304]
```

Notice how both processes have exactly the same initial state of their data before the os.fork() call. They both have the same PID number (process identifier) as a first value of the pid_list collection.

Later, both states diverge. We can see that the child process added the 9304 value while the parent duplicated its 9303 PID. This is because the memory contexts of these two processes are not shared. They have the same initial conditions but cannot affect each other after the os.fork() call.

After the fork, each process gets its own address space. To communicate, processes need to work with system-wide resources or use low-level tools like signals.

Unfortunately, `os.fork` is not available under Windows, where a new interpreter needs to be spawned in order to mimic the fork feature. Therefore, the multiprocessing implementation depends on the platform. The `os` module also exposes functions that allow you to spawn new processes under Windows. Python provides the great `multiprocessing` module, which creates a high-level interface for multiprocessing.

The great advantage of the `multiprocessing` module is that it provides some of the abstractions that we had to code from scratch when we discussed multithreading. It allows you to limit the amount of boilerplate code, so it improves application maintainability and reduces complexity. Surprisingly, despite its name, the `multiprocessing` module exposes a similar interface for threads, so you will probably want to use the same interface for both approaches.

Let's take a closer look at the built-in `multiprocessing` module in the next section.

The built-in multiprocessing module

The `multiprocessing` module provides a portable way to work with processes as if they were threads. This module contains a `Process` class that is very similar to the `Thread` class, and can be used on any platform, as follows:

```python
from multiprocessing import Process
import os


def work(identifier):
    print(
        f'Hey, I am the process '
        f'{identifier}, pid: {os.getpid()}'
    )


def main():
    processes = [
        Process(target=work, args=(number,))
        for number in range(5)
    ]
    for process in processes:
        process.start()

    while processes:
        processes.pop().join()
```

```
if __name__ == "__main__":
    main()
```

The `Process` class has `start()` and `join()` methods that are similar to the methods in the `Thread` class. The `start()` method spawns a new process and `join()` waits until the child process exits.

Let's save that script in a file called `basic_multiprocessing.py` and execute it to see how it works in action:

```
$ python3 basic_multiprocessing.py
```

On your own computer, you will be able to see output similar to the following:

```
Hey, I am the process 3, pid: 9632
Hey, I am the process 1, pid: 9630
Hey, I am the process 2, pid: 9631
Hey, I am the process 0, pid: 9629
Hey, I am the process 4, pid: 9633
```

When processes are created, the memory is forked (on POSIX and POSIX-like systems). Besides the memory state that is copied, the `Process` class also provides an extra `args` argument in its constructor so that data can be passed along.

Communication between processes requires some additional work because their local memory is not shared by default. To ease this, the `multiprocessing` module provides the following few ways of communicating between processes:

- Using the `multiprocessing.Queue` class, which is a functional equivalent of `queue.Queue`, which was used earlier for communication between threads.

- Using `multiprocessing.Pipe`, which is a socket-like two-way communication channel.

- Using the `multiprocessing.sharedctypes` module, which allows you to create arbitrary C types (from the `ctypes` module) in a dedicated pool of memory that is shared between processes.

The `multiprocessing.Queue` and `queue.Queue` classes have the same interface. The only difference is that the first is designed for usage in multiprocess environments, rather than with multiple threads, so it uses different internal transports and locking primitives. We've already seen how to use `Queue` with multithreading in the *Multithreading* section, so we won't do the same for multiprocessing. The usage stays exactly the same, so such an example would not bring anything new.

A more interesting communication pattern is provided by the `Pipe` class. It is a duplex (two-way) communication channel that is very similar in concept to UNIX pipes. The interface of `Pipe` is very similar to a simple socket from the built-in `socket` module. The difference between raw system pipes and sockets is that it automatically applies object serialization through the `pickle` module. From a developer's perspective, it looks like sending ordinary Python objects. With plain system pipes or sockets, you need to apply your own serialization manually in order to reconstruct sent objects from byte streams.

> The `pickle` module can easily serialize and deserialize Python objects to and from byte streams. It handles various types of objects including instances of user-defined classes. You can learn more about the `pickle` module and which objects are picklable at `https://docs.python.org/3/library/pickle.html`.

This allows for much easier communication between processes because you can send almost any basic Python type. Consider the following `worker()` class, which will read an object from the `Pipe` object and output its representation on standard output:

```python
def worker(connection):
    while True:
        instance = connection.recv()
        if instance:
            print(f"CHLD: recv: {instance}")
        if instance is None:
            break
```

Later on, we can use the `Pipe` in our `main()` function to send various objects (including custom classes) to a child process:

```python
from multiprocessing import Process, Pipe

class CustomClass:
    pass

def main():
    parent_conn, child_conn = Pipe()

    child = Process(target=worker, args=(child_conn,))

    for item in (
        42,
        'some string',
```

```
        {'one': 1},
        CustomClass(),
        None,
    ):
        print(
            "PRNT: send: {}".format(item)
        )
        parent_conn.send(item)

    child.start()
    child.join()

if __name__ == "__main__":
    main()
```

When looking at the following example output of the preceding script, you will see that you can easily pass custom class instances and that they have different addresses, depending on the process:

```
PRNT: send: 42
PRNT: send: some string
PRNT: send: {'one': 1}
PRNT: send: <__main__.CustomClass object at 0x101cb5b00>
PRNT: send: None
CHLD: recv: 42
CHLD: recv: some string
CHLD: recv: {'one': 1}
CHLD: recv: <__main__.CustomClass object at 0x101cba400>
```

The other way to share a state between processes is to use raw types in a shared memory pool with classes provided in `multiprocessing.sharedctypes`. The most basic ones are `Value` and `Array`. Here is some example code from the official documentation of the `multiprocessing` module:

```python
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]


if __name__ == '__main__':
```

```
    num = Value('d', 0.0)
    arr = Array('i', range(10))

    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print(num.value)
    print(arr[:])
```

And this example will print the following output:

```
3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

When working with `multiprocessing.sharedctypes`, you need to remember that you are dealing with shared memory, so to avoid the risk of race conditions, you still need to use locking primitives. `multiprocessing` provides some of the classes similar to those available in the `threading` module, such as `Lock`, `RLock`, and `Semaphore`. The downside of classes from `sharedctypes` is that they allow you only to share the basic C types from the `ctypes` module. If you need to pass more complex structures or class instances, you need to use `Queue`, `Pipe`, or other inter-process communication channels instead. In most cases, it is reasonable to avoid types from `sharedctypes` because they increase code complexity and bring all the dangers of multithreading.

We've already mentioned that the `multiprocessing` module allows you to reduce the amount of boilerplate thanks to some extra functionalities. One such functionality is built-in process pools. We will take a look at how to use them in the next section.

Using process pools

Using multiple processes instead of threads adds some overhead. Mostly, it increases the memory footprint because each process has its own and independent memory context. This means allowing unbound numbers of child processes may be more of an issue than allowing an unbounded number of threads in multithreaded applications.

> If the OS supports the `fork()` system call with **copy-on-write** (**COW**) semantics, the memory overhead of starting new subprocesses will be greatly reduced. COW allows an OS to deduplicate the same memory pages and copy them only if one of the processes attempts to modify them. For instance, Linux provides the `fork()` system call with COW semantics but Windows does not. Also, COW benefits may be diminished in long-running processes.

The best pattern to control resource usage in applications that rely on multiprocessing is to build a process pool in a similar way to what we described for threads in the *Using a thread pool* section.

And the best thing about the `multiprocessing` module is that it provides a ready-to-use `Pool` class that handles all the complexity of managing multiple process workers for you. This pool implementation greatly reduces the amount of required boilerplate and the number of issues related to two-way communication. You also don't have to use the `join()` method manually, because `Pool` can be used as a context manager (using the `with` statement). Here is one of our previous threading examples, rewritten to use the `Pool` class from the `multiprocessing` module:

```python
import time
from multiprocessing import Pool

import requests

SYMBOLS = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')
BASES = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')

POOL_SIZE = 4

def fetch_rates(base):
    response = requests.get(
        f"https://api.vatcomply.com/rates?base={base}"
    )

    response.raise_for_status()
    rates = response.json()["rates"]
    # note: same currency exchanges to itself 1:1
    rates[base] = 1.
    return base, rates
```

```python
def present_result(base, rates):
    rates_line = ", ".join(
        [f"{rates[symbol]:7.03} {symbol}" for symbol in SYMBOLS]
    )
    print(f"1 {base} = {rates_line}")


def main():
    with Pool(POOL_SIZE) as pool:
        results = pool.map(fetch_rates, BASES)

    for result in results:
        present_result(*result)


if __name__ == "__main__":
    started = time.time()
    main()
    elapsed = time.time() - started

    print()
    print("time elapsed: {:.2f}s".format(elapsed))
```

As you can see, dealing with the worker pool is now simpler as we don't have to maintain our own work queues and `start()`/`join()` methods. The code would now be easier to maintain and debug in the case of issues. Actually, the only part of the code that explicitly deals with multiprocessing is the `main()` function:

```python
def main():
    with Pool(POOL_SIZE) as pool:
        results = pool.map(fetch_rates, BASES)

    for result in results:
        present_result(*result)
```

We no longer have to deal with explicit queues for passing results and we don't have to wonder what happens when one of the subprocesses raises an exception. This is a great improvement on the situation where we had to build the worker pool from scratch. Now, we don't even need to care about communication channels because they are created implicitly inside the `Pool` class implementation.

This doesn't mean that multithreading always needs to be troublesome. Let's take a look at how to use `multiprocessing.dummy` as a multithreading interface in the next section.

Using multiprocessing.dummy as the multithreading interface

The high-level abstractions from the `multiprocessing` module, such as the `Pool` class, provide great advantages over the simple tools provided in the `threading` module. But this does not mean that multiprocessing is always better than multithreading. There are a lot of use cases where threads may be a better solution than processes. This is especially true for situations where low latency and/or high resource efficiency are required.

Still, it does not mean that you need to sacrifice all the useful abstractions from the `multiprocessing` module whenever you want to use threads instead of processes. There is the `multiprocessing.dummy` module, which replicates the multiprocessing API but uses multiple threads instead of forking/spawning new processes.

This allows you to reduce the amount of boilerplate in your code and also have a more pluggable code structure. For instance, let's take yet another look at our `main()` function from the previous section. We could give the user control over which processing backend to use (processes or threads). We could do that simply by replacing the `Pool` object constructor class, as follows:

```python
from multiprocessing import Pool as ProcessPool
from multiprocessing.dummy import Pool as ThreadPool


def main(use_threads=False):
    if use_threads:
        pool_cls = ThreadPool
    else:
        pool_cls = ProcessPool

    with pool_cls(POOL_SIZE) as pool:
        results = pool.map(fetch_rates, BASES)

    for result in results:
        present_result(*result)
```

> The dummy threading pool can also be imported from the `multiprocessing.pool` module as the `ThreadPool` class. It will have the same implementation; the actual import path is just a matter of personal preference.

This aspect of the `multiprocessing` module shows that multiprocessing and multithreading have a lot in common. They both rely on the OS to facilitate concurrency. They also can be operated in a similar fashion and often utilize similar abstractions to ensure communication or memory safety.

A completely different approach to concurrency is asynchronous programming, which does not rely on any OS capabilities to ensure the concurrent processing of information. Let's take a look at this model of concurrency in the next section.

Asynchronous programming

Asynchronous programming has gained a lot of traction in the last few years. In Python 3.5, we finally got some syntax features that solidified the concepts of asynchronous execution. But this does not mean that asynchronous programming wasn't possible before Python 3.5. A lot of libraries and frameworks were provided a lot earlier, and most of them have origins in the old versions of Python 2. There is even a whole alternate implementation of Python called **Stackless** Python that concentrates on this single programming approach.

The easiest way to think about asynchronous programming in Python is to imagine something similar to threads, but without system scheduling involved. This means that an asynchronous program can concurrently process information, but the execution context is switched internally and not by the system scheduler.

But, of course, we don't use threads to concurrently handle the work in an asynchronous program. Many asynchronous programming solutions use different kinds of concepts and, depending on the implementation, they are named differently. The following are some example names that are used to describe such concurrent program entities:

- Green threads or greenlets (greenlet, gevent, or eventlet projects)
- Coroutines (Python 3.5 native asynchronous programming)
- Tasklets (Stackless Python)

> The name green threads comes from the original threads library
> for the Java language implemented by The Green Team at the Sun
> Microsystems company. Green threads were introduced in Java 1.1
> and abandoned in Java 1.3

These are mainly the same concepts but often implemented in slightly different ways.

For obvious reasons, in this section, we will concentrate only on coroutines that are natively supported by Python, starting from version 3.5.

Cooperative multitasking and asynchronous I/O

Cooperative multitasking is at the core of asynchronous programming. In this style of computer multitasking, it's not the responsibility of the OS to initiate a context switch (to another process or thread). Instead, every process voluntarily releases the control when it is idle to enable the simultaneous execution of multiple programs. This is why it is called cooperative multitasking. All processes need to cooperate in order to multitask smoothly.

This model of multitasking was sometimes employed in the OS, but now it is hardly found as a system-level solution. This is because there is a risk that one poorly designed service might easily break the whole system's stability. Thread and process scheduling with context switches managed directly by the OS is now the dominant approach for concurrency at the OS level. But cooperative multitasking is still a great concurrency tool at the application level.

When doing cooperative multitasking at the application level, we do not deal with threads or processes that need to release control because all the execution is contained within a single process and thread. Instead, we have multiple tasks (coroutines, tasklets, or green threads) that release the control to the single function that handles the coordination of tasks. This function is usually some kind of event loop.

> To avoid confusion later (due to Python terminology), from now
> on, we will refer to such concurrent tasks as coroutines.

The most important problem in cooperative multitasking is when to release the control. In most asynchronous applications, the control is released to the scheduler or event loop on I/O operations. It doesn't matter if the program reads data from the filesystem or communicates through a socket, as such I/O operations always result in some waiting time when the process becomes idle. The waiting time depends on the external resource, so it is a good opportunity to release the control so that other coroutines can do their work until they too would need to wait.

This makes such an approach somewhat similar in behavior to how multithreading is implemented in Python. We know that the GIL serializes Python threads, but it is also released on every I/O operation. The main difference is that threads in Python are implemented as system-level threads so that the OS can preempt the currently running thread and give control to the other one at any point in time. In asynchronous programming, tasks are never preempted by the main event loop and must instead return control explicitly. That's why this style of multitasking is also called non-preemptive multitasking. This reduces time lost on context switching and plays better with CPython's GIL implementation.

Of course, every Python application runs on an OS where there are other processes competing for resources. This means that the OS always has the right to preempt the whole process and give control to another process. But when our asynchronous application is running back, it continues from the same place where it was paused when the system scheduler stepped in. This is why coroutines are still considered non-preemptive.

In the next section, we will take a look at the `async` and `await` keywords, which are the backbone of cooperative multitasking in Python.

Python async and await keywords

The `async` and `await` keywords are the main building blocks in Python asynchronous programming.

The `async` keyword, when used before the `def` statement, defines a new coroutine. The execution of the coroutine function may be suspended and resumed in strictly defined circumstances. Its syntax and behavior are very similar to generators. In fact, generators need to be used in the older versions of Python whenever you want to implement coroutines. Here is an example of a function declaration that uses the `async` keyword:

```python
async def async_hello():
    print("hello, world!")
```

Functions defined with the `async` keyword are special. When called, they do not execute the code inside, but instead return a coroutine object. Consider the following example from an interactive Python session:

```python
>>> async def async_hello():
...     print("hello, world!")
...
>>> async_hello()
<coroutine object async_hello at 0x1014129e8>
```

The `coroutine` object does not do anything until its execution is scheduled in the event loop. The `asyncio` module is available in order to provide the basic event loop implementation, as well as a lot of other asynchronous utilities. The following example presents an attempt to manually schedule a coroutine execution in an interactive Python session:

```python
>>> import asyncio
>>> async def async_hello():
...     print("hello, world!")
...
>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(async_hello())
hello, world!
>>> loop.close()
```

Obviously, since we have created only one simple coroutine, there is no concurrency involved in our program. In order to see something that is actually concurrent, we need to create more tasks that will be executed by the event loop.

New tasks can be added to the loop by calling the `loop.create_task()` method or by providing an "awaitable" object to the `asyncio.wait()` function. If you have multiple tasks or coroutines to wait for, you can use `asyncio.gather()` to aggregate them into a single object. We will use the latter approach and try to asynchronously print a sequence of numbers that's been generated with the `range()` function, as follows:

```python
import asyncio
import random


async def print_number(number):
    await asyncio.sleep(random.random())
    print(number)


if __name__ == "__main__":
```

```
    loop = asyncio.get_event_loop()

    loop.run_until_complete(
        asyncio.gather(*[
            print_number(number)
            for number in range(10)
        ])
    )
    loop.close()
```

Let's save our script in an `async_print.py` file and see how it works:

```
$ python async_print.py
```

The output you will see may look as follows:

```
0
7
8
3
9
4
1
5
2
6
```

The `asyncio.gather()` function accepts multiple coroutine objects and returns immediately. It accepts a variable number of positional arguments. That's why we used the argument unpacking syntax (the * operator) to unpack the list of coroutines as arguments. As the name suggests, `asyncio.gather()` is used to gather multiple coroutines to execute them concurrently. The result is an object that represents a future result (a so-called **future**) of running all of the provided coroutines. The `loop.run_until_complete()` method runs the event loop until the given future is completed.

We used `asyncio.sleep(random.random())` to emphasize the asynchronous operation of coroutines. Thanks to this, coroutines can interweave with each other.

We couldn't achieve the same result (that is, the interweaving of coroutines) with an ordinary `time.sleep()` function. Coroutines can start to interweave when they release control of execution. This is done through the `await` keyword. It suspends the execution of the coroutine that is waiting for the results of another coroutine or future.

Whenever a function awaits, it releases the control over execution to the event loop. To better understand how this works, we need to review a more complex example of code.

Let's say we want to create two coroutines that will both perform the same simple task in a loop:

- Wait a random number of seconds
- Print some text provided as an argument and the amount of time spent in sleep

Let's start with the following simple implementation that does not use the `await` keyword:

```python
import time
import random

async def waiter(name):
    for _ in range(4):
        time_to_sleep = random.randint(1, 3) / 4
        time.sleep(time_to_sleep)
        print(f"{name} waited { time_to_sleep } seconds")
```

We can schedule the execution of multiple `waiter()` coroutines using `asyncio.gather()` the same way as we did in the `async_print.py` script:

```python
import asyncio

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(
        asyncio.gather(waiter("first"), waiter("second"))
    )
    loop.close()
```

Let's save the code in the `waiters.py` file and see how those two `waiter()` coroutines execute in the event loop:

```
$ time python3 waiters.py
```

Note that we've used the `time` utility to measure total execution time. The preceding execution can give the following output:

```
$ time python waiters.py
first waited 0.25 seconds
first waited 0.75 seconds
first waited 0.5 seconds
first waited 0.25 seconds
second waited 0.75 seconds
second waited 0.5 seconds
second waited 0.5 seconds
second waited 0.75 seconds

real    0m4.337s
user    0m0.050s
sys     0m0.014s
```

As we can see, both the coroutines completed their execution, but not in an asynchronous manner. The reason is that they both use the `time.sleep()` function, which is blocking but not releasing the control to the event loop. This would work better in a multithreaded setup, but we don't want to use threads now. So, how can we fix this?

The answer is to use `asyncio.sleep()`, which is the asynchronous version of `time.sleep()`, and await its result using the `await` keyword. Let's see the following improved version of the `waiter()` coroutine, which uses the `await asyncio.sleep()` statement:

```python
async def waiter(name):
    for _ in range(4):
        time_to_sleep = random.randint(1, 3) / 4
        await asyncio.sleep(time_to_sleep)
        print(f"{name} waited {time_to_sleep} seconds")
```

If we save a modified version of this script in the `waiters_await.py` file and execute it in the shell, we will hopefully see how the outputs of the two functions interweave with each other:

```
$ time python waiters_await.py
```

The output you will see should look something like the following:

```
first waited 0.5 seconds
second waited 0.75 seconds
second waited 0.25 seconds
first waited 0.75 seconds
second waited 0.75 seconds
first waited 0.75 seconds
second waited 0.75 seconds
first waited 0.5 seconds


real    0m2.589s
user    0m0.053s
sys     0m0.016s
```

The additional advantage of this simple improvement is that the code ran faster. The overall execution time was less than the sum of all sleeping times because coroutines were cooperatively releasing the control.

Let's take a look at a more practical example of asynchronous programming in the next section.

A practical example of asynchronous programming

As we have already mentioned multiple times in this chapter, asynchronous programming is a great tool for handling I/O-bound operations. So, it's time to build something more practical than a simple printing of sequences or asynchronous waiting.

For the sake of consistency, we will try to handle the same problem that we solved previously with the help of multithreading and multiprocessing. So, we will try to asynchronously fetch some information about current currency exchange rates from an external resource through a network connection. It would be great if we could use the same `requests` library as in the previous sections. Unfortunately, we can't do so. Or to be more precise, we can't do so effectively.

Unfortunately, the `requests` library does not support asynchronous I/O with the `async` and `await` keywords. There are some other projects that aim to provide some concurrency to the `requests` project, but they either rely on Gevent (like `grequests`, available at `https://github.com/kennethreitz/grequests`) or thread/process pool execution (like `requests-futures`, available at `https://github.com/ross/requests-futures`). Neither of these solves our problem.

Knowing the limitation of the library that was so easy to use in our previous examples, we need to build something that will fill the gap. The foreign exchange rates API is really simple to use, so we just need to use a natively asynchronous HTTP library for the job. The standard library of Python in version 3.9 still lacks any library that would make asynchronous HTTP requests as simple as calling `urllib.urlopen()`. We definitely don't want to build the whole protocol support from scratch, so we will use a little help from the `aiohttp` package, which is available on PyPI. It's a really promising library that adds both client and server implementations for asynchronous HTTP. Here is a small module built on top of `aiohttp` that creates a single `get_rates()` helper function that makes requests to the foreign exchange rates API service:

```python
import aiohttp


async def get_rates(session: aiohttp.ClientSession, base: str):
    async with session.get(
        f"https://api.vatcomply.com/rates?base={base}"
    ) as response:
        rates = (await response.json())['rates']
        rates[base] = 1.

        return base, rates
```

We will save that module in the `asyncrates.py` file so later we will be able to import it as the `asyncrates` module.

Now, we are ready to rewrite the example used when we discussed multithreading and multiprocessing. Previously, we split the whole operation into the following two separate steps:

- Perform all requests to an external service in parallel using the `asyncrates.get_rates()` function
- Display all the results in a loop using the `present_result()` function

The core of our program will be a simple `main()` function that gathers results from multiple `get_rates()` coroutines and passes them to the `present_result()` function:

```python
async def main():
    async with aiohttp.ClientSession() as session:
        for result in await asyncio.gather(*[
            get_rates(session, base)
```

```
            for base in BASES
        ]):
            present_result(*result)
```

And the full code, together with imports and event loop initialization, will be as follows:

```python
import asyncio
import time

import aiohttp

from asyncrates import get_rates

SYMBOLS = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')
BASES = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')


def present_result(base, rates):
    rates_line = ", ".join(
        [f"{rates[symbol]:7.03} {symbol}" for symbol in SYMBOLS]
    )
    print(f"1 {base} = {rates_line}")


async def main():
    async with aiohttp.ClientSession() as session:
        for result in await asyncio.gather(*[
            get_rates(session, base)
            for base in BASES
        ]):
            present_result(*result)


if __name__ == "__main__":
    started = time.time()
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
    elapsed = time.time() - started

    print()
    print("time elapsed: {:.2f}s".format(elapsed))
```

The output of running this program will be similar to the output of versions that relied on multithreading and multiprocessing:

```
$ python async_aiohttp.py
1 USD =     1.0 USD,   0.835 EUR,    3.81 PLN,    8.39 NOK,    21.7 CZK
1 EUR =     1.2 USD,     1.0 EUR,    4.56 PLN,    10.0 NOK,    25.9 CZK
1 PLN =   0.263 USD,    0.22 EUR,     1.0 PLN,     2.2 NOK,    5.69 CZK
1 NOK =   0.119 USD,  0.0996 EUR,   0.454 PLN,     1.0 NOK,    2.58 CZK
1 CZK =  0.0461 USD,  0.0385 EUR,   0.176 PLN,   0.387 NOK,     1.0 CZK

time elapsed: 0.33s
```

The advantage of using `asyncio` over multithreading and multiprocessing is that we didn't have to deal with process pools and memory safety to achieve concurrent network communication. The downside is that we couldn't use a popular synchronous communication library like the `requests` package. We used `aiohttp` instead, and that's fairly easy for a simple API. But sometimes, you need a specialized client library that isn't asynchronous and cannot be easily ported. We will cover such a situation in the next section.

Integrating non-asynchronous code with async using futures

Asynchronous programming is great, especially for backend developers interested in building scalable applications. In practice, it is one of the most important tools for building highly concurrent servers.

But the reality is painful. A lot of popular packages that deal with I/O-bound problems are not meant to be used with asynchronous code. The main reasons for that are as follows:

- The low adoption of advanced Python 3 features (especially asynchronous programming)
- The low understanding of various concurrency concepts among Python beginners

This means that often, the migration of existing synchronous multithreaded applications and packages is either impossible (due to architectural constraints) or too expensive. A lot of projects could benefit greatly from incorporating the asynchronous style of multitasking, but only a few of them will eventually do that.

This means that right now, you will experience a lot of difficulties when trying to build asynchronous applications from scratch. In most cases, this will be something similar to the problem of the `requests` library mentioned in the *A practical example of asynchronous programming* section—incompatible interfaces and the synchronous blocking of I/O operations.

Of course, you can sometimes resign from `await` when you experience such incompatibility and just fetch the required resources synchronously. But this will block every other coroutine from executing its code while you wait for the results. It technically works but also ruins all the gains of asynchronous programming. So, in the end, joining asynchronous I/O with synchronous I/O is not an option. It is kind of an all-or-nothing game.

The other problem is long-running CPU-bound operations. When you are performing an I/O operation, it is not a problem to release control from a coroutine. When writing/reading from a socket, you will eventually wait, so using `await` is the best you can do. But what should you do when you need to actually compute something, and you know it will take a while? You can, of course, slice the problem into parts and release control with `asyncio.wait(0)` every time you move the work forward a bit. But you will shortly find that this is not a good pattern. Such a thing will make the code a mess, and also does not guarantee good results. Time slicing should be the responsibility of the interpreter or OS.

So, what should you do if you have some code that makes long synchronous I/O operations that you can't or are unwilling to rewrite? Or what should you do when you have to make some heavy CPU-bound operations in an application designed mostly with asynchronous I/O in mind? Well... you need to use a workaround. And by a workaround, I mean multithreading or multiprocessing.

This may not sound obvious, but sometimes the best solution may be the one that we tried to escape from. Parallel processing of CPU-intensive tasks in Python is always better with multiprocessing. And multithreading may deal with I/O operations equally as well (quickly and without a lot of resource overhead) as `async` and `await`, if you set it up properly and handle it with care.

So, when something simply does not fit your asynchronous application, use a piece of code that will defer it to a separate thread or process. You can pretend that this was a coroutine and release control to the event loop using `await`. You will eventually process results when they are ready. Fortunately for us, the Python standard library provides the `concurrent.futures` module, which is also integrated with the `asyncio` module. These two modules together allow you to schedule blocking functions to execute in threads or additional processes as if they were asynchronous non-blocking coroutines.

Let's take a closer look at executors and futures in the next section.

Executors and futures

Before we see how to inject threads or processes into an asynchronous event loop, we will take a closer look at the `concurrent.futures` module, which will later be the main ingredient of our so-called workaround. The most important classes in the `concurrent.futures` module are `Executor` and `Future`.

Executor represents a pool of resources that may process work items in parallel. This may seem very similar in purpose to classes from the `multiprocessing` module — `Pool` and `dummy.Pool` — but it has a completely different interface and semantics. The `Executor` class is a base class not intended for instantiation and has the following two concrete implementations:

- `ThreadPoolExecutor`: This is the one that represents a pool of threads
- `ProcessPoolExecutor`: This is the one that represents a pool of processes

Every executor provides the following three methods:

- `submit(func, *args, **kwargs)`: This schedules the `func` function for execution in a pool of resources and returns the `Future` object representing the execution of a callable
- `map(func, *iterables, timeout=None, chunksize=1)`: This executes the `func` function over an iterable in a similar way to the `multiprocessing.Pool.map()` method
- `shutdown(wait=True)`: This shuts down the executor and frees all of its resources

The most interesting method is `submit()` because of the `Future` object it returns. It represents the asynchronous execution of the callable and only indirectly represents its result. In order to obtain the actual return value of the submitted callable, you need to call the `Future.result()` method. And if the callable has already finished, the `result()` method will not block and will just return the function output. If it is not true, it will block until the result is ready. Treat it like a promise of a result (actually, it is the same concept as a promise in JavaScript). You don't need to unpack it immediately after receiving it (with the `result()` method), but if you try to do that, it is guaranteed to eventually return something.

Let's consider the following interaction with `ThreadPoolExecutor` in an interactive Python session:

```python
>>> def loudly_return():
...     print("processing")
...     return 42
...
```

```
>>> from concurrent.futures import ThreadPoolExecutor
>>> with ThreadPoolExecutor(1) as executor:
...     future = executor.submit(loudly_return)
...
processing
>>> future
<Future at 0x33cbf98 state=finished returned int>
>>> future.result()
42
```

As you can see, `loudly_return()` immediately printed the processing string after it was submitted to the executor. This means that execution started even before we decided to unpack its value using the `future.result()` method.

In the next section, we'll see how to use executors in an event loop.

Using executors in an event loop

The `Future` class instances returned by the `Executor.submit()` method are conceptually very close to the coroutines used in asynchronous programming. This is why we can use executors to make a hybrid between cooperative multitasking and multiprocessing or multithreading.

The core of this workaround is the `run_in_executor(executor, func, *args)` method of the event loop class. It allows you to schedule the execution of the `func` function in the process or thread pool represented by the executor argument. The most important thing about that method is that it returns a new awaitable (an object that can be awaited with the `await` statement). So, thanks to this, you can execute a blocking function that is not a coroutine exactly as if it was a coroutine. And most importantly, it will not block the event loop from processing other coroutines, no matter how long it will take to finish. It will stop only the function that is awaiting results from such a call, but the whole event loop will still keep spinning.

And a useful fact is that you don't even need to create your executor instance. If you pass `None` as an executor argument, the `ThreadPoolExecutor` class will be used with the default number of threads (for Python 3.9, it is the number of processors multiplied by 5).

So, let's assume that we did not want to rewrite the problematic part of our API-facing code that was the cause of our headache. We can easily defer the blocking call to a separate thread with the `loop.run_in_executor()` call, while still leaving the `fetch_rates()` function as an awaitable coroutine, as follows:

```python
async def fetch_rates(base):
    loop = asyncio.get_event_loop()
    response = await loop.run_in_executor(
        None, requests.get,
        f"https://api.vatcomply.com/rates?base={base}"
    )
    response.raise_for_status()
    rates = response.json()["rates"]
    # note: same currency exchanges to itself 1:1
    rates[base] = 1.
    return base, rates
```

Such a solution is not as good as having a fully asynchronous library to do the job, but half a loaf is better than none.

Asynchronous programming is a great tool for building performant concurrent applications that have to communicate a lot with other services over the network. You can do that easily without all the memory safety problems that usually come with multithreading and (to some extent) multiprocessing. A lack of involuntary context switching also reduces the number of necessary locking primitives because it is easy to predict when coroutines return control to the event loop.

Unfortunately, that comes at the cost of having to use dedicated asynchronous libraries. Synchronous and threaded applications usually have better coverage of client and communication libraries for interacting with popular services. Executors and futures allow you to fill that gap but are less optimal than native asynchronous solutions.

Summary

It was a long journey, but we successfully struggled through most of the common approaches to concurrent programming that are available for Python programmers.

After explaining what concurrency really is, we jumped into action and dissected one of the typical concurrent problems with the help of multithreading. After identifying the basic deficiencies of our code and fixing them, we turned to multiprocessing to see how it would work in our case. We found that multiple processes with the `multiprocessing` module are a lot easier to use than plain threads coming with the `threading` module. But just after that, we realized that we can use the same API for threads too, thanks to the `multiprocessing.dummy` module. So, the decision between multiprocessing and multithreading is now only a matter of which solution better suits the problem and not which solution has a better interface.

And speaking about problem fit, we finally tried asynchronous programming, which should be the best solution for I/O-bound applications, only to realize that we cannot completely forget about threads and processes. So, we made a circle! Back to the place where we started.

And this leads us to the final conclusion of this chapter. There is no silver bullet. There are some approaches that you may prefer. There are some approaches that may fit better for a given set of problems, but you need to know them all in order to be successful. In real-life scenarios, you may find yourself using the whole arsenal of concurrency tools and styles in a single application, and this is not uncommon.

In the next chapter, we will take a look at a topic somewhat related to concurrency: event-driven programming. In that chapter, we will be concentrating on various communication patterns that form the backbone of distributed asynchronous and highly concurrent systems.

7
Event-Driven Programming

In the previous chapter, we discussed various concurrency implementation models that are available in Python. To better explain the concept of concurrency, we used the following definition:

Two events are concurrent if neither can causally affect the other.

We often think about events as ordered points in time that happen one after another, often with some kind of cause-effect relationship. But, in programming, events are understood a bit differently. They are not necessarily "things that happen." Events in programming are more often understood as independent units of information that can be processed by the program. And that very notion of events is a real cornerstone of concurrency.

Concurrent programming is a programming paradigm for processing concurrent events. And there is a generalization of that paradigm that deals with the bare concept of events—no matter whether they are concurrent or not. This approach to programming, which treats programs as a flow of events, is called **event-driven programming**.

It is an important paradigm because it allows you to easily decouple even large and complex systems. It helps in defining clear boundaries between independent components and improves isolation between them.

In this chapter, we will cover the following topics:

- What exactly is event-driven programming?
- Various styles of event-driven programming
- Event-driven architectures

After reading this chapter, you will know the common techniques of event-driven programming and how to extrapolate these techniques to event-driven architectures. You'll also be able to easily identify problems that can be solved using event-driven programs.

Technical requirements

The following are Python packages that are mentioned in this chapter that you can download from PyPI:

- `flask`
- `blinker`

Information on how to install packages is included in *Chapter 2*, *Modern Python Development Environments*.

The code files for this chapter can be found at `https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition/tree/main/Chapter%207`.

In this chapter, we will build a small application using a **Graphical User Interface** (**GUI**) package named `tkinter`. To run the `tkinter` examples, you will need the Tk library for Python. It should be available by default with most Python distributions, but on some operating systems, it will require additional system packages to be installed. On Debian-based Linux distributions, this package is usually named `python3-tk`. Python installed though official macOS and Windows installers should already come with the Tk library.

What exactly is event-driven programming?

Event-driven programming focuses on the events (often called **messages**) and their flow between different software components. In fact, it can be found in many types of software. Historically, event-based programming is the most common paradigm for software that deals with direct human interaction. It means that it is a natural paradigm for GUIs. Anywhere the program needs to wait for some human input, that input can be modeled as events or messages. In such a framing, an event-driven program is often just a collection of event/message handlers that respond to human interaction.

Events of course don't have to be a direct result of user interaction. The architecture of any web application is also event-driven. Web browsers send requests to web servers on behalf of the user, and these requests are often processed as separate interaction events. Some of the requests will indeed be the result of direct user input (for example, submitting a form or clicking on a link), but don't always have to be. Many modern applications can asynchronously synchronize information with a web server without any interaction from the user, and that communication happens silently without the user noticing.

In summary, event-driven programming is a general way of coupling software components of various sizes and happens on various levels of software architecture. Depending on the scale and type of software architecture we're dealing with, it can take various forms:

- It can be a concurrency model directly supported by a semantic feature of a given programming language (for example, `async`/`await` in Python)
- It can be a way of structuring application code with event dispatchers/ handlers, signals, and so on
- It can be a general inter-process or inter-service communication architecture that allows for the coupling of independent software components in a larger system

Let's discuss how event-driven programming is different from asynchronous programming in the next section.

Event-driven != asynchronous

Although event-driven programming is a paradigm that is extremely common for asynchronous systems, it doesn't mean that every event-driven application must be asynchronous. It also doesn't mean that event-driven programming is suited only for concurrent and asynchronous applications. Actually, the event-driven approach is extremely useful, even for decoupling problems that are strictly synchronous and definitely not concurrent.

Consider, for instance, database triggers, which are available in almost every relational database system. A database trigger is a stored procedure that is executed in response to a certain event that happens in the database. This is a common building block of database systems that, among other things, allows the database to maintain data consistency in scenarios that cannot be easily modeled with the mechanism of database constraints.

For instance, the PostgreSQL database distinguishes three types of row-level events that can occur in either a table or a view:

- INSERT: Emitted when a new row is inserted
- UPDATE: Emitted when an existing row is updated
- DELETE: Emitted when an existing row is deleted

In the case of table rows, triggers can be defined to be executed either BEFORE or AFTER a specific event. So, from the perspective of event-procedure coupling, we can treat each AFTER/BEFORE trigger as a separate event. To better understand this, let's consider the following example of database triggers in PostgreSQL:

```
CREATE TRIGGER before_user_update
    BEFORE UPDATE ON users
    FOR EACH ROW
    EXECUTE PROCEDURE check_user();

CREATE TRIGGER after_user_update
    AFTER UPDATE ON users
    FOR EACH ROW
    EXECUTE PROCEDURE log_user_update();
```

In the preceding example, we have two triggers that are executed when a row in the users table is updated. The first one is executed before a real update occurs and the second one is executed after the update is done. This means that BEFORE UPDATE and AFTER UPDATE events are casually dependent and cannot be handled concurrently. On the other hand, similar sets of events occurring on different rows from different sessions can still be concurrent, although that will depend on multiple factors (transaction or not, the isolation level, the scope of the trigger, and so on). This is a valid example of a situation where data modification in a database system can be modeled with event-based processing although the system as a whole isn't fully asynchronous.

In the next section, we'll take a look at event-driven programming in GUIs.

Event-driven programming in GUIs

GUIs are what many people think of when they hear the term "event-driven programming." Event-driven programming is an elegant way of coupling user input with code in GUIs because it naturally captures the way people interact with graphical interfaces. Such interfaces often present the user with a plethora of components to interact with, and that interaction is almost always nonlinear.

In complex interfaces, this interaction is often modeled through a collection of events that can be emitted by the user from different interface components.

The concept of events is common to most user interface libraries and frameworks, but different libraries use different design patterns to achieve event-driven communication. Some libraries even use other notions to describe their architecture (for example, signals in the Qt library). Still, the general pattern is almost always the same—every interface component (often called a **widget**) can emit events upon interaction. Other components receive those events either by subscription or by directly attaching themselves to emitters as their event handlers. Depending on the GUI library, events can just be plain named signals stating that something has happened (for example, "widget A was clicked"), or they can be more complex messages containing additional information about the nature of the interaction. Such messages can for instance contain the specific key that has been pressed or the position of the mouse when an event was emitted.

We will discuss the differences of actual design patterns later in the *Various styles of event-driven programming* section, but first let's take a look at the example Python GUI application that can be created with the use of the built-in `tkinter` module:

> Note that the Tk library that powers the `tkinter` module is usually bundled with Python distributions. If it's somehow not available on your operating system, you should be easily able to install it through your system package manager. For instance, on Debian-based Linux distributions, you can easily install it for Python as the `python3-tk` package using the following command:
>
> ```
> sudo apt-get install python3-tk
> ```

The following GUI application displays a single **Python Zen** button. When the button is clicked, the application will open a new window containing the *Zen of Python* text that was imported from the `this` module. The `this` module is a Python easter egg. After import, it prints on standard output the 19 aphorisms that are the guiding principles of Python's design.

```python
import this
from tkinter import Tk, Frame, Button, LEFT, messagebox


rot13 = str.maketrans(
    "ABCDEFGHIJKLMabcdefghijklmNOPQRSTUVWXYZnopqrstuvwxyz",
    "NOPQRSTUVWXYZnopqrstuvwxyzABCDEFGHIJKLMabcdefghijklm"
)


def main_window(root: Tk):
```

```
    frame = Frame(root)
    frame.pack()

    zen_button = Button(frame, text="Python Zen", command=show_zen)
    zen_button.pack(side=LEFT)


def show_zen():
    messagebox.showinfo("Zen of Python", this.s.translate(rot13))

if __name__ == "__main__":
    root = Tk()
    main_window(root)
    root.mainloop()
```

Our script starts with imports and the definition of a simple string translation table. It is necessary because the text of the Zen of Python is encrypted inside the `this` module using an **ROT13** letter substitution cipher (also known as a Caesar cipher). It is a simple encryption algorithm that shifts every letter in the alphabet by 13 positions.

The binding of events happens directly in the `Button` widget constructor:

```
Button(frame, text="Python Zen", command=show_zen)
```

The `command` keyword argument defines the event handler that will be executed when the user clicks the button. In our example, we have provided the `show_zen()` function, which will display the decoded text of the *Zen of Python* in a separate message box.

> Every `tkinter` widget offers also a `bind()` method that can be used to define the handlers of very specific events, like mouse press/release, hover, and so on.

Most GUI frameworks work in a similar manner—you rarely work with raw keyboard and mouse inputs, but instead attach your commands/callbacks to higher-level events such as the following:

- Checkbox state change
- Button clicked

- Option selected
- Window closed

In the next section, we'll take a look at event-driven communication.

Event-driven communication

Event-driven programming is a very common practice for building distributed network applications. With event-driven programming, it is easier to split complex systems into isolated components that have a limited set of responsibilities, and because of that, it is especially popular in service-oriented and microservice architectures. In such architectures, the flow of events happens not between classes or functions living inside of a single computer process, but between many networked services. In large and distributed architectures, the flow of events between services is usually coordinated using special communication protocols (for example, AMQP and ZeroMQ), often with the help of dedicated services acting as message brokers. We will discuss some of these solutions later in the *Event-driven architectures section*.

However, you don't need to have a formalized way of coordinating events, nor a dedicated event-handling service, to consider your networked code an event-based application. Actually, if you take a more detailed look at a typical Python web application, you'll notice that most Python web frameworks have many things in common with GUI applications. Let's, for instance, consider a simple web application that was written using the Flask microframework:

```python
import this

from flask import Flask

app = Flask(__name__)

rot13 = str.maketrans(
    "ABCDEFGHIJKLMabcdefghijklmNOPQRSTUVWXYZnopqrstuvwxyz",
    "NOPQRSTUVWXYZnopqrstuvwxyzABCDEFGHIJKLMabcdefghijklm"
)


def simple_html(body):
    return f"""
    <!DOCTYPE html>
    <html lang="en">
      <head>
```

```
        <meta charset="utf-8">
        <title>Book Example</title>
      </head>
      <body>
        {body}
      </body>
    </html>
    """


@app.route('/')
def hello():
    return simple_html("<a href=/zen>Python Zen</a>")


@app.route('/zen')
def zen():
    return simple_html(
        "<br>".join(this.s.translate(rot13).split("\n"))
    )

if __name__ == '__main__':
    app.run()
```

> We discussed examples of writing and executing simple Flask applications in *Chapter 2*, *Modern Python Development Environments*.

If you compare the preceding listing with the example of the `tkinter` application from the previous section, you'll notice that, structurally, they are very similar. Specific routes (paths) of HTTP requests translate to dedicated handlers. If we consider our application to be event-driven, then the request path can be treated as a binding between a specific event type (for example, a link being clicked) and the action handler. Similar to events in GUI applications, HTTP requests can contain additional data about interaction context. This information is, of course, structured. The HTTP protocol defines multiple request methods (for example, POST, GET, PUT, and DELETE) and a few ways to transfer additional data (query string, request body, and headers).

The user does not communicate with our application directly as they would when using a GUI, but instead they use a web browser as their interface. This also makes it somewhat similar to traditional graphical applications, as many cross-platform user interface libraries (such as Tcl/Tk, Qt, and GTK+) are in fact just proxies between the application and the user's operating system's windowing APIs. So, in both cases, we deal with communication and events flowing through multiple system layers. It is just that, in web applications, layers are more evident and communication is always explicit.

Modern web applications often provide interactive interfaces based on JavaScript. They are very often built using event-driven frontend frameworks that communicate asynchronously with an application backend service through backend APIs. This only emphasizes the event-driven nature of web applications.

We've seen so far that depending on the use case, event-driven programming can be used in multiple types of applications. It can also take different forms. In the next section, we will go through the three major styles of event-driven programming.

Various styles of event-driven programming

As we already stated, event-driven programming can be implemented at various levels of software architecture. It is also often applied to very specific software engineering areas, such as networking, system programming, and GUI programming. So, event-driven programming isn't a single cohesive programming approach, but rather a collection of diverse patterns, tools, and algorithms that form a common paradigm that concentrates on programming around the flow of events.

Due to this, event-driven programming exists in different flavors and styles. The actual implementations of event-driven programming can be based on different design patterns and techniques. Some of these event-driven techniques and tools don't even use the term event. Despite this variety, we can easily identify three major event-driven programming styles that are the foundation for more concrete patterns:

- **Callback-based style**: This concentrates on the act of coupling event emitters with their handlers in a one-to-one fashion. In this style, event emitters are responsible for defining actions that will happen when a specific event occurs.

- **Subject-based style**: This concentrates on the one-to-many subscription of events originating at specific emitters. In this style, emitters are subjects of a subscription. Whoever wants to receive events needs to subscribe directly to the source of events.

- **Topic-based style**: This concentrates on the types of events rather than their origin and destination. In this style, event emitters are not necessarily aware of event subscribers and vice versa. Instead, communication happens through independent event channels—topics—that anyone can publish to or subscribe to.

In the next sections, we will do a brief review of the three major styles of event-driven programming that you may encounter when programming in Python.

Callback-based style

The **callback-based style** of event programming is one of the most common styles of event-driven programming. In this style, objects that emit events are the ones that are responsible for defining their event handlers. This means a one-to-one or (at most) many-to-one relationship between event emitters and event handlers.

This style of event-based programming is the dominant pattern among GUI frameworks and libraries. The reason for this is simple—it really captures how both users and programmers think about user interfaces. For every action we do, whether we toggle a switch, press a button, or tick a checkbox, we do it usually with a clear and single purpose.

We've already seen an example of callback-based event-driven programming and discussed an example of a graphical application written using the `tkinter` library (see the *Event-driven programming in GUIs* section). Let's recall one line from that application listing:

```
zen_button = Button(root, text="Python Zen", command=show_zen)
```

The previous instantiation of the `Button` class defines that the `show_zen()` function should be called whenever the button is pressed. Our event is implicit. The `show_zen()` callback (in `tkinter`, callbacks are called **commands**) does not receive any object that would describe the event behind the call. This makes sense because the responsibility of attaching event handlers lies closer to the event emitter. Here, it is the `zen_button` instance. The event handler is barely concerned about the actual origin of the event.

In some implementations of callback-based event-driven programming, the actual binding between event emitters and event handlers is a separate step that can be performed after the event emitter is initialized. This style of binding is possible in `tkinter` too, but only for raw user interaction events. The following is an updated excerpt of the previous `tkinter` application that uses this style of event binding:

```python
def main_window(root):
    frame = Frame(root)

    zen_button = Button(frame, text="Python Zen")
    zen_button.bind("<ButtonRelease-1>", show_zen)
    zen_button.pack(side=LEFT)


def show_zen(event):
    messagebox.showinfo("Zen of Python", this.s.translate(rot13))
```

In the preceding example, the event is no longer implicit. Because of that, the `show_zen()` callback must be able to accept the `event` object. The `event` instance contains basic information about user interaction, such as the position of the mouse cursor, the time of the event, and the associated widget. What is important to remember is that this type of event binding is still unicast. This means that one event from one object can be bound to only one callback. It is possible to attach the same handler to multiple events and/or multiple objects, but a single event that comes from a single source can be dispatched to only one callback. Any attempt to attach a new callback using the `bind()` method will override the old one.

The unicast nature of callback-based event programming has obvious limitations as it requires the tight coupling of application components. The inability to attach multiple fine-grained handlers to single events often means that every handler is usually specialized to serve a single emitter and cannot be bound to objects of a different type.

The subject-based style is a style that reverses the relationship between event emitters and event handlers. Let's take a look at it in the next section.

Subject-based style

The **subject-based style** of event programming is a natural extension of unicast callback-based event handling. In this style of programming, event emitters (subjects) allow other objects to subscribe/register for notifications about their events. In practice, this is very similar to the callback-based style, as event emitters usually store a list of functions or methods to call when some new event happens.

In subject-based event programming, the focus moves from the event to the subject (the event emitter). The most common product of this style is the **observer** design pattern.

In short, the observer design pattern consists of two classes of objects—observers and subjects (sometimes called **observables**). The `Subject` instance is an object that maintains a list of `Observer` instances that are interested in what happens to the `Subject` instance. In other words, `Subject` is an event emitter and `Observer` instances are event handlers.

If we would like to define common interfaces for the observer design pattern, we could do that by creating the following abstract base classes:

```python
from abc import ABC, abstractmethod

class ObserverABC(ABC):
    @abstractmethod
    def notify(self, event): ...

class SubjectABC(ABC):
    @abstractmethod
    def register(self, observer: ObserverABC): ...
```

The instances of the `ObserverABC` subclasses will be the event handlers. We will be able to register them as observers of subject events using the `register()` method of the `SubjectABC` subclass instances. What is interesting about this design is that it allows for multicast communication between components. A single observer can be registered in multiple subjects and a single subject can have multiple subscribers.

To better understand the potential of this mechanism, let's build a more practical example. We will try to build a naïve implementation of a grep-like utility. It will be able to recursively scan through the filesystem looking for files containing some specified text. We will use the built-in `glob` module for the recursive traversal of the filesystem and the `re` module for matching regular expressions.

The core of our program will be the `Grepper` class, which will be a subclass of `SubjectABC`. Let's start by defining the base scaffolding for the registration and notification of observers:

```python
class Grepper(SubjectABC):
    _observers: list[ObserverABC]

    def __init__(self):
        self._observers = []
```

```
    def register(self, observer: ObserverABC):
        self._observers.append(observer)

    def notify_observers(self, event):
        for observer in self._observers:
            observer.notify(event)
```

The implementation is fairly simple. The __init__() function initializes an empty list of observers. Every new `Grepper` instance will start with no observers. The `register()` method was defined in the `SubjectABC` class as an abstract method, so we are obliged to provide the actual implementation of it. It is the only method that is able to add new observers to the subject state. Last is the `notify_observers()` method, which will pass the specified event to all registered observers.

Since our scaffolding is ready, we are now able to define the `Grepper.grep()` method, which will do the actual work:

```
from glob import glob
import os.path
import re

class Grepper(SubjectABC):
    ...

    def grep(self, path: str, pattern: str):
        r = re.compile(pattern)

        for item in glob(path, recursive=True):
            if not os.path.isfile(item):
                continue

            try:
                with open(item) as f:
                    self.notify_observers(("opened", item))
                    if r.findall(f.read()):
                        self.notify_observers(("matched", item))
            finally:
                self.notify_observers(("closed", item))
```

The `glob(pattern, recursive=True)` function allows us to do recursive filesystem path names search with "glob" patterns. We will use it to iterate over files in the location designated by the user. For searching through actual file contents, we use regular expressions provided in the `re` module.

As we don't know at this point what the possible observer use cases are, we decided to emit three types of events:

- `"opened"`: Emitted when a new file has been opened
- `"matched"`: Emitted when Grepper found a match in a file
- `"closed"`: Emitted when a file has been closed

Let's save that class in a file named `observers.py` and finish it with the following fragment of code, which initializes the `Grepper` class instance with input arguments:

```python
import sys

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("usage: program PATH PATTERN")
        sys.exit(1)

    grepper = Grepper()
    grepper.grep(sys.argv[1], sys.argv[2])
```

Our `observers.py` program is now able to search through files, but it won't output any visible output yet. If we would like to find out which file contents match our expression, we could change it by creating a subscriber that is able to respond to`"matched"` events. The following is an example of a `Presenter` subscriber that simply prints the name of the file associated with a `"matched"` event:

```python
class Presenter(ObserverABC):
    def notify(self, event):
        event_type, file = event
        if event_type == "matched":
            print(f"Found in: {file}")
```

And here is how it could be attached to the `Grepper` class instance:

```python
if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("usage: program PATH PATTERN")
        sys.exit(1)
```

```
    grepper = Grepper()
    grepper.register(Presenter())
    grepper.grep(sys.argv[1], sys.argv[2])
```

If we would like to find out which of the examples from this chapter's code bundle contain the substring `grep`, we could use the following program invocation:

```
$ python observers.py 'Chapter 7/**' grep
Found in: Chapter 7/04 - Subject-based style/observers.py
```

The main benefit of this design pattern is extensibility. We can easily extend our application capabilities by introducing new observers. If, for instance, we would like to trace all opened files, we could create a special `Auditor` subscriber that logs all opened and closed files. It could be as simple as the following:

```python
class Auditor(ObserverABC):
    def notify(self, event):
        event_type, file = event
        print(f"{event_type:8}: {file}")
```

Moreover, observers aren't tightly coupled to the subject and have only minimal assumptions on the nature of events delivered to them. If you decide to use a different matching mechanism (for instance, the `fnmatch` module for glob-like patterns instead of regular expressions from the `re` module), you can easily reuse existing observers by registering them to a completely new subject class.

Subject-based event programming allows for the looser coupling of components and thus increases application modularity. Unfortunately, the change of focus from events to subjects can become a burden. In our example, observers will be notified about every event emitted from the `Subject` class. They have no option to register for only specific types and we've seen already how the `Presenter` class had filtered out events other than `"matched"`.

It is either the observer that must filter all incoming events or the subject that should allow observers to register for specific events at the source. The first approach will be inefficient if the number of events filtered out by every subscriber is large enough. The second approach may make the observer registration and event dispatch overly complex.

Despite the finer granularity of handlers and multicast capabilities, the subject-based approach to event programming rarely makes the application components more loosely coupled than the callback-based approach. This is why it isn't a good choice for the overall architecture of large applications, but rather a tool for specific problems.

This is mostly due to the focus on subjects, which requires all handlers to maintain a lot of assumptions about the observed subjects. Also, in the implementation of that style (that is, the observer design pattern), both observers and subjects must, at some point, meet in the same context. In other words, observers cannot register to events if there is no actual subject that would emit them.

Fortunately, there is a style of event-driven programming that allows fine-grained multicast event handling in a way that really fosters loose coupling of large applications. It is a topic-based style and is a natural evolution of subject-based event programming.

Topic-based style

Topic-based event programming concentrates on the types of events that are passed between software components without skewing toward either side of the emitter-handler relationship. Topic-based event programming is a generalization of previous styles. Event-driven applications written in the topic-based style allow components (for example, classes, objects, and functions) to both emit events and/or register to event types, completely ignoring the other side of the emitter-handler relation.

In other words, handlers can be registered to event types, even if there is no emitter that would emit them, and emitters can emit events even if there is no one subscribed to receive them. In this style of event-driven programming, events are first-class entities that are often defined separately from emitters and handlers. Such events are often given a dedicated class or are just global singleton instances of one generic Event class. This is why handlers can subscribe to events even if there is no object that would emit them.

Depending on the framework or library of choice, the abstraction that's used to encapsulate such observable event types/classes can be named differently. Popular terms are **channels**, **topics**, and **signals**. The term **signal** is particularly popular, and, because of that, this style of programming is sometimes called **signal-driven programming**. Signals can be found in such popular libraries and frameworks as Django (web framework), Flask (web microframework), SQLAlchemy (database ORM), and Scrapy (web crawling and scraping framework).

Amazingly, successful Python projects do not build their own signaling frameworks from scratch, but instead use an existing dedicated library. The most popular signaling library in Python seems to be blinker. It is characterized by extremely wide Python version compatibility (Python 2.4 or later, Python 3.0 or later, Jython 2.5 or later, or PyPy 1.6 or later) and has an extremely simple and concise API that allows it to be used in almost any project.

`blinker` is built on the concept of named signals. To create a new signal definition, you simply use the `signal(name)` constructor. Two separate calls to the `signal()` constructor with the same `name` value will return the same signal object. This allows you to easily refer to signals at any time. The following is an example of the `SelfWatch` class, which uses named signals to notify its instances every time a new sibling is created:

```python
import itertools

from blinker import signal


class SelfWatch:
    _new_id = itertools.count(1)

    def __init__(self):
        self._id = next(self._new_id)
        init_signal = signal("SelfWatch.init")
        init_signal.send(self)
        init_signal.connect(self.receiver)

    def receiver(self, sender):
        print(f"{self}: received event from {sender}")

    def __str__(self):
        return f"<{self.__class__.__name__}: {self._id}>"
```

Let's save above code in topic_based_events.py file. The following transcript of the interactive session shows how new instances of the `SelfWatch` class notify the siblings about their initialization:

```python
>>> from topic_based_events import SelfWatch
>>> selfwatch1 = SelfWatch()
>>> selfwatch2 = SelfWatch()
<SelfWatch: 1>: received event from <SelfWatch: 2>
>>> selfwatch3 = SelfWatch()
<SelfWatch: 2>: received event from <SelfWatch: 3>
<SelfWatch: 1>: received event from <SelfWatch: 3>
>>> selfwatch4 = SelfWatch()
<SelfWatch: 2>: received event from <SelfWatch: 4>
<SelfWatch: 3>: received event from <SelfWatch: 4>
<SelfWatch: 1>: received event from <SelfWatch: 4>
```

Other interesting features of the `blinker` library are as follows:

- **Anonymous signals:** Empty `signal()` calls always create a completely new anonymous signal. By storing the signal as a module variable or class attribute, you will avoid typos in string literals or accidental signal name collisions.

- **Subject-aware subscription:** The `signal.connect()` method allows us to select a specific sender; this allows you to use subject-based event dispatching on top of topic-based dispatching.

- **Signal decorators:** The `signal.connect()` method can be used as a decorator; this shortens code and makes event handling more evident in the code base.

- **Data in signals:** The `signal.send()` method accepts arbitrary keyword arguments that will be passed to the connected handler; this allows signals to be used as a message-passing mechanism.

One really interesting thing about the topic-based style of event-driven programming is that it does not enforce subject-dependent relations between components. Both sides of the relation can be event emitters and handlers to each other, depending on the situation. This way of event handling becomes just a communication mechanism. This makes topic-based event programming a good choice for the architectural pattern.

The loose coupling of software components allows for smaller incremental changes. Also, an application process that is loosely coupled internally through a system of events can be easily split into multiple services that communicate through message queues. This allows transforming event-driven applications into distributed event-driven architectures.

Let's take a look at event-driven architectures in the next section.

Event-driven architectures

From event-driven applications, there is only one minor step to event-driven architectures. Event-driven programming allows you to split your application into isolated components that communicate with each other only by exchanging events or signals. If you already did this, you should be also able to split your application into separate services that do the same, but transfer events to each other, either through some kind of **inter-process communication** (**IPC**) mechanism or over the network.

Event-driven architectures transfer the concept of event-driven programming to the level of inter-service communication. There are many good reasons for considering such architectures:

- **Scalability and utilization of resources:** If your workload can be split into many order-independent events, architectures that are event-driven allow the work to be easily distributed across many computing nodes (hosts). The amount of computing power can also be dynamically adjusted to the number of events being processed in the system at any given moment.

- **Loose coupling:** Systems that are composed of many (preferably small) services communicating over queues tend to be more loosely coupled than monolithic systems. Loose coupling allows for easier incremental changes and the steady evolution of system architecture.

- **Failure resiliency:** Event-driven systems with proper event transport technology (distributed message queues with built-in message persistency) tend to be more resilient to transient issues. Modern message queues, such as Kafka or RabbitMQ, offer multiple ways to ensure that the message will always be delivered to at least one recipient and are able to ensure that the message will be redelivered in the case of unexpected errors.

Event-driven architectures work best for problems that can be dealt with asynchronously, such as file processing or file/email delivery, or for systems that deal with regular and/or scheduled events (for example, **cron jobs**). In Python, it can also be used as a way of overcoming the CPython interpreter's performance limitations (such as **Global Interpreter Lock** (**GIL**), which was discussed in *Chapter 6*, *Concurrency*) by splitting the workload across multiple independent processes.

Last but not least, event-driven architectures seem to have a natural affinity for serverless computing. In this cloud-computing execution model, you're not concerned about infrastructure and you don't have to purchase computing capacity units. You leave all of the scaling and infrastructure management for your cloud service operator and provide them only with your code to run. Often, the pricing for such services is based only on the resources that are used by your code. The most prominent category of serverless computing services is **Function as a Service** (**FaaS**), which executes small units of code (functions) in response to events.

In the next section, we will discuss in more detail event and message queues, which form the foundation of most event-based architectures.

Event and message queues

In most single-process implementations of event-driven programming, events are handled as soon as they appear and are usually processed in a serial fashion. Whether it is a callback-based style of GUI application or full-fledged signaling in the style of the `blinker` library, an event-driven application usually maintains some kind of mapping between events and lists of handlers to execute.

This style of information passing in distributed applications is usually realized through a request-response communication model. Request-response is a bidirectional and obviously synchronous way of communication between services. It can definitely be a basis for simple event handling but has many downsides that make it really inefficient in large-scale or complex systems. The biggest problem with request-response communication is that it introduces relatively high coupling between components:

- Every communicating component needs to be able to locate dependent services. In other words, event emitters need to know the network addresses of network handlers.

- A subscription happens directly in the service that emits the event. This means that, in order to create a completely new event connection, usually more than one service has to be modified.

- Both sides of communication must agree on the communication protocol and message format. This makes potential changes more complex.

- A service that emits events must handle potential errors that are returned in responses from dependent services.

- Request-response communication often cannot be easily handled in an asynchronous way. This means that event-based architecture built on top of a request-response communication system rarely benefits from concurrent processing flows.

Due to the preceding reasons, event-driven architectures are usually implemented using the concept of message queues, rather than request-response cycles. A message queue is a communication mechanism in the form of a dedicated service or library that is only concerned with the messages and their intended delivery mechanism. It just acts as a communication hub between various parties. In a contract, the request-response flow requires both communicating parties to know each other and be "alive" during every information exchange.

Typically, writing a new message to the message queue is a fast operation as it does not require immediate action (a callback) to be executed on the subscriber's side. Moreover, event emitters don't need their subscribers to be running at the time that the new message is emitted, and asynchronous messaging can increase failure resilience. The request-response flow, in contrast, assumes that dependent services are always available, and the synchronous processing of events can introduce large processing delays.

Message queues allow for the loose coupling of services because they isolate event emitters and handlers from each other. Event emitters publish messages directly to the queue but don't need to care if any other service listens to its events. Similarly, event handlers consume events directly from the queue and don't need to worry about who produced the events (sometimes, information about the event emitter is important, but, in such situations, it is either in the contents of the delivered message or takes part in the message routing mechanism). In such a communication flow, there is never a direct synchronous connection between event emitters and event handlers, and all the exchange of information happens through the queue.

In some circumstances, this decoupling can be taken to such an extreme that a single service can communicate with itself by an external queuing mechanism. This isn't so surprising, because using message queues is already a great way of inter-thread communication that allows you to avoid locking (see *Chapter 6*, *Concurrency*).

Besides loose coupling, message queues (especially in the form of dedicated services) have many additional capabilities:

- **Persistence**: Most message queues are able to provide message persistence. This means that, even if a message queue's service dies, no messages will be lost.

- **Retrying**: Many message queues support message delivery/processing confirmations and allow you to define a retry mechanism for messages that fail to deliver. This, with the support of message persistency, guarantees that if a message was successfully submitted, it will eventually be processed, even in the case of transient network or service failures.

- **Natural concurrency:** Message queues are naturally concurrent. With various message distribution semantics (for example, fan-out and round-robin), it is a great basis for a highly scalable and distributed architecture.

When it comes to the actual implementation of the message queue, we can distinguish two major architectures:

- **Brokered message queues:** In this architecture, there is one service (or cluster of services) that is responsible for accepting and distributing events. The most common examples of open-source brokered message queue systems are RabbitMQ and Apache Kafka. A popular cloud-based service is Amazon SQS. These types of systems are most capable in terms of message persistence and built-in message delivery semantics.

- **Brokerless message queues:** These are implemented solely as programming libraries. A popular brokerless messaging library is **ZeroMQ** (often spelled as **ØMQ** or **zmq**). The biggest advantage of brokerless messaging is elasticity. Brokerless messaging libraries trade operational simplicity (no additional centralized service or cluster of services to maintain) for feature completeness and complexity (things like persistence and complex message delivery need to be implemented inside of services).

Both types of messaging approach have advantages and disadvantages. In brokered message queues, there is always an additional service to maintain in the case of open-source queues running on their own infrastructure, or an additional entry on your cloud provider invoice in the case of cloud-based services. Such messaging systems quickly become a critical part of your architecture. If not designed with high availability in mind, such a central message queue can become **a single point of failure** for your whole system architecture. Anyway, modern queue systems have a lot of features available out of the box, and integrating them into your code is usually a matter of proper configuration or a few API calls. With the AMQP standard, it's also quite easy to run local ad hoc queues for testing.

With brokerless messaging, your communication is often more distributed. This means that your system architecture does not depend on a single messaging service or cluster. Even if some services are dead, the rest of the system can still communicate. The downside of this approach is that you're usually on your own when it comes to things like message persistency, delivery/processing confirmations, delivery retries, and handling complex network failure scenarios like network splits. If you have such needs, you will either have to implement such capabilities directly in your services or build your own messaging broker from scratch using brokerless messaging libraries. For larger distributed applications, it is usually better to use proven and battle-tested message brokers.

Event-driven architectures encourage modularity and the decomposition of large applications into smaller services. This has both advantages and disadvantages. With many components communicating over the queues, it may be harder to debug applications and understand how they work.

On the other hand, good system architecture practices like separation of concerns, domain isolation, and the use of formal communication contracts improve overall architecture and make the development of separate components easier.

> Examples of standards focused on creating formal communication contracts include OpenAPI and AsyncAPI. These are YAML-based specification languages for defining specifications for application communication protocols and schemas. You can learn more about them at `https://swagger.io/specification/` and `https://www.asyncapi.com`.

Summary

In this chapter, we discussed the elements of event-driven programming. We started with the most common examples and applications of event-driven programming to better introduce ourselves to this programming paradigm. Then, we precisely described the three main styles of event-driven programming, **callback-based style**, **subject-based style**, and **topic-based style**. There are many event-driven design patterns and programming techniques, but all of them fall into one of these three categories. The last part of this chapter focused on event-driven programming architectures.

With this chapter, we end something that we could call an "architecture and design arc." From now on, we will be talking less about architecture, design patterns, programming, and paradigms and more about Python internals and advanced syntax features.

The next chapter is all about metaprogramming in Python, that is, how to write programs that can treat themselves as data, analyze themselves, and modify themselves at runtime.

8
Elements of Metaprogramming

Metaprogramming is a collection of programming techniques that focus on the ability of programs to introspect themselves, understand their own code, and modify themselves. Such an approach to programming gives programmers a lot of power and flexibility. Without metaprogramming techniques, we probably wouldn't have modern programming frameworks, or at least those frameworks would be way less expressive.

The term **metaprogramming** is often shrouded in an aura of mystery. Many programmers associate it almost exclusively with programs that can inspect and manipulate their own code at the source level. Programs manipulating their own source code are definitely one of the most striking and complex examples of applied metaprogramming, but metaprogramming takes many forms and doesn't always have to be complex nor hard. Python is especially rich in features and modules that make certain metaprogramming techniques simple and natural.

In this chapter, we will explain what metaprogramming really is and present a few practical approaches to metaprogramming in Python. We will start with simple metaprogramming techniques like function and class decorators but will also cover advanced techniques to override the class instance creation process and the use of metaclasses. We will finish with examples of the most powerful but also dangerous approach to metaprogramming, which is code generation patterns.

In this chapter, we will cover the following topics:

- What is metaprogramming?
- Using decorators to modify function behavior before use
- Intercepting the class instance creation process
- Metaclasses
- Code generation

Before we get into some metaprogramming techniques available for Python developers, let's begin by considering the technical requirements.

Technical requirements

The following are Python packages that are mentioned in this chapter that you can download from PyPI:

- `inflection`
- `macropy3`
- `falcon`
- `hy`

Information on how to install packages is included in *Chapter 2, Modern Python Development Environments*.

The code files for this chapter can be found at `https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition/tree/main/Chapter%208`.

What is metaprogramming?

Maybe we could find a good academic definition of metaprogramming, but this is a book that is more about good software craftsmanship than about computer science theory. This is why we will use our own informal definition of metaprogramming:

> *"Metaprogramming is a technique of writing computer programs that can treat themselves as data, so they can introspect, generate, and/or modify themselves while running."*

Using this definition, we can distinguish between two major branches of metaprogramming in Python:

- **Introspection-oriented metaprogramming**: Focused on natural introspection capabilities of the language and dynamic definitions of functions and types
- **Code-oriented metaprogramming**: Metaprogramming focused on treating code as mutable data structures

Introspection-oriented metaprogramming concentrates on the language's ability to introspect its basic elements, such as functions, classes, or types, and to create or modify them on the go. Python really provides a lot of tools in this area. This feature of the Python language is often used by **Integrated Development Environments (IDEs)** to provide real-time code analysis and name suggestions. The easiest possible metaprogramming tools in Python that utilize language introspection features are decorators that allow for adding extra functionality to existing functions, methods, or classes. Next are special methods of classes that allow you to interfere with the class instance creation process. The most powerful are metaclasses, which allow programmers to even completely redesign Python's implementation of object-oriented programming.

Code-oriented metaprogramming allows programmers to work directly with code, either in its raw (plain text) format or in the more programmatically accessible **abstract syntax tree** (**AST**) form. This second approach is, of course, more complicated and difficult to work with but allows for really extraordinary things, such as extending Python's language syntax or even creating your own **domain-specific language** (**DSL**).

In the next section, we'll discuss what decorators are in the context of metaprogramming.

Using decorators to modify function behavior before use

Decorators are one of the most common inspection-oriented metaprogramming techniques in Python. Because functions in Python are first-class objects, they can be inspected and modified at runtime. Decorators are special functions capable of inspecting, modifying, or wrapping other functions.

The decorator syntax was explained in *Chapter 4*, *Python in Comparison with Other Languages*, and is in fact a syntactic sugar that is supposed to make it easier to work with functions that extend existing code objects with additional behavior.

You can write code that uses the simple decorator syntax as follows:

```
@some_decorator
def decorated_function():
    pass
```

You can also write it in the following (more verbose) way:

```
def decorated_function():
    pass
decorated_function = some_decorator(decorated_function)
```

This verbose form of function decoration clearly shows what the decorator does. It takes a function object and modifies it at runtime. A decorator usually returns a new function object that replaces the pre-existing decorated function name.

We've already seen how function decorators are indispensable in implementing many design patterns, in *Chapter 5, Interfaces, Patterns, and Modularity.* Function decorators are often used to intercept and preprocess original function arguments, modify the return values, or enhance the function call context with additional functional aspects like logging, profiling, or evaluating a caller's authorization/ authentication claims.

Let's, for instance, consider the following example usage of the `@lru_cache` decorator from the `functools` module:

```
from functools import lru_cache

@lru_cache(size=100)
def expensive(*args, **kwargs):
    ...
```

The `@lru_cache` decorator creates a **Last Recently Used** (**LRU**) cache of return values for a given function. It intercepts incoming function arguments and compares them with a list of recently used argument sets. If there is a match, it returns the cached value instead of calling the decorated function. If there is no match, the original function will be called first and the return value will be stored in the cache for later use. In our example, the cache will hold no more than 100 values.

What is really interesting is that the use of `@lru_cache` is already a metaprogramming technique. It takes an existing code object (here, the `expensive()` function) and modifies its behavior. It also intercepts arguments and inspects their value and type to decide whether these can be cached or not.

This is good news. We've seen already, in *Chapter 4*, *Python in Comparison with Other Languages*, that decorators are relatively easy to write and use in Python. In most cases, decorators make code shorter, easier to read, and also cheaper to maintain. This means that they serve as a perfect introductory technique to metaprogramming. Other metaprogramming tools that are available in Python may be more difficult to understand and master.

The natural step forward from function decorators is class decorators. We take a look at them in the next section.

One step deeper: class decorators

One of the lesser-known syntax features of Python is class decorators. Their syntax and implementation are exactly the same as function decorators. The only difference is that they are expected to return a class instead of the function object.

We've already used some class decorators in previous chapters. These were the `@dataclass` decorator from the `dataclasses` module explained in *Chapter 4*, *Python in Comparison with Other Languages*, and `@runtime_checkable` from the `typing` module, explained in *Chapter 5*, *Interfaces, Patterns, and Modularity*. Both decorators rely on Python's introspection capabilities to enhance existing classes with extra behavior:

- The `@dataclass` decorator inspects class attribute annotations to create a default implementation of the `__init__()` method and comparison protocol that saves developers from writing repeatable boilerplate code. It also allows you to create custom "frozen" classes with immutable and hashable instances that can be used as dictionary keys.

- The `@runtime_checkable` decorator marks `Protocol` subclasses as "runtime checkable." It means that the argument and return value annotation of the `Protocol` subclass can be used to determine at runtime if another class implements an interface defined by the protocol class.

The best way to understand how class decorators work is to learn by doing. The `@dataclass` and `@runtime_checkable` decorators have rather complex inner workings, so instead of looking at their actual code, we will try to build our own simple example.

One of the great features of dataclasses is the ability to provide a default implementation of the `__repr__()` method. That method returns a string representation of the object that can be displayed in an interactive session, logs, or in standard output.

For custom classes, this __repr__() method will by default include only the class name and memory address, but for dataclasses it automatically includes a representation of each individual field of the dataclass. We will try to build a dataclass decorator that will provide a similar capability for any class.

We'll start by writing a function that can return a human-readable representation of any class instance if given a list of attributes to represent:

```python
from typing import Any, Iterable

UNSET = object()

def repr_instance(instance: object, attrs: Iterable[str]) -> str:
    attr_values: dict[str, Any] = {
        attr: getattr(instance, attr, UNSET)
        for attr in attrs
    }
    sub_repr = ", ".join(
        f"{attr}={repr(val) if val is not UNSET else 'UNSET'}"
        for attr, val in attr_values.items()
    )
    return f"<{instance.__class__.__qualname__}: {sub_repr}>"
```

Our repr_instance() function starts by traversing instance attributes using the getattr() function over all attribute names provided in the attrs argument. Some instance attributes may not be set at the time we are creating the representation. The getattr() function will return None if the attribute is not set—however, None is also a valid attribute value so we need to have a way to distinguish unset attributes from None values. That's why we use the UNSET sentinel values.

> UNSET = object() is a common pattern for creating a unique sentinel value. The bare object type instance returns True on the is operator check only when compared with itself.

Once attributes and their values are known, our function uses f-strings to create an actual representation of the class instance that will include a representation of each individual attribute defined in the attrs argument.

We will soon look at how to automatically include such representations in custom classes, but first, let's try to see how it deals with existing objects. Here, for instance, is an example of using the `instance_repr()` function in an interactive session to get a representation of an imaginary number:

```
>>> repr_instance(1+10j, ["real", "imag"])
'<complex: real=1.0, imag=10.0>'
```

It's good so far but we need to pass the object instance explicitly and know all the possible attribute names before we want to print them. That's not very convenient because we will have to update the arguments of `repr_instance()` every time the structure of the class changes. We will write a class decorator that will be able to take the `repr_instance()` function and inject it into a decorated class. We will also use class attribute annotations stored under a class's `__annotations__` attribute to determine what attributes we want to include in the representation. Following is the code of our decorator:

```python
def autorepr(cls):
    attrs = set.union(
        *(
            set(c.__annotations__.keys())
            for c in cls.mro()
            if hasattr(c, "__annotations__")
        )
    )

    def __repr__(self):
        return repr_instance(self, sorted(attrs))

    cls.__repr__ = __repr__
    return cls
```

In those few lines, we use a lot of things that we learned about in *Chapter 4, Python in Comparison with Other Languages*. We start by obtaining a list of annotated attributes from the `cls.__annotations__` dictionary from each class in the class **Method Resolution Order** (**MRO**). We have to traverse the whole MRO because annotations are not inherited from base classes.

Later, we use a closure to define an inner `__repr__()` function that has access to the `attrs` variable from the outer scope. When that's done, we override the existing `cls.__repr__()` method with a new implementation. We can do that because function objects are already non-data descriptors. It means that in the class context they become methods and simply receive an instance object as a first argument.

Now we can test our decorator on some custom instance. Let's save our code in the `autorepr.py` file and define some trivial class with attribute annotations that will be decorated with our `@autorepr` decorator:

```python
from typing import Any

@autorepr
class MyClass:
    attr_a: Any
    attr_b: Any
    attr_c: Any

    def __init__(self, a, b):
        self.attr_a = a
        self.attr_b = b
```

If you are vigilant, you've probably noticed that we have missed the `attr_c` attribute initialization. This is intentional. It will allow us to see how `@autorepr` deals with unset attributes. Let's start Python, import our class, and see the automatically generated representations:

```python
>>> from autorepr import MyClass
>>> MyClass("Ultimate answer", 42)
<MyClass: attr_a='Ultimate answer', attr_b=42, attr_c=UNSET>
>>> MyClass([1, 2, 3], ["a", "b", "c"])
<MyClass: attr_a=[1, 2, 3], attr_b=['a', 'b', 'c'], attr_c=UNSET>
>>> instance = MyClass(None, None)
>>> instance.attr_c = None
>>> instance
<MyClass: attr_a=None, attr_b=None, attr_c=None>
```

The above example from an interactive Python session shows how the `@autorepr` decorator can use class attribute annotations to discover the fields that need to be included in instance representation. It is also able to distinguish unset attributes from those that have an explicit `None` value. A decorator is reusable so you can easily apply it to any class that has type annotations for attributes instead of creating new `__repr__()` methods manually.

Moreover, it does not require constant maintenance. If you extend the class with an additional attribute annotation, it will be automatically included in instance representation.

Modifying existing classes in place (also known as **monkey patching**) is a common technique used in class decorators. The other way to enhance existing classes with decorators is through utilizing closures to create new subclasses on the fly. If we had to rewrite our example as a subclassing pattern, we could write it as follows:

```python
def autorepr(cls):
    attrs = cls.__annotations__.keys()

    class Klass(cls):
        def __repr__(self):
            return repr_instance(self, attrs)

    return Klass
```

The major drawback of using closures in class decorators this way is that this technique affects class hierarchy. Among others, this will override the class's __name__, __qualname__, and __doc__ attributes. In our case, that would also mean that part of the intended functionality would be lost. The following are would-be example representations of MyClass decorated with such a decorator:

```
<autorepr.<locals>.Klass: attr_a='Ultimate answer', attr_b=42, attr_
c=UNSET>
<autorepr.<locals>.Klass: attr_a=[1, 2, 3], attr_b=['a', 'b', 'c'],
attr_c=UNSET>
```

This cannot be easily fixed. The functools module provides the @wraps utility decorator, which can be used in ordinary function decorators to preserve the metadata of an annotated function. Unfortunately, it can't be used with class decorators. This makes the use of subclassing in class decorators limited. They can, for instance, break the results of automated documentation generation tools.

Still, despite this single caveat, class decorators are a simple and lightweight alternative to the popular **mixin** class pattern. A mixin in Python is a class that is not meant to be instantiated but is instead used to provide some reusable API or functionality to other existing classes. Mixin classes are almost always added using multiple inheritance. Their usage usually takes the following form:

```python
class SomeConcreteClass(MixinClass, SomeBaseClass):
    pass
```

Mixin classes form a useful design pattern that is utilized in many libraries and frameworks. To name one, Django is an example framework that uses them extensively. While useful and popular, mixin classes can cause some trouble if not designed well, because, in most cases, they require the developer to rely on multiple inheritance. As we stated earlier, Python handles multiple inheritance relatively well, thanks to its clear MRO implementation. Anyway, try to avoid subclassing multiple classes if you can, as multiple inheritance makes code complex and hard to work with. This is why class decorators may be a good replacement for mixin classes.

In general, decorators concentrate on modifying the behavior of functions and classes before they are actually used. Function decorators replace existing functions with their wrapped alternatives and class decorators, usually modifying the class definition. But there are some metaprogramming techniques that concentrate more on modifying code behavior when it is actually in use. One of those techniques relies on intercepting the class instance creation process through the overriding of the `__new__()` method. We will discuss this in the next section.

Intercepting the class instance creation process

There are two special methods concerned with the class instance creation and initialization process. These are `__init__()` and `__new__()`.

The `__init__()` method is the closest to the concept of the constructor found in many object-oriented programming languages. It receives a fresh class instance together with initialization arguments and is responsible for initializing the class instance state.

The special method `__new__()` is a static method that is actually responsible for creating class instances. This `__new__(cls, [,...])` method is called prior to the `__init__()` initialization method. Typically, the implementation of the overridden `__new__()` method invokes its superclass version using `super().__new__()` with suitable arguments and modifies the instance before returning it.

> The `__new__()` method is a special-cased static method so there is no need to declare it as a static method using the `staticmethod` decorator.

The following is an example class with the overridden __new__() method implementation in order to count the number of class instances:

```python
class InstanceCountingClass:
    created = 0
    number: int

    def __new__(cls, *args, **kwargs):
        instance = super().__new__(cls)
        instance.number = cls.created
        cls.created += 1

        return instance

    def __repr__(self):
        return (
            f"<{self.__class__.__name__}: "
            f"{self.number} of {self.created}>"
        )
```

Here is the log of the example interactive session that shows how our InstanceCountingClass implementation works:

```python
>>> instances = [InstanceCountingClass() for _ in range(5)]
>>> for i in instances:
...     print(i)
...
<InstanceCountingClass: 0 of 5>
<InstanceCountingClass: 1 of 5>
<InstanceCountingClass: 2 of 5>
<InstanceCountingClass: 3 of 5>
<InstanceCountingClass: 4 of 5>
>>> InstanceCountingClass.created
5
```

The __new__() method should usually return the instance of the featured class, but it is also possible for it to return other class instances. If this does happen (a different class instance is returned), then the call to the __init__() method is skipped. This fact is useful when there is a need to modify the creation/initialization behavior of immutable class instances like some of Python's built-in types.

Following is an example of the subclassed `int` type that does not include a zero value:

```python
class NonZero(int):
    def __new__(cls, value):
        return super().__new__(cls, value) if value != 0 else None

    def __init__(self, skipped_value):
        # implementation of __init__ could be skipped in this case
        # but it is left to present how it may be not called
        print("__init__() called")
        super().__init__()
```

The above example includes several `print` statements to present how Python skips the `__init__()` method call in certain situations. Let's review these in the following interactive session:

```python
>>> type(NonZero(-12))
__init__() called
<class '__main__.NonZero'>
>>> type(NonZero(0))
<class 'NoneType'>
>>> NonZero(-3.123)
__init__() called
-3
```

So, when should we use `__new__()`? The answer is simple: only when `__init__()` is not enough. One such case was already mentioned, that is, subclassing immutable built-in Python types such as `int`, `str`, `float`, `frozenset`, and so on. This is because there was no way to modify such an immutable object instance in the `__init__()` method once it was created.

Some programmers would argue that `__new__()` may be useful for performing important object initialization that may be missed if the user forgets to use the `super().__init__()` call in the overridden initialization method. While it sounds reasonable, this has a major drawback. With such an approach, it becomes harder for the programmer to explicitly skip previous initialization steps if this is the already desired behavior. It also breaks an unspoken rule of all initializations performed in `__init__()`.

Because `__new__()` is not constrained to return the same class instance, it can be easily abused. Irresponsible usage of this method might do a lot of harm to code readability, so it should always be used carefully and backed with extensive documentation. Generally, it is better to search for other solutions that may be available for the given problem, instead of affecting object creation in a way that will break a basic programmer's expectations. Even the overridden initialization of immutable types can be replaced with more predictable and well-established design patterns like factory methods.

> Factory methods in Python are usually defined with the use of the `classmethod` decorator, which can intercept arguments before the class constructor is invoked. That usually allows you to pack more than one initialization semantic into a single class. Following is an example of a `list` type subclass that has two factory methods for creating list instances that are doubled or tripled in size:
>
> ```python
> from collections import UserList
>
>
> class XList(UserList):
>     @classmethod
>     def double(cls, iterable):
>         return cls(iterable) * 2
>
>     @classmethod
>     def triple(cls, iterable):
>         return cls(iterable) * 3
> ```

There is at least one aspect of Python programming where extensive usage of the `__new__()` method is well justified. These are metaclasses, which are described in the next section.

Metaclasses

A metaclass is a Python feature that is considered by many as one of the most difficult things to understand in the language and is thus avoided by a great number of developers. In reality, it is not as complicated as it sounds once you understand a few basic concepts. As a reward, knowing how to use metaclasses grants you the ability to do things that are not possible without them.

A metaclass is a type (class) that defines other types (classes). The most important thing to know in order to understand how they work is that classes (so, types that define object structure and behavior) are objects too. So, if they are objects, then they have an associated class. The basic type of every class definition is simply the built-in `type` class (see *Figure 8.1*).
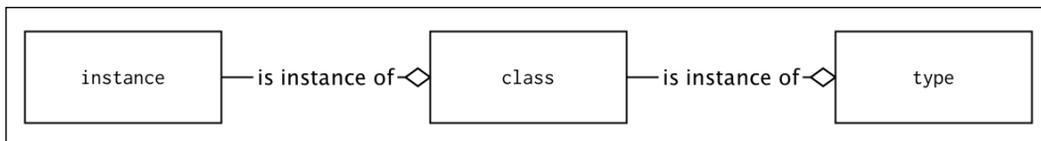
Figure 8.1: How classes are typed

In Python, it is possible to substitute the metaclass for a class object with your own type. Usually, the new metaclass is still the subclass of the `type` metaclass (refer to *Figure 8.2*) because not doing so would make the resulting classes highly incompatible with other classes in terms of inheritance:
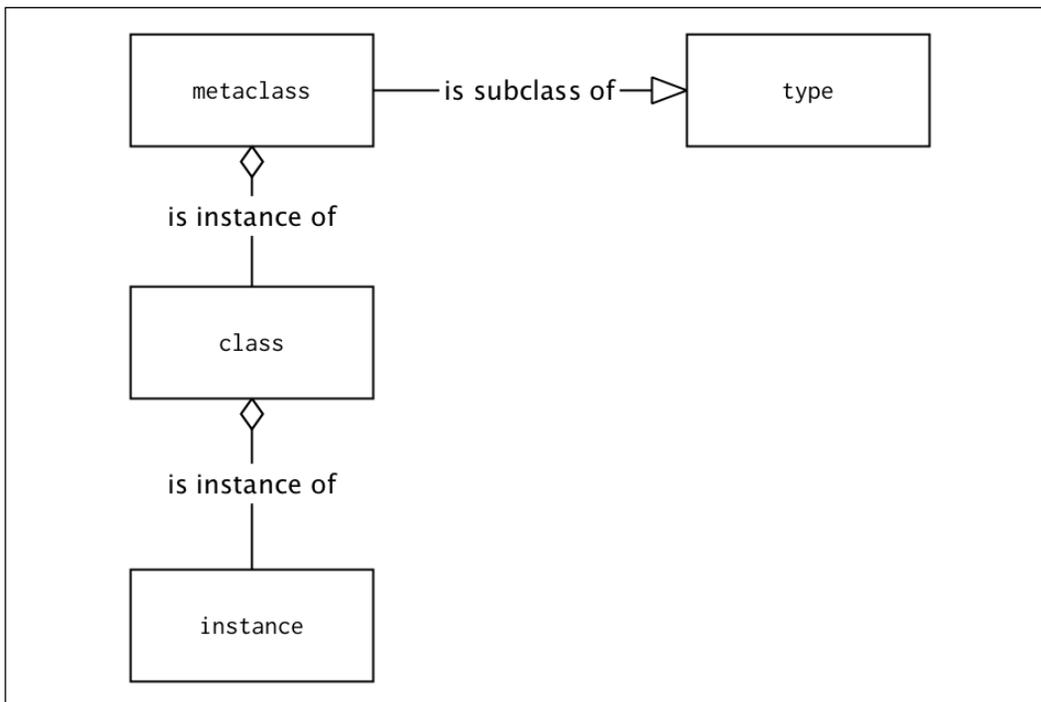
Figure 8.2: Usual implementation of custom metaclasses

Let's take a look at the general syntaxes for metaclasses in the next section.

The general syntax

The call to the built-in `type()` class can be used as a dynamic equivalent of the `class` statement. The following is an example of a class definition with the `type()` call:

```python
def method(self):
    return 1


MyClass = type('MyClass', (), {'method': method})
```

The first argument is the class name, the second is a list of base classes (here, an empty tuple), and the third is a dictionary of class attributes (usually methods). This is equivalent to the explicit definition of the class with the `class` keyword:

```python
class MyClass:
    def method(self):
        return 1
```

Every class that's created with the `class` statement implicitly uses `type` as its metaclass. This default behavior can be changed by providing the `metaclass` keyword argument to the `class` statement, as follows:

```python
class ClassWithAMetaclass(metaclass=type):
    pass
```

The value that's provided as a `metaclass` argument is usually another class object, but it can be any other callable that accepts the same arguments as the `type` class and is expected to return another class object.

The detailed call signature of a metaclass is `type(name, bases, namespace)` and the meaning of the arguments are as follows:

- `name`: This is the name of the class that will be stored in the `__name__` attribute
- `bases`: This is the list of parent classes that will become the `__bases__` attribute and will be used to construct the MRO of a newly created class
- `namespace`: This is a namespace (mapping) with definitions for the class body that will become the `__dict__` attribute

One way of thinking about metaclasses is the `__new__()` method but at a higher level of class definition.

Despite the fact that functions that explicitly call `type()` can be used in place of metaclasses, the usual approach is to use a different class that inherits from `type` for this purpose. The common template for a metaclass is as follows:

```python
class Metaclass(type):
    def __new__(mcs, name, bases, namespace):
        return super().__new__(mcs, name, bases, namespace)

    @classmethod
    def __prepare__(mcs, name, bases, **kwargs):
        return super().__prepare__(name, bases, **kwargs)

    def __init__(cls, name, bases, namespace, **kwargs):
        super().__init__(name, bases, namespace)

    def __call__(cls, *args, **kwargs):
        return super().__call__(*args, **kwargs)
```

The `name`, `bases`, and `namespace` arguments have the same meaning as in the `type()` call we explained earlier, but each of these four methods is invoked at a different stage of the class lifecycle:

- `__new__(mcs, name, bases, namespace)`: This is responsible for the actual creation of the class object in the same way as it does for ordinary classes. The first positional argument is a metaclass object. In the preceding example, it would simply be `Metaclass`. Note that `mcs` is the popular naming convention for this argument.

- `__prepare__(mcs, name, bases, **kwargs)`: This creates an empty namespace object. By default, it returns an empty `dict` instance, but it can be overridden to return any other `dict` subclass instance. Note that it does not accept `namespace` as an argument because, before calling it, the namespace does not exist yet. Example usage of that method will be explained later, in the *Metaclass usage* section.

- `__init__(cls, name, bases, namespace, **kwargs)`: This is not common in metaclass implementations but has the same meaning as in ordinary classes. It can perform additional class object initialization once it is created with `__new__()`. The first positional argument is now named `cls` by convention to mark that this is already a created class object (metaclass instance) and not a metaclass object. When `__init__()` is called, the class has been already constructed and so the `__init__()` method can do less than the `__new__()` method. Implementing such a method is very similar to using class decorators, but the main difference is that `__init__()` will be called for every subclass, while class decorators are not called for subclasses.

- `__call__(cls, *args, **kwargs)`: This is called when an instance of a metaclass is called. The instance of a metaclass is a class object (refer to *Figure 8.1*); it is invoked when you create new instances of a class. This can be used to override the default way of how class instances are created and initialized.

Each of the preceding methods can accept additional extra keyword arguments, all of which are represented by `**kwargs`. These arguments can be passed to the `metaclass` object using extra keyword arguments in the class definition in the form of the following code:

```python
class Klass(metaclass=Metaclass, extra="value"):
    pass
```

This amount of information can be overwhelming at first without proper examples, so let's trace the creation of metaclasses, classes, and instances with some `print()` calls:

```python
class RevealingMeta(type):
    def __new__(mcs, name, bases, namespace, **kwargs):
        print(mcs, "METACLASS __new__ called")
        return super().__new__(mcs, name, bases, namespace)

    @classmethod
    def __prepare__(mcs, name, bases, **kwargs):
        print(mcs, " METACLASS __prepare__ called")
        return super().__prepare__(name, bases, **kwargs)

    def __init__(cls, name, bases, namespace, **kwargs):
        print(cls, " METACLASS __init__ called")
        super().__init__(name, bases, namespace)

    def __call__(cls, *args, **kwargs):
        print(cls, " METACLASS __call__ called")
        return super().__call__(*args, **kwargs)
```

Using `RevealingMeta` as a metaclass to create a new class definition will give the following output in the Python interactive session:

```python
>>> class RevealingClass(metaclass=RevealingMeta):
...     def __new__(cls):
...         print(cls, "__new__ called")
...         return super().__new__(cls)
...     def __init__(self):
...         print(self, "__init__ called")
```

```
...            super().__init__()
...
<class '__main__.RevealingMeta'> METACLASS __prepare__ called
<class '__main__.RevealingMeta'> METACLASS __new__ called
<class '__main__.RevealingClass'> METACLASS __init__ called
```

As you can see, during the class definition, only metaclass methods are called. The first one is the __prepare__() method, which prepares a new class namespace. It is immediately followed by the __new__() method, which is responsible for the actual class creation and receives the namespace created by the __prepare__() method. Last is the __init__() method, which receives the class object created by the __new__() method (here, the RevealingClass definition).

Metaclass methods cooperate with class methods during class instance creation. We can trace the order of method calls by creating a new RevealingClass instance in the Python interactive session:

```
>>> instance = RevealingClass()
<class '__main__.RevealingClass'> METACLASS __call__ called
<class '__main__.RevealingClass'> CLASS __new__ called
<__main__.RevealingClass object at 0x10f594748> CLASS __init__ called
```

The first method called was the __call__() method of a metaclass. At this point, it has access to the class object (here, the RevealingClass definition) but no class instance has been created yet. It is called just before class instance creation, which should happen in the __new__() method of the class definition. The last step of the class instance creation process is the call to the class __init__() method responsible for instance initialization.

We know roughly how metaclasses work in theory so let's now take a look at example usage of metaclasses.

Metaclass usage

Metaclasses are a great tool for doing unusual things. They give a lot of flexibility and power in modifying typical class behavior. So, it is hard to tell what common examples of their usage are. It would be easier to say that most usages of metaclasses are pretty uncommon.

For instance, let's take a look at the __prepare__() method of every object type. It is responsible for preparing the namespace of class attributes. The default type for a class namespace is a plain dictionary. For years, the canonical example of the __prepare__() method was providing a collections.OrderedDict instance as a class namespace.

Preserving the order of attributes in the class namespace allowed for things like repeatable object representation and serialization. But since Python 3.7 dictionaries are guaranteed to preserve key insertion order, that use case is gone. But it doesn't mean that we can't play with namespaces.

Let's imagine the following problem: we have a large Python code base that was developed over dozens of years and the majority of the code was written way before anyone in the team cared about coding standards. We may have, for instance, classes mixing `camelCase` and `snake_case` as the method naming convention. If we cared about consistency, we would be forced to spend a tremendous amount of effort to refactor the whole code base into either of the naming conventions. Or we could just use some clever metaclass that could be added on top of existing classes that would allow for calling methods in both ways. We could write new code using the new calling convention (preferably `snake_case`) while leaving the old code untouched and waiting for a gradual update.

That's an example of a situation where the `__prepare__()` method could be used! Let's start by writing a `dict` subclass that automatically interpolates `camelCase` names into `snake_case` keys:

```python
from typing import Any
import inflection


class CaseInterpolationDict(dict):
    def __setitem__(self, key: str, value: Any):
        super().__setitem__(key, value)
        super().__setitem__(inflection.underscore(key), value)
```

> To save some work, we use the `inflection` module, which is not a part of the standard library. It is able to convert strings between various "string cases." You can download it from PyPI using `pip`:
>
> ```
> $ pip install inflection
> ```

Our `CaseInterpolationDict` class works almost like an ordinary `dict` type but whenever it stores a new value, it saves it under two keys: the original one and one converted to `snake_case`. Note that we used the `dict` type as a parent class instead of the recommended `collections.UserDict`. This is because we will use this class in the metaclass `__prepare__()` method and Python requires namespaces to be `dict` instances.

Now it's time to write an actual metaclass that will override the class namespace type. It will be surprisingly short:

```python
class CaseInterpolatedMeta(type):
    @classmethod
    def __prepare__(mcs, name, bases):
        return CaseInterpolationDict()
```

Since we are set up, we can now use the `CaseInterpolatedMeta` metaclass to create a dummy class with a few methods that uses the `camelCase` naming convention:

```python
class User(metaclass=CaseInterpolatedMeta):
    def __init__(self, firstName: str, lastName: str):
        self.firstName = firstName
        self.lastName = lastName

    def getDisplayName(self):
        return f"{self.firstName} {self.lastName}"

    def greetUser(self):
        return f"Hello {self.getDisplayName()}!"
```

Let's save all that code in the `case_user.py` file and start an interactive session to see how the `User` class behaves:

```python
>>> from case_user import User
```

The first important thing to notice is the contents of the `User.__dict__` attribute:

```python
>>> User.__dict__
mappingproxy({
    '__module__': 'case_class',
    '__init__': <function case_class.User.__init__(self, firstName:
str, lastName: str)>,
    'getDisplayName': <function case_class.User.getDisplayName(self)>,
    'get_display_name': <function case_class.User.
getDisplayName(self)>,
    'greetUser': <function case_class.User.greetUser(self)>,
    'greet_user': <function case_class.User.greetUser(self)>,
    '__dict__': <attribute '__dict__' of 'User' objects>,
    '__weakref__': <attribute '__weakref__' of 'User' objects>,
    '__doc__': None
})
```

The first thing that catches the eye is the fact that methods got duplicated. That was exactly what we wanted to achieve. The second important thing is the fact that `User.__dict__` is of the `mappingproxy` type. That's because Python always copies the contents of the namespace object to a new `dict` when creating the final class object. The mapping proxy also allows proxy access to superclasses within the class MRO.

So, let's see if our solution works by invoking all of its methods:

```
>>> user = User("John", "Doe")
>>> user.getDisplayName()
'John Doe'
>>> user.get_display_name()
'John Doe'
>>> user.greetUser()
'Hello John Doe!'
>>> user.greet_user()
'Hello John Doe!'
```

It works! We could call all the `snake_case` methods even though we haven't defined them. For an unaware developer, that could look almost like magic!

However, this is a kind of magic that should be used very carefully. Remember that what you have just seen is a toy example. The real purpose of it was to show what is possible with metaclasses and just a few lines of code. In fact, doing something similar in a large and complex code base could be really dangerous. Metaclasses interact with the very core of the Python data model and can lead to various pitfalls. Some of them are discussed in the next section.

Metaclass pitfalls

Metaclasses, once mastered, are a powerful feature, but always complicate the code. Metaclasses also do not compose well and you'll quickly run into problems if you try to mix multiple metaclasses through inheritance.

Like some other advanced Python features, metaclasses are very elastic and can be easily abused. While the call signature of the class is rather strict, Python does not enforce the type of the return parameter. It can be anything as long as it accepts incoming arguments on calls and has the required attributes whenever it is needed.

One such object that can be anything-anywhere is the instance of the `Mock` class that's provided in the `unittest.mock` module. `Mock` is not a metaclass and also does not inherit from the `type` class. It also does not return the class object on instantiating. Still, it can be included as a `metaclass` keyword argument in the class definition, and this will not raise any syntax errors.

Using `Mock` as a metaclass is, of course, complete nonsense, but let's consider the following example:

```python
>>> from unittest.mock import Mock
>>> class Nonsense(metaclass=Mock):  # pointless, but illustrative
...     pass
...
>>> Nonsense
<Mock spec='str' id='4327214664'>
```

It's not hard to predict that any attempt to instantiate our `Nonsense` pseudo-class will fail. What is really interesting is the following exception and traceback you'll get trying to do so:

```python
>>> Nonsense()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Library/Frameworks/Python.framework/Versions/3.9/lib/
python3.9/unittest/mock.py", line 917, in __call__
    return _mock_self._mock_call(*args, **kwargs)
  File "/Library/Frameworks/Python.framework/Versions/3.9/lib/
python3.9/unittest/mock.py", line 976, in _mock_call
    result = next(effect)
StopIteration
```

Does the `StopIteration` exception give you any clue that there may be a problem with our class definition on the metaclass level? Obviously not. This example illustrates how hard it may be to debug metaclass code if you don't know where to look for errors.

But there are situations where things cannot be easily done without metaclasses. For instance, it is hard to imagine Django's ORM implementation built without extensive use of metaclasses. It could be possible, but it is rather unlikely that the resulting solution would be similarly easy to use. Frameworks are the place where metaclasses really shine. They usually have a lot of complex internal code that is not easy to understand and follow but eventually allow other programmers to write more condensed and readable code that operates on a higher level of abstraction.

For simple things, like changing the read/write attributes or adding new ones, metaclasses can be avoided in favor of simpler solutions, such as properties, descriptors, or class decorators. There is also a special method named __init__
subclass__(), which can be used as an alternative to metaclasses in many situations.
Let's take a closer look at it in the next section.

Using the __init__subclass__() method as an alternative to metaclasses

The `@autorepr` decorator presented in the *One step deeper: class decorators* section was fairly simple and useful. Unfortunately, it has one problem that we haven't discussed yet: it doesn't play well with subclassing.

It will work well with simple one-off classes that do not have any descendants but once you start subclassing the originally decorated class, you will notice that it doesn't work as one might expect. Consider the following class inheritance:

```python
from typing import Any
from autorepr import autorepr

@autorepr
class MyClass:
    attr_a: Any
    attr_b: Any
    attr_c: Any

    def __init__(self, a, b):
        self.attr_a = a
        self.attr_b = b

class MyChildClass(MyClass):
    attr_d: Any

    def __init__(self, a, b):
        super().__init__(a, b)
```

If you try to obtain a representation of `MyChildClass` instances in an interactive interpreter session, you will see the following output:

```
<MyChildClass: attr_a='Ultimate answer', attr_b=42, attr_c=UNSET>
<MyChildClass: attr_a=[1, 2, 3], attr_b=['a', 'b', 'c'], attr_c=UNSET>
```

That's understandable. The `@autorepr` decorator was used only on the base class so didn't have access to the subclass annotations. `MyChildClass` inherited the unmodified `__repr__()` method.

The way to fix that is to add the `@autorepr` decorator to the subclass as well:

```python
@autorepr
class MyChildClass(MyClass):
    attr_d: Any

    def __init__(self, a, b):
        super().__init__(a, b)
```

But how can we make the class decorator auto-apply on subclasses? We could clearly replicate the same behavior with the use of metaclasses but we already know that this can really complicate things. That would also make usage way harder as you can't really mix the inheritance of classes using different metaclasses.

Fortunately, there's a method for that. Python classes provide the `__init_subclass__()` hook method that will be invoked for every subclass. It is a convenient alternative for otherwise problematic metaclasses. This hook just lets the base class know that it has subclasses. It is often used to facilitate various event-driven and signaling patterns (see *Chapter 7, Event-Driven Programming*) but can also be employed to create "inheritable" class decorators.

Consider the following modification to our `@autorepr` decorator:

```python
def autorepr(cls):
    attrs = set.union(
        *(
            set(c.__annotations__.keys())
            for c in cls.mro()
            if hasattr(c, "__annotations__")
        )
    )

    def __repr__(self):
        return repr_instance(self, sorted(attrs))
    cls.__repr__ = __repr__

    def __init_subclass__(cls):
        autorepr(cls)
    cls.__init_subclass__ = classmethod(__init_subclass__)

    return cls
```

What is new is the `__init_subclass__()` method, which will be invoked with the new class object every time the decorated class is subclassed. In that method, we simply re-apply the `@autorepr` decorator. It will have access to all new annotations and will also be able to hook itself in for further subclasses. That way you don't have to manually add the decorator for every new subclass and can be sure that all `__repr__()` methods will always have access to the latest annotations.

So far, we have discussed the built-in features of Python that facilitate the metaprogramming techniques. We've seen that Python is quite generous in this area thanks to natural introspection capabilities, metaclasses, and the flexible object model. But there's a branch of metaprogramming available to practically any language, and regardless of its features. It is code generation. We will discuss that in the next section.

Code generation

As we already mentioned, dynamic code generation is the most difficult approach to metaprogramming. There are tools in Python that allow you to generate and execute code or even make some modifications to already compiled code objects.

Various projects such as **Hy** (mentioned later) show that even whole languages can be reimplemented in Python using code generation techniques. This proves that the possibilities are practically limitless. Knowing how vast this topic is and how badly it is riddled with various pitfalls, I won't even try to give detailed suggestions on how to create code this way, or to provide useful code samples.

Anyway, knowing what is possible may be useful for you if you plan to study this field in more depth by yourself. So, treat this section only as a short summary of possible starting points for further learning.

Let's take a look at how to use the `exec()`, `eval()`, and `compile()` functions.

exec, eval, and compile

Python provides the following three built-in functions to manually execute, evaluate, and compile arbitrary Python code:

- `exec(object, globals, locals)`: This allows you to dynamically execute Python code. The `object` attribute should be a string or code object (see the `compile()` function) representing a single statement or a sequence of multiple statements. The `globals` and `locals` arguments provide global and local namespaces for the executed code and are optional.

If they are not provided, then the code is executed in the current scope. If provided, `globals` must be a dictionary, while `locals` may be any mapping object. The `exec()` function always returns `None`.

- `eval(expression, globals, locals)`: This is used to evaluate the given expression by returning its value. It is similar to `exec()`, but it expects the `expression` argument to be a single Python expression and not a sequence of statements. It returns the value of the evaluated expression.

- `compile(source, filename, mode)`: This compiles the source into the code object or AST object. The source code is provided as a string value in the `source` argument. The `filename` should be the name of the file from which the code was read. If it has no file associated (for example, because it was created dynamically), then `"<string>"` is the value that is commonly used. The `mode` argument should be either `"exec"` (a sequence of statements), `"eval"` (a single expression), or `"single"` (a single interactive statement, such as in a Python interactive session).

The `exec()` and `eval()` functions are the easiest to start with when trying to dynamically generate code because they can operate on strings. If you already know how to program in Python, then you may already know how to correctly generate working source code programmatically.

The most useful in the context of metaprogramming is obviously `exec()` because it allows you to execute any sequence of Python statements. The word *any* should be alarming for you. Even `eval()`, which allows only the evaluation of expressions in the hands of a skillful programmer (when fed with the user input), can lead to serious security holes.

> Note that crashing the Python interpreter is the scenario you should be least afraid of. Introducing vulnerability to remote execution exploits due to irresponsible use of `exec()` and `eval()` could cost you your image as a professional developer, or even your job. This means that neither `exec()` nor `eval()` should ever be used with untrusted input. And every input coming from end users should always be considered unsafe.

Even if used with trusted input, there is a list of little details about `exec()` and `eval()` that is too long to be included here but might affect how your application works in ways you would not expect. Armin Ronacher has a good article that lists the most important of them, titled *Be careful with exec and eval in Python* (refer to `http://lucumr.pocoo.org/2011/2/1/exec-in-python/`).

Despite all these frightening warnings, there are natural situations where the usage of exec() and eval() is really justified. Still, in the case of even the tiniest doubt, you should not use them and try to find a different solution.

> The signature of the eval() function might make you think that if you provide empty globals and locals namespaces and wrap them with proper try ... except statements, then it will be reasonably safe. There could be nothing more wrong. Ned Batchelder has written a very good article in which he shows how to cause an interpreter segmentation fault in the eval() call, even with erased access to all Python built-ins (see http://nedbatchelder.com/blog/201206/eval_really_is_dangerous.html). This should be enough proof that both exec() and eval() should never be used with untrusted input.

We'll take a look at the abstract syntax tree in the next section.

The abstract syntax tree

The Python syntax is converted into **AST** format before it is compiled into byte code. This is a tree representation of the abstract syntactic structure of the source code. Processing of Python grammar is available thanks to the built-in ast module. Raw ASTs of Python code can be created using the compile() function with the ast.PyCF_ONLY_AST flag, or by using the ast.parse() helper. Direct translation in reverse is not that simple and there is no function provided in the standard library that can do so. Some projects, such as PyPy, do such things though.

The ast module provides some helper functions that allow you to work with the AST, for example:

```
>>> import ast
>>> tree = ast.parse('def hello_world(): print("hello world!")')
>>> tree
<_ast.Module object at 0x00000000038E9588>
>>> print(ast.dump(tree, indent=4))
Module(
    body=[
        FunctionDef(
            name='hello_world',
            args=arguments(
                posonlyargs=[],
                args=[],
```

```
            kwonlyargs=[],
            kw_defaults=[],
            defaults=[]),
        body=[
            Expr(
                value=Call(
                    func=Name(id='print', ctx=Load()),
                    args=[
                        Constant(value='hello world!')],
                    keywords=[]))],
            decorator_list=[])],
    type_ignores=[])
```

It is important to know that the AST can be modified before being passed to `compile()`. This gives you many new possibilities. For instance, new syntax nodes can be used for additional instrumentation, such as test coverage measurement. It is also possible to modify the existing code tree in order to add new semantics to the existing syntax. Such a technique is used by the MacroPy project (`https://github.com/lihaoyi/macropy`) to add syntactic macros to Python using the already existing syntax (refer to *Figure 8.3*):
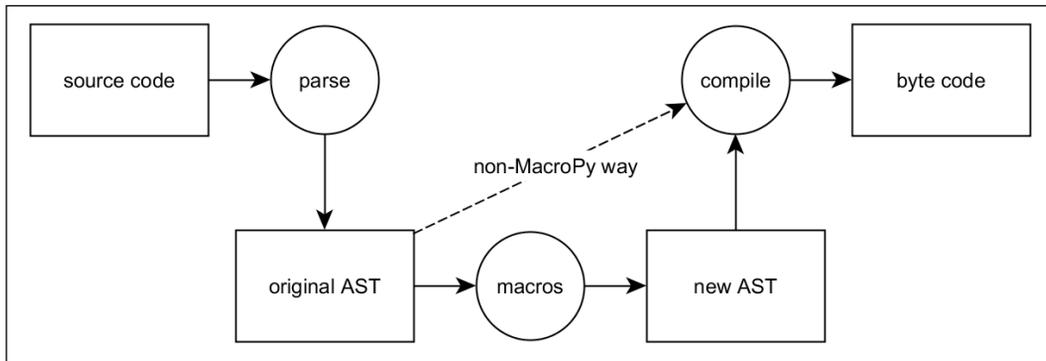
Figure 8.3: How MacroPy adds syntactic macros to Python modules on import

> Unfortunately, MacroPy isn't compatible with the latest Python versions and is only tested to run on Python 3.4. Anyway, it is a very interesting project that shows what can be achieved with AST manipulation.

ASTs can also be created in a purely artificial manner, and there is no need to parse any source code at all. This gives Python programmers the ability to create Python bytecode for custom DSLs, or even to completely implement other programming languages on top of Python VMs.

Import hooks

Taking advantage of MacroPy's ability to modify original ASTs is as easy as using the `import macropy.activate` statement, because it is able to somehow override the Python import behavior. It is not magic and Python provides a way to intercept imports for every developer using the following two kinds of import hooks:

- **Meta hooks**: These are called before any other `import` processing has occurred. Using meta hooks, you can override the way in which `sys.path` is processed for even frozen and built-in modules. To add a new meta hook, a new **meta path finder** object must be added to the `sys.meta_path` list.

- **Import path hooks**: These are called as part of `sys.path` processing. They are used if the path item associated with the given hook is encountered. The import path hooks are added by extending the `sys.path_hooks` list with a new **path entry finder** object.

The details of implementing both path finders and meta path finders are extensively documented in the official Python documentation (see `https://docs.python.org/3/reference/import.html`). The official documentation should be your primary resource if you want to interact with imports on that level. This is so because the import machinery in Python is rather complex and any attempt to summarize it in a few paragraphs would inevitably fail. Here, we just want to make you aware that such things are possible.

We'll take a look at projects that use code generation patterns in the following sections.

Notable examples of code generation in Python

It is hard to find a really usable implementation of the library that relies on code generation patterns that is not only an experiment or simple proof of concept. The reasons for that situation are fairly obvious:

- Deserved fear of the `exec()` and `eval()` functions because, if used irresponsibly, they can cause real disasters

- Successful code generation is very difficult to develop and maintain because it requires a deep understanding of the language and exceptional programming skills in general

Despite these difficulties, there are some projects that successfully take this approach either to improve performance or achieve things that would be impossible by other means.

Falcon's compiled router

Falcon (`http://falconframework.org/`) is a minimalist Python WSGI web framework for building fast and lightweight web APIs. It strongly encourages the REST architectural style that is currently very popular around the web. It is a good alternative to other rather heavy frameworks, such as Django or Pyramid. It is also a strong competitor for other micro-frameworks that aim for simplicity, such as Flask, Bottle, or web2py.

One of the best Falcon features is its very simple routing mechanism. It is not as complex as the routing provided by Django `urlconf` and does not provide as many features, but in most cases is just enough for any API that follows the REST architectural design. What is most interesting about Falcon's routing is the internal construction of that router. Falcon's router is implemented using the code generated from the list of routes, and the code changes every time a new route is registered. This is the effort that's needed to make routing fast.

Consider this very short API example, taken from Falcon's web documentation:

```python
# sample.py
import falcon
import json


class QuoteResource:
    def on_get(self, req, resp):
        """Handles GET requests"""
        quote = {
            'quote': 'I\'ve always been more interested in '
                     'the future than in the past.',
            'author': 'Grace Hopper'
        }

        resp.body = json.dumps(quote)

api = falcon.API()
api.add_route('/quote', QuoteResource())
```

In short, the call to the `api.add_route()` method dynamically updates the whole generated code tree for Falcon's request router. It also compiles it using the `compile()` function and generates the new route-finding function using `eval()`. Let's take a closer look at the following `__code__` attribute of the `api._router._find()` function:

```
>>> api._router._find.__code__
<code object find at 0x00000000033C29C0, file "<string>", line 1>
>>> api.add_route('/none', None)
>>> api._router._find.__code__
<code object find at 0x00000000033C2810, file "<string>", line 1>
```

This transcript shows that the code of this function was generated from the string and not from the real source code file (the "`<string>`" file). It also shows that the actual code object changes with every call to the `api.add_route()` method (the object's address in memory changes).

Hy

Hy (`http://docs.hylang.org/`) is the dialect of **Lisp** that is written entirely in Python. Many similar projects that implement other programming languages in Python usually try only to tokenize the plain form of code that's provided either as a file-like object or string and interpret it as a series of explicit Python calls. Unlike others, Hy can be considered as a language that runs fully in the Python runtime environment, just like Python does. Code written in Hy can use the existing built-in modules and external packages and vice versa. Code written with Hy can be imported back into Python.

To embed Lisp in Python, Hy translates Lisp code directly into Python AST. Import interoperability is achieved using the `import` hook that is registered once the Hy module is imported into Python. Every module with the `.hy` extension is treated as a Hy module and can be imported like the ordinary Python module. The following is a "hello world" program written in this Lisp dialect:

```
;; hyllo.hy
(defn hello [] (print "hello world!"))
```

It can be imported and executed with the following Python code:

```
>>> import hy
>>> import hyllo
>>> hyllo.hello()
    hello world!
```

If we dig deeper and try to disassemble `hyllo.hello` using the built-in `dis` module, we will notice that the bytecode of the Hy function does not differ significantly from its pure Python counterpart, as shown in the following code:

```
>>> import dis
>>> dis.dis(hyllo.hello)
  2           0 LOAD_GLOBAL          0 (print)
              3 LOAD_CONST           1 ('hello world!')
              6 CALL_FUNCTION        1 (1 positional, 0 keyword pair)
              9 RETURN_VALUE
>>> def hello(): print("hello world!")
...
>>> dis.dis(hello)
  1           0 LOAD_GLOBAL          0 (print)
              3 LOAD_CONST           1 ('hello world!')
              6 CALL_FUNCTION        1 (1 positional, 0 keyword pair)
              9 POP_TOP
             10 LOAD_CONST           0 (None)
             13 RETURN_VALUE
```

As you can see, the bytecode for the Hy-based function is shorter than the bytecode for the plain Python counterpart. Maybe a similar tendency can be observed for larger chunks of code. It shows that creating a completely new language on top of a Python VM is definitely possible and may be worth experimenting with.

Summary

In this chapter, we've explored the vast topic of metaprogramming in Python. We've described the syntax features that favor the various metaprogramming patterns in detail. These are mainly decorators and metaclasses.

We've also taken a look at another important aspect of metaprogramming, dynamic code generation. We described it only briefly as it is too vast to fit into the limited size of this book. However, it should be a good starting point that gives you a quick summary of the possible options in that field.

With the example of Hy, we've seen that metaprogramming can even be used to implement other languages on top of the Python runtime. The road taken by Hy developers is of course quite unusual and, generally the best way to bridge Python with other languages is through custom Python interpreter extensions or using shared libraries and foreign function interfaces. And these are exactly the topics of the next chapter.

9
Bridging Python with C and C++

Python is great but it isn't suited for everything. Sometimes you may find that particular problems can be solved more easily with a different language. Those different languages may be better due to greater expressiveness for certain technical domains (like control engineering, image processing, or system programming) or offer natural performance gains over Python. And Python (together with default CPython implementation) has a few characteristics that aren't necessarily good for performance:

- Threading usability is greatly reduced for CPU-bound tasks due to the existence of **Global Interpreter Lock** (**GIL**) in CPython and is dependent on the Python implementation of choice

- Python is not a compiled language (in the way C and Go are) so it lacks many compile-time optimizations

- Python does not provide static typing and the possible optimizations that come with it

But the fact that some languages are better for specific tasks doesn't mean that you have to completely forgo Python when faced with such problems. With proper techniques, it is possible to write applications that take advantage of many technologies.

One such technique is architecting applications as independent components that communicate with each other through well-defined communication channels. This often comes in the form of **service-oriented** or **microservice architectures**. This is extremely common in distributed systems where every component (service) of system architecture can run independently on a different host. Systems written in multiple languages are often nicknamed **polyglot systems**.

The disadvantage of using polyglot service-oriented or microservice architectures is that you will usually have to recreate a lot of application scaffolding for every such language. This includes things like application configuration, logging, monitoring, and communication layers, as well as different frameworks, libraries, build tools, common conventions, and design patterns. Introducing those tools and conventions will cost time and future maintenance effort, which can often exceed the gains of adding another language to your architecture.

Fortunately, there's another way to overcome this problem. Often what we really need from a different language can be packaged as an isolated library that does one thing and does it well. What we need to do is to find a bridge between Python and other languages that will allow us to use their libraries in Python applications. This can be done either through custom CPython extensions or so-called **Foreign Function Interfaces** (**FFIs**).

In both cases, the C and C++ programming languages act as a gateway to libraries and code written in different languages. The CPython interpreter is itself written in C and provides the Python/C API (defined in the `Python.h` header file) that allows you to create shared C libraries that can be loaded by the interpreter. C (and C++, due to its native interoperability with the C language) can be used to create such extensions. The FFIs on the other hand can be used to interact with any compatible compiled shared library regardless of the language it is written in. These libraries will still rely on C calling conventions and basic types.

This chapter will discuss the main reasons for writing your own extensions in other languages and introduce you to the popular tools that help to create them. We will learn about the following topics in this chapter:

- C and C++ as the core of Python extensibility
- Compiling and loading Python C extensions
- Writing extensions
- Downsides of using extensions
- Interfacing with compiled dynamic libraries without extensions

In order to bridge Python with different languages, we will need a handful of extra tools and libraries so let's take a look at the technical requirements for this chapter.

Technical requirements

In order to compile the Python extensions mentioned in this chapter, you will need C and C++ compilers. The following are suitable compilers that you can download for free on selected operating systems:

- Visual Studio 2019 (Windows): `https://visualstudio.microsoft.com`
- GCC (Linux and most POSIX systems): `https://gcc.gnu.org`
- Clang (Linux and most POSIX systems): `https://clang.llvm.org`

On Linux, GCC and Clang compilers are usually available through package management systems specific to the given system distribution. On macOS, the compiler is part of the Xcode IDE (available through the App Store).

The following are Python packages that are mentioned in this chapter that you can download from PyPI:

- `Cython`
- `Cffi`

Information on how to install packages is included in *Chapter 2*, *Modern Python Development Environments*.

The code files for this chapter can be found at `https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition/tree/main/Chapter%209`.

C and C++ as the core of Python extensibility

The reference implementation of Python—the CPython interpreter—is written in C. Because of that, Python interoperability with other languages revolves around C and C++, which has native interoperability with C. There is even a full superset of the Python language called Cython, which uses a source-to-source compiler for creating C extensions for CPython using extended Python syntax.

In fact, you can use dynamic/shared libraries written in any language if the language supports compilation in the form of dynamic/shared libraries. So, interlanguage integration possibilities go way beyond C and C++. This is because shared libraries are intrinsically generic. They can be used in any language that supports their loading. So, even if you write such a library in a completely different language (let's say Delphi or Prolog), you can use it in Python. Still, it is hard to call such a library a Python extension if it does not use the Python/C API.

Unfortunately, writing your own extensions only in C or C++ using the bare Python/C API is quite demanding. Not only because it requires a good understanding of one of the two languages that are relatively hard to master, but also because it requires an exceptional amount of boilerplate. You will have to write a lot of repetitive code that is used only to provide an interface that will glue your core C or C++ code with the Python interpreter and its datatypes.

Anyway, it is good to know how pure C extensions are built because of the following reasons:

- You will better understand how Python works in general
- One day, you may need to debug or maintain a native C/C++ extension
- It helps in understanding how higher-level tools for building extensions work

That's why in this chapter we will first learn how to build a simple Python C extension from scratch. We will later reimplement it with different techniques that do not require the usage of the low-level Python/C API.

But before we dive into the details of writing extensions, let's see how to Compile and load one.

Compiling and loading Python C extensions

The Python interpreter is able to load extensions from dynamic/shared libraries such as Python modules if they provide an applicable interface using the Python/C API. The definition of all functions, types, and macros constituting the Python/C API is included in a `Python.h` C header file that is distributed with Python sources. In many distributions of Linux, this header file is contained in a separate package (for example, `python-dev` in Debian/Ubuntu) but under Windows, it is distributed by default with the interpreter. On POSIX and POSIX-compatible systems (for example, Linux and macOS), it can be found in the `include/` directory of your Python installation. On Windows, it can be found in the `Include/` directory of your Python installation.

The Python/C API traditionally changes with every release of Python. In most cases, these are only additions of new features to the API so are generally **source-compatible**. Anyway, in most cases, they are not **binary-compatible** due to changes in the **Application Binary Interface** (**ABI**). This means that extensions must be compiled separately for every major version of Python. Also, different operating systems have incompatible ABIs, so this makes it practically impossible to create a single binary distribution for every possible environment. This is the reason why most Python extensions are distributed in source form.

Since Python 3.2, a subset of the Python/C API has been defined to have a stable ABI. Thanks to this, it is possible to build extensions using this limited API (with a stable ABI), so extensions can be compiled only once for a given operating system and it will work with any version of Python higher than or equal to 3.2 without the need for recompilation. Anyway, this limits the number of API features and does not solve the problems of older Python versions. It also does not allow you to create a single binary distribution that would work on multiple operating systems. This is a trade-off and the price of the stable ABI sometimes may be a bit too high for a very low gain.

It is important to know that the Python/C API is a feature that is limited only to CPython implementations. Some efforts were made to bring extension support to alternative implementations such as PyPI, Jython, or IronPython, but it seems that there is no stable and complete solution for them at the moment. The only alternative Python implementation that should deal easily with extensions is Stackless Python because it is in fact only a modified version of CPython.

C extensions for Python need to be compiled into shared/dynamic libraries before they can be imported. There is no native way to import C/C++ code in Python directly from sources. Fortunately, the `setuptools` package provides helpers to define compiled extensions as modules, so compilation and distribution can be handled using the `setup.py` script as if they were ordinary Python packages.

> We will learn more details about creating Python packages in *Chapter 11*, *Packaging and Distributing Python Code*.

The following is an example of the `setup.py` script from the official documentation that handles the preparation of a simple package distribution that has an extension written in C:

```python
from setuptools import setup, Extension
```

```python
module1 = Extension(
    'demo',
    sources=['demo.c']
)


setup(
    name='PackageName',
    version='1.0',
    description='This is a demo package',
    ext_modules=[module1]
)
```

> We will learn more about distributing Python packages and the `setup.py` script in *Chapter 11, Packaging and Distributing Python Code.*

Once prepared this way, the following additional step is required in your distribution flow:

```
python3 setup.py build
```

This step will compile all your extensions defined as the `ext_modules` argument according to all additional compiler settings provided with the `Extension()` constructor. The compiler that will be used is the one that is a default for your environment. This compilation step is not required if the package is going to be distributed as a source distribution. In that case, you need to be sure that the target environment has all the compilation prerequisites such as the compiler, header files, and additional libraries that are going to be linked to your binary (if your extension needs any). More details of packaging the Python extensions will be explained later in the *Downsides of using extensions* section.

In the next section, we will discuss why you may need to use extensions.

The need to use extensions

It's not easy to say when it is a reasonable decision to write extensions in C/C++. The general rule of thumb could be "never unless you have no other choice". But this is a very subjective statement that leaves a lot of place for the interpretation of what is not doable in Python. In fact, it is hard to find a thing that cannot be done using pure Python code.

Still, there are some problems where extensions may be especially useful by adding the following benefits:

- Bypassing GIL in the CPython threading model
- Improving performance in critical code sections
- Integrating source code written in different languages
- Integrating third-party dynamic libraries
- Creating efficient custom datatypes

Of course, for every such problem, there is usually a viable native Python solution. For example, the core CPython interpreter constraints, such as GIL, can easily be overcome with a different approach to concurrency, such as coroutines or multiprocessing, instead of a threading model (we discussed these options in *Chapter 6, Concurrency*). To work around third-party dynamic libraries and custom datatypes, third-party libraries can be integrated with the `ctypes` module, and every datatype can be implemented in Python.

Still, the native Python approach may not always be optimal. The Python-only integration of an external library may be clumsy and hard to maintain. The implementation of custom datatypes may be suboptimal without access to low-level memory management. So, the final decision of what path to take must always be taken very carefully and take many factors into consideration. A good approach is to start with a pure Python implementation first and consider extensions only when the native approach proves to be not good enough.

The next section will explain how extensions can be used to improve the performance in critical code sections.

Improving performance in critical code sections

Let's be honest, Python is not chosen by developers because of its performance. It does not execute fast but allows you to develop fast. Still, no matter how performant we are as programmers, thanks to this language, we may sometimes find a problem that may not be solved efficiently using pure Python.

In most cases, solving performance problems is really mostly about choosing proper algorithms and data structures and not about limiting the constant factor of language overhead. Usually, it is not a good approach to rely on extensions in order to shave off some CPU cycles if the code is already written poorly or does not use efficient algorithms.

It is often possible that performance can be improved to an acceptable level without the need to increase the complexity of your project by adding yet another language to your technology stack. And if it is possible to use only one programming language, it should be done that way in the first place.

Anyway, it is also very likely that even with a state-of-the-art algorithmic approach and the best-suited data structures, you will not be able to fit some arbitrary technological constraints using Python alone.

The example field that puts some well-defined limits on the application's performance is the **Real-Time Bidding** (**RTB**) business. In short, the whole of RTB is about buying and selling advertisement inventory (places for online ads) in a way that is similar to how real auctions or stock exchanges work. The whole trading usually takes place through some ad exchange service that sends the information about available inventory to **demand-side platforms** (**DSPs**) interested in buying areas for their advertisements. And this is the place where things get exciting. Most of the ad exchanges use the OpenRTB protocol (which is based on HTTP) for communication with potential bidders. The DSP is the site responsible for serving responses to its OpenRTB HTTP requests. And ad exchanges always put very strict time constraints on how long the whole process can take. It can be as little as 50 ms—from the first TCP packet received to the last byte written by the DSP server. To spice things up, it is not uncommon for DSP platforms to process tens of thousands of requests per second. Being able to shave off a few milliseconds from the response times often determines service profitability. This means that porting even trivial code to C may be reasonable in that situation but only if it's a part of some performance bottleneck and cannot be improved any further algorithmically. As Guido once said:

> *If you feel the need for speed, (...) – you can't beat a loop written in C.*

A completely different use-case for custom extensions is integrating code written in different languages, which is explained in the next section.

Integrating existing code written in different languages

Although computer science is young when compared to other fields of technical studies, we are already standing on the shoulders of giants. Many great programmers have written a lot of useful libraries for solving common problems using many programming languages. It would be a great loss to forget about all that heritage every time a new programming language pops out, but it is also impossible to reliably port any piece of software that was ever written to every new language.

The C and C++ languages seem to be the most important languages that provide a lot of libraries and implementations that you would like to integrate into your application code without the need to port them completely to Python. Fortunately, CPython is already written in C, so the most natural way to integrate such code is precisely through custom extensions.

The next section explains a very similar use-case: integrating third-party dynamic libraries.

Integrating third-party dynamic libraries

Integrating code written using different technologies does not end with C/C++. A lot of libraries, especially third-party software with closed sources, are distributed as compiled binaries. In C, it is really easy to load such shared/dynamic libraries and call their functions. This means that you can use any C library as long as you wrap it as a Python extension using the Python/C API.

This, of course, is not the only solution and there are tools such as `ctypes` and `CFFI` that allow you to interact with dynamic libraries directly using pure Python code without the need for writing extensions in C. Very often, the Python/C API may still be a better choice because it provides better separation between the integration layer (written in C) and the rest of your application.

Last but not least, extensions can be used to enhance Python with novel and performant data structures.

Creating efficient custom datatypes

Python provides a very versatile selection of built-in datatypes. Some of them really use state-of-the-art internal implementations (at least in CPython) that are specifically tailored for usage in the Python language. The number of basic types and collections available out of the box may look impressive for newcomers, but it is clear that it does not cover all of a programmer's needs.

You can, of course, create many custom data structures in Python, either by subclassing built-in types or by building them from scratch as completely new classes. Unfortunately, sometimes the performance of such a data structure may be suboptimal. The whole power of complex collections such as `dict` or `set` comes from their underlying C implementation. Why not do the same and implement some of your custom data structures in C too?

Since we already know the possible reasons to create custom Python extensions, let's see how to actually build one.

Writing extensions

As already said, writing extensions is not a simple task but, in return for your hard work, it can give you a lot of advantages. The easiest approach to creating extensions is to use tools such as Cython. Cython allows you to write C extensions using language that greatly resembles Python without all the intricacies of the Python/C API. It will increase your productivity and make code easier to develop, read, and maintain.

Anyway, if you are new to this topic, it is good to start your adventure with extensions by writing one using nothing more than bare C language and the Python/C API. This will improve your understanding of how extensions work and will also help you to appreciate the advantages of alternative solutions. For the sake of simplicity, we will take a simple algorithmic problem as an example and try to implement it using the two following different approaches:

- Writing a pure C extension
- Using Cython

Our problem will be finding the *n*th number of the Fibonacci sequence. This is a sequence of numbers where each element is the sum of two preceding ones. The sequence starts with 0 and 1. The first 10 numbers of the sequence are as follows:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

As you see, the sequence is easy to explain and also easy to implement. It is very unlikely that you would need to create a compiled extension solely for solving this problem. But it is very simple so it will serve as a very good example of wiring any C function to the Python/C API. Our goals are only clarity and simplicity, so we won't try to provide the most efficient solution.

Before we create our first extension, let's define a reference implementation that will allow us to compare different solutions. Our reference implementation of the Fibonacci function implemented in pure Python looks as follows:

```python
"""Python module that provides fibonacci sequence function"""


def fibonacci(n):
    """Return nth Fibonacci sequence number computed recursively."""
    if n == 0:
        return 0
    if n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Note that this is one of the most simple implementations of the `fibonnaci()` function. A lot of improvements could be applied to it. We don't optimize our implementation (using a memoization pattern, for instance) because this is not the purpose of our example. In the same manner, we won't optimize our code later when discussing implementations in C or Cython, even though the compiled code gives us many more possibilities to do so.

> Memoization is a popular technique of saving past results of function calls for later reference to optimize application performance. We explain it in detail in *Chapter 13*, *Code Optimization*.

Let's look into pure C extensions in the next section.

Pure C extensions

If you have decided that you need to write C extensions for Python, I assume that you already know the C language at a level that will allow you to fully understand the examples that are presented. This book is about Python, and as such nothing other than the Python/C API details will be explained here. This API, despite being crafted with great care, is definitely not a good introduction to C, so if you don't know C at all, you should avoid attempting to write Python extensions in C until you gain experience in this language. Leave it to others and stick with Cython or Pyrex, which are a lot safer from a beginner's perspective.

As announced earlier, we will try to port the `fibonacci()` function to C and expose it to the Python code as an extension. Let's start with a base implementation that would be analogous to the previous Python example. The bare function without any Python/C API usage could be roughly as follows:

```c
long long fibonacci(unsigned int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fibonacci(n - 2) + fibonacci(n - 1);
    }
}
```

And here is the example of a complete, fully functional extension that exposes this single function in a compiled module:

```c
#define PY_SSIZE_T_CLEAN
#include <Python.h>

long long fibonacci(unsigned int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fibonacci(n - 2) + fibonacci(n - 1);
    }
}

static PyObject* fibonacci_py(PyObject* self, PyObject* args) {
    PyObject *result = NULL;
    long n;

    if (PyArg_ParseTuple(args, "l", &n)) {
        result = Py_BuildValue("L", fibonacci((unsigned int)n));
    }

    return result;
}

static char fibonacci_docs[] =
    "fibonacci(n): Return nth Fibonacci sequence number "
    "computed recursively\n";


static PyMethodDef fibonacci_module_methods[] = {
    {"fibonacci", (PyCFunction)fibonacci_py,
     METH_VARARGS, fibonacci_docs},
    {NULL, NULL, 0, NULL}
};


static struct PyModuleDef fibonacci_module_definition = {
    PyModuleDef_HEAD_INIT,
    "fibonacci",
```

```
    "Extension module that provides fibonacci sequence function",
    -1,
    fibonacci_module_methods
};


PyMODINIT_FUNC PyInit_fibonacci(void) {
    Py_Initialize();

    return PyModule_Create(&fibonacci_module_definition);
}
```

I know what you think. The preceding example might be a bit overwhelming at first glance. We had to add four times more code just to make the `fibonacci()` C function accessible from Python. We will discuss every bit of that code step by step later, so don't worry. But before we do that, let's see how it can be packaged and executed in Python.

The following minimal `setuptools` configuration for our module needs to use the `setuptools.Extension` class in order to instruct the interpreter how our extension is compiled:

```
from setuptools import setup, Extension

setup(
    name='fibonacci',
    ext_modules=[
        Extension('fibonacci', ['fibonacci.c']),
    ]
)
```

The build process for extensions can be initialized with the `setup.py build` command, but it will also be automatically performed upon package installation. The following transcript presents the result of the installation in editable mode (using `pip` with the `-e` flag):

```
$ python3 -m pip install -e .
Obtaining file:///Users/.../Expert-Python-Programming-Fourth-Edition/
Chapter%209/02%20-%20Pure%20C%20extensions
Installing collected packages: fibonacci
  Running setup.py develop for fibonacci
Successfully installed fibonacci
```

Using the editable mode of `pip` allows us to take a peek at files created during the `build` step. The following is an example of files that could be created in your working directory during the installation:

```
$ ls -1ap
./
../
build/
fibonacci.c
fibonacci.cpython-39-darwin.so
fibonacci.egg-info/
setup.py
```

The `fibonacci.c` and `setup.py` files are our source files. `fibonacci.egg-info/` is a special directory that stores package metadata, and we should not be concerned about it at the moment. What is really important is the `fibonacci.cpython-39-darwin.so` file. This is our binary shared library that is compatible with the CPython interpreter. That's the library that the Python interpreter will load when we attempt to import our `fibonacci` module. Let's try to import it and review it in an interactive session:

```
$ python3
Python 3.9.1 (v3.9.1:1e5d33e9b9, Dec  7 2020, 12:10:52)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import fibonacci
>>> help(fibonacci)
Help on module fibonacci:

NAME
    fibonacci - Extension module that provides fibonacci sequence
function

FUNCTIONS
    fibonacci(...)
        fibonacci(n): Return nth Fibonacci sequence number computed
recursively

FILE
    /(...)/fibonacci.cpython-39-darwin.so
>>> [fibonacci.fibonacci(n) for n in range(10)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Now let's take a closer look at the anatomy of our extension.

A closer look at the Python/C API

Since we know how to properly package, compile, and install a custom C extension and we are sure that it works as expected, now is the right time to discuss our code in detail.

The extensions module starts with the following single C preprocessor directive, which includes the `Python.h` header file:

```
#include <Python.h>
```

This pulls the whole Python/C API and is everything you need to include to be able to write your extensions. In more realistic cases, your code will require a lot more preprocessor directives to benefit from the C standard library functions or to integrate other source files. Our example was simple, so no more directives were required.

Next, we have the core of our module as follows:

```
long long fibonacci(unsigned int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fibonacci(n - 2) + fibonacci(n - 1);
    }
}
```

The preceding `fibonacci()` function is the only part of our code that does something useful. It is a pure C implementation that Python by default can't understand. The rest of our example will create the interface layer that will expose it through the Python/C API.

The first step of exposing this code to Python is the creation of the C function that is compatible with the CPython interpreter. In Python, everything is an object. This means that C functions called in Python also need to return real Python objects. Python/C APIs provide a `PyObject` type and every callable must return the pointer to it. The signature of our function is as follows:

```
static PyObject* fibonacci_py(PyObject* self, PyObject* args)
```

Note that the preceding signature does not specify the exact list of arguments, but `PyObject* args` will hold the pointer to the structure that contains the tuple of the provided values.

The actual validation of the argument list must be performed inside the function body and this is exactly what `fibonacci_py()` does. It parses the `args` argument list assuming it is the single `unsigned int` type and uses that value as an argument to the `fibonacci()` function to retrieve the Fibonacci sequence element as shown in the following code:

```
static PyObject* fibonacci_py(PyObject* self, PyObject* args) {
    PyObject *result = NULL;
    long n;

    if (PyArg_ParseTuple(args, "l", &n)) {
        result = Py_BuildValue("L", fibonacci((unsigned int)n));
    }

    return result;
}
```

> The preceding example function has a serious bug, which the eyes of an experienced developer should spot very easily. Try to find it as an exercise in working with C extensions. For now, we'll leave it as it is for the sake of brevity. We will try to fix it later when discussing the details of dealing with errors and exceptions in the *Exception handling* section.

The `"l"` (lowercase L) string in the `PyArg_ParseTuple(args, "l", &n)` call means that we expect `args` to contain only a single `long` value. In the case of failure, it will return `NULL` and store information about the exception in the per-thread interpreter state.

The actual signature of the parsing function is `int PyArg_ParseTuple(PyObject *args, const char *format, ...)` and what goes after the format string is a variable-length list of arguments that represents parsed value output (as pointers). This is analogous to how the `scanf()` function from the C standard library works. If our assumption fails and the user provides an incompatible arguments list, then `PyArg_ParseTuple()` will raise the proper exception. This is a very convenient way to encode function signatures once you get used to it but has a huge downside when compared to plain Python code. Such Python call signatures implicitly defined by the `PyArg_ParseTuple()` calls cannot be easily inspected inside the Python interpreter. You need to remember this fact when using the code provided as extensions.

As already said, Python expects objects to be returned from callables. This means that we cannot return a raw long value obtained from the `fibonacci()` function as a result of `fibonacci_py()`. Such an attempt would not even compile and there is no automatic casting of basic C types to Python objects.

The `Py_BuildValue(*format, ...)` function must be used instead. It is the counterpart of `PyArg_ParseTuple()` and accepts a similar set of format strings. The main difference is that the list of arguments is not a function output but an input, so actual values must be provided instead of pointers.

After `fibonacci_py()` is defined, most of the heavy work is done. The last step is to perform module initialization and add metadata to our function that will make usage a bit simpler for the users. This is the boilerplate part of our extension code. For simple examples, such as this one, it can take up more space than the actual functions that we want to expose. In most cases, it simply consists of some static structures and one initialization function that will be executed by the interpreter on module import.

At first, we create a static string that will be the content of the Python docstring for the `fibonacci_py()` function as follows:

```
static char fibonacci_docs[] =
    "fibonacci(n): Return nth Fibonacci sequence number "
    "computed recursively\n";
```

Note that this could be inlined somewhere later in `fibonacci_module_methods`, but it is a good practice to have docstrings separated and stored in close proximity to the actual function definition that they refer to.

The next part of our definition is the array of the `PyMethodDef` structures that define methods (functions) that will be available in our module. The `PyMethodDef` structure contains exactly four fields:

- `char* ml_name`: This is the name of the method.
- `PyCFunction ml_meth`: This is the pointer to the C implementation of the function.
- `int ml_flags`: This includes the flags indicating either the calling convention or binding convention. The latter is applicable only for the definition of class methods.
- `char* ml_doc`: This is the pointer to the content of the method/function docstring.

Such an array must always end with a sentinel value of `{NULL, NULL, 0, NULL}`. This sentinel value simply indicates the end of the structure. In our simple case, we created the static `PyMethodDef fibonacci_module_methods[]` array that contains only two elements (including sentinel value):

```
static PyMethodDef fibonacci_module_methods[] = {
    {"fibonacci", (PyCFunction)fibonacci_py,
     METH_VARARGS, fibonacci_docs},
```

```
    {NULL, NULL, 0, NULL}
};
```

And this is how the first entry maps to the `PyMethodDef` structure:

- `ml_name = "fibonacci"`: Here, the `fibinacci_py()` C function will be exposed as a Python function under the `fibonacci` name.

- `ml_meth = (PyCFunction)fibonacci_py`: Here, the casting to `PyCFunction` is simply required by the Python/C API and is dictated by the call convention defined later in `ml_flags`.

- `ml_flags = METH_VARARGS`: Here, the `METH_VARARGS` flag indicates that the calling convention of our function accepts a variable list of arguments and no keyword arguments.

- `ml_doc = fibonacci_docs`: Here, the Python function will be documented with the content of the `fibonacci_docs` string.

When an array of function definitions is complete, we can create another structure that contains the definition of the whole module. It is described using the `PyModuleDef` type and contains multiple fields. Some of them are useful only for more complex scenarios, where fine-grained control over the module initialization process is required. Here, we are interested only in the first five of them:

- `PyModuleDef_Base m_base`: This should always be initialized with `PyModuleDef_HEAD_INIT`.

- `char* m_name`: This is the name of the newly created module. In our case, it is `fibonacci`.

- `char* m_doc`: This is the pointer to the docstring content for the module. We usually have only a single module defined in one C source file, so it is OK to inline our documentation string in the whole structure.

- `Py_ssize_t m_size`: This is the size of the memory allocated to keep the module state. This is used only when support for multiple subinterpreters or multiphase initialization is required. In most cases, you don't need that and it gets the value `-1`.

- `PyMethodDef* m_methods`: This is a pointer to the array containing module-level functions described by the `PyMethodDef` values. It could be `NULL` if the module does not expose any functions. In our case, it is `fibonacci_module_methods`.

The other fields are explained in detail in the official Python documentation (refer to `https://docs.python.org/3/c-api/module.html`) but are not needed in our example extension. They should be set to `NULL` if not required and they will be initialized with that value implicitly when not specified. This is why our module description contained in the `fibonacci_module_definition` variable can take the following simple form:

```
static struct PyModuleDef fibonacci_module_definition = {
    PyModuleDef_HEAD_INIT,
    "fibonacci",
    "Extension module that provides fibonacci sequence function",
    -1,
    fibonacci_module_methods
};
```

The last piece of code that crowns our work is the module initialization function. This must follow a very specific naming convention, so the Python interpreter can easily find it when the dynamic/shared library is loaded. It should be named `PyInit_<name>`, where `<name>` is the name of your module. So it is exactly the same string that was used as the `m_base` field in the `PyModuleDef` definition and as the first argument of the `setuptools.Extension()` call. If you don't require a complex initialization process for the module, it takes a very simple form, exactly like in our example:

```
PyMODINIT_FUNC PyInit_fibonacci(void) {
    return PyModule_Create(&fibonacci_module_definition);
}
```

The `PyMODINIT_FUNC` macro is a preprocessor macro that will declare the return type of this initialization function as `PyObject*` and add any special linkage declarations if required by the platform.

One very important difference between Python and C functions is the calling and binding conventions. This is quite a verbose topic, so let's discuss that in a separate section.

Calling and binding conventions

Python is an object-oriented language with flexible calling conventions using both positional and keyword arguments. Consider the following `print()` function call:

```
print("hello", "world", sep=" ", end="!\n")
```

The first two expressions provided to the call (the `"hello"` and `"world"` expressions) are positional and will be matched with the positional argument of the `print()` function. Order is important and if we modify it, the function call will give a different result. On the other hand, the following `" "` and `"!\n"` expressions will be matched with keyword arguments. Their order is irrelevant as long as the names don't change.

C is a procedural language with only positional arguments. When writing Python extensions, there is a need to support Python's argument flexibility and object-oriented data model. That is done mostly through the explicit declaration of supported calling and binding conventions.

As explained in the *A closer look at the Python/C API* section, the `ml_flags` bit field of the `PyMethodDef` structure contains flags for calling and binding conventions. Calling convention flags are as follows:

- `METH_VARARGS`: This is a typical convention for the Python function or method that accepts only arguments as its parameters. The type provided as the `ml_meth` field for such a function should be `PyCFunction`. The function will be provided with two arguments of the `PyObject*` type. The first is either the `self` object (for methods) or the `module` object (for module functions). A typical signature for the C function with that calling convention is `PyObject* function(PyObject* self, PyObject* args)`.

- `METH_KEYWORDS`: This is the convention for the Python function that accepts keyword arguments when called. Its associated C type is `PyCFunctionWithKeywords`. The C function must accept three arguments of the `PyObject*` type — `self`, `args`, and a dictionary of keyword arguments. If combined with `METH_VARARGS`, the first two arguments have the same meaning as for the previous calling convention, otherwise, `args` will be `NULL`. The typical C function signature is `PyObject* function(PyObject* self, PyObject* args, PyObject* keywds)`.

- `METH_NOARGS`: This is the convention for Python functions that do not accept any other argument. The C function should be of the `PyCFunction` type, so the signature is the same as that of the `METH_VARARGS` convention (with `self` and `args` arguments). The only difference is that `args` will always be `NULL`, so there is no need to call `PyArg_ParseTuple()`. This cannot be combined with any other calling convention flag.

- `METH_O`: This is the shorthand for functions and methods accepting single object arguments. The type of the C function is again `PyCFunction`, so it accepts two `PyObject*` arguments: `self` and `args`. Its difference from `METH_VARARGS` is that there is no need to call `PyArg_ParseTuple()` because `PyObject*` provided as `args` will already represent the single argument provided in the Python call to that function. This also cannot be combined with any other calling convention flag.

A function that accepts keywords is described either with `METH_KEYWORDS` or bitwise combinations of calling convention flags in the form of `METH_VARARGS | METH_KEYWORDS`. If so, it should parse its arguments with `PyArg_ParseTupleAndKeywords()` instead of `PyArg_ParseTuple()` or `PyArg_UnpackTuple()`.

Here is an example module with a single function that returns `None` and accepts two named arguments that are printed on standard output:

```c
#define PY_SSIZE_T_CLEAN
#include <Python.h>

static PyObject* print_args(PyObject *self, PyObject *args,
 PyObject *keywds)
{
    char *first;
    char *second;

    static char *kwlist[] = {"first", "second", NULL};

    if (!PyArg_ParseTupleAndKeywords(args, keywds, "ss", kwlist,
                                     &first, &second))
        return NULL;

    printf("%s %s\n", first, second);

    Py_INCREF(Py_None);
    return Py_None;
}


static PyMethodDef module_methods[] = {
    {"print_args", (PyCFunction)print_args,
     METH_VARARGS | METH_KEYWORDS,
     "print provided arguments"},
    {NULL, NULL, 0, NULL}
};


static struct PyModuleDef module_definition = {
    PyModuleDef_HEAD_INIT,
    "kwargs",
    "Keyword argument processing example",
    -1,
```

```
    module_methods
};


PyMODINIT_FUNC PyInit_kwargs(void) {
    return PyModule_Create(&module_definition);
}
```

> Argument parsing in the Python/C API is very elastic and is
> extensively described in the official documentation at `https://docs.python.org/3/c-api/arg.html`.

The format argument in `PyArg_ParseTuple()` and `PyArg_ParseTupleAndKeywords()` allows fine-grained control over the argument number and types. Every advanced calling convention known from Python can be coded in C with this API, including the following:

- Functions with default values for arguments
- Functions with arguments specified as keyword-only
- Functions with arguments specified as positional-only
- Functions with a variable number of arguments
- Functions without arguments

The additional binding convention flags `METH_CLASS`, `METH_STATIC`, and `METH_COEXIST` are reserved for methods and cannot be used to describe module functions. The first two are quite self-explanatory. They are C counterparts of the `@classmethod` and `@staticmethod` decorators and change the meaning of the `self` argument passed to the C function.

`METH_COEXIST` allows loading a method in place of the existing definition. It is useful very rarely. This is mostly in the case when you would like to provide an implementation of the C method that would be generated automatically from the other features of the type that was defined. The Python documentation gives the example of the `__contains__()` wrapper method that would be generated if the type has the `sq_contains` slot defined. Unfortunately, defining your own classes and types using the Python/C API is beyond the scope of this introductory chapter.

Let's take a look at exception handling in the next section.

Exception handling

C, unlike Python or even C++, does not have syntax for raising and catching exceptions. All error handling is usually handled with function return values and optional global state for storing details that can explain the cause of the last failure.

Exception handling in the Python/C API is built around that simple principle. There is a global per-thread indicator of the last error that occurred. It is set to describe the cause of a problem. There is also a standardized way to inform the caller of a function if this state was changed during the call, for example:

- If the function is supposed to return a pointer, it returns NULL
- If the function is supposed to return a value of type int, it returns -1

The only exceptions from the preceding rules in the Python/C API are the `PyArg_*()` functions that return 1 to indicate success and 0 to indicate failure.

To see how this works in practice, let's recall our `fibonacci_py()` function from the example in the previous sections:

```
static PyObject* fibonacci_py(PyObject* self, PyObject* args) {
    PyObject *result = NULL;
    long n;

    if (PyArg_ParseTuple(args, "l", &n)) {
        result = Py_BuildValue("L", fibonacci((unsigned int) n));
    }

    return result;
}
```

Error handling starts at the very beginning of our function with the initialization of the `result` variable. This variable is supposed to store the return value of our function. It is initialized with NULL, which, as we already know, is an indicator of error. And this is how you will usually code your extensions—assuming that error is the default state of your code.

Later we have the `PyArg_ParseTuple()` call that will set error information in the case of an exception and return 0. This is part of the `if` statement, so in the case of an exception, we don't do anything more and the function will return NULL. Whoever calls our function will be notified about the error.

`Py_BuildValue()` can also raise an exception. It is supposed to return `PyObject*` (pointer), so in the case of failure, it gives `NULL`. We can simply store this as our `result` variable and pass it on as a `return` value.

But our job does not end with caring for exceptions raised by Python/C API calls. It is very probable that you will need to inform the extension user about what kind of error or failure occurred. The Python/C API has multiple functions that help you to raise an exception but the most common one is `PyErr_SetString()`. It sets an error indicator with the given exception type and with the additional string provided as the explanation of the error cause. The full signature of this function is as follows:

```
void PyErr_SetString(PyObject* type, const char* message)
```

You could have already noticed a problematic issue in the `fibonacci_py()` function from the section *A closer look at the Python/C API*. If not, now is the right time to uncover it and fix it. Fortunately, we finally have the proper tools to do that.

The problem lies in the insecure casting of the long type to `unsigned int` in the following lines:

```
if (PyArg_ParseTuple(args, "l", &n)) {
    result = Py_BuildValue("L", fibonacci((unsigned int) n));
}
```

Thanks to the `PyArg_ParseTuple()` call, the first and only argument will be interpreted as a long type (the `"l"` specifier) and stored in the local n variable. Then it is cast to `unsigned int` so the issue will occur if the user calls the `fibonacci()` function from Python with a negative value. For instance, `-1` as a signed 32-bit integer will be interpreted as `4294967295` when casting to an unsigned 32-bit integer. Such a value will cause a very deep recursion and will result in a stack overflow and segmentation fault. Note that the same may happen if the user gives an arbitrarily large positive argument. We cannot fix this without a complete redesign of the C `fibonacci()` function, but we can at least try to ensure that the function input argument meets some preconditions. Here, we check whether the value of the n argument is greater than or equal to 0 and we raise a `ValueError` exception if that's not true, as follows:

```
static PyObject* fibonacci_py(PyObject* self, PyObject* args) {
    PyObject *result = NULL;
    long n;
    long long fib;

    if (PyArg_ParseTuple(args, "l", &n)) {
        if (n<0) {
```

```
            PyErr_SetString(PyExc_ValueError,
                            "n must not be less than 0");
        } else {
            result = Py_BuildValue("L", fibonacci((unsigned int) n));
        }
    }

    return result;
}
```

The last note about exception handling is that the global error state does not clear by itself. Some of the errors can be handled gracefully in your C functions (the same as using the `try ... except` clause in Python) and you need to be able to clear the error indicator if it is no longer valid. The function for that is `PyErr_Clear()`.

One of the great advantages of C extensions is the ability to bypass the GIL, which can be detrimental to threaded concurrency in Python applications. In the next section, we will discuss the possibility of releasing GIL in C extensions.

Releasing GIL

We have already mentioned that extensions can be a way to bypass Python's GIL. It is a famous limitation of the CPython implementation that only one thread at a time can execute the Python code. Multiprocessing is the suggested approach to circumvent this problem (see *Chapter 6*, *Concurrency*) but it may not be the best solution for some highly parallelizable algorithms, due to the resource overhead of running additional processes.

Because extensions are mostly used in cases where a bigger part of the work is performed in pure C without any calls to the Python/C API, it is possible (or even advisable) to release GIL in some application sections while doing non-Python data processing. Thanks to this, you can still benefit from having multiple CPU cores and multithreaded application designs. The only thing you need to do is to wrap blocks of code that are known to not use any of the Python/C API calls or Python structures with specific macros provided by the Python/C API. These two following preprocessor macros are provided to simplify the whole procedure of releasing and reacquiring the GIL:

- `Py_BEGIN_ALLOW_THREADS`: This declares the hidden local variable where the current thread state is saved and it releases GIL.

- `Py_END_ALLOW_THREADS`: This reacquires GIL and restores the thread state from the local variable declared with the previous macro.

When we look carefully at our `fibonacci` extension example, we can clearly see that the `fibonacci()` function does not execute any Python code and does not touch any of the Python structures. This means that the `fibonacci_py()` function that simply wraps the `fibonacci(n)` execution could be updated to release GIL around that call as follows:

```c
static PyObject* fibonacci_py(PyObject* self, PyObject* args) {
    PyObject *result = NULL;
    long n;
    long long fib;

    if (PyArg_ParseTuple(args, "l", &n)) {
        if (n<0) {
            PyErr_SetString(PyExc_ValueError,
                            "n must not be less than 0");
        } else {
            Py_BEGIN_ALLOW_THREADS;
            fib = fibonacci(n);
            Py_END_ALLOW_THREADS;

            result = Py_BuildValue("L", fib);
        }
    }

    return result;
}
```

Another important topic regarding the Python/C API is memory management and garbage collection. The most common garbage collection mechanism among dynamic programming languages is **tracing garbage collection**, which works by tracing whether objects can be reached from a program's root reference. If objects become unreachable, they can be released from program memory to reclaim memory space.

Python has a minimal tracing garbage collector for finding reference cycles but in fact uses reference counting as a main memory management mechanism. That's not a problem in plain Python code but adds some substantial work when writing C extensions. Let's dive deeper into this topic in the next section.

Tracing garbage collection is such a common garbage collection strategy that it is often treated as a synonym to garbage collection. That's why some people argue that Python isn't garbage collected (because it uses reference counting as the main memory management technique) and others argue that it is (because it uses tracing for finding reference cycles and reference counting can be understood as an alternative garbage collection strategy).

Reference counting

Finally, we come to the important topic of memory management in Python. Python has its own garbage collector, but it is designed only to solve the issue of cyclic references in the reference counting algorithm. Reference counting is the primary method of managing the deallocation of objects that are no longer needed.

The Python/C API documentation introduces ownership of references to explain how it deals with the deallocation of objects. Objects in Python are never owned by extension code and thus cannot be created or released by extensions themselves. The actual creation of objects is managed by Python's memory manager. That's why we say that objects in Python are owned by the memory manager.

The memory manager is the internal component of the CPython interpreter that is the only one responsible for allocating and deallocating memory for objects that are stored in a private heap. What can be owned instead is a reference to the object.

Every object in Python that is represented by a reference (`PyObject*` pointer) has an associated reference count. When it goes to zero, it means that no one holds any valid references to that object and the deallocator associated with its type can be invoked. The Python/C API provides a few macros for increasing and decreasing reference counts:

- `Py_INCREF()` and `Py_DECREF()`: The first one increases the reference count and the second one decreases it. These macros accept object pointers that must not be `NULL`.

- `Py_XINCREF()` and `Py_XDECREF()`: The first one increases the reference count and the second one decreases it. These macros accept `NULL` values so you should use them whenever you are not sure if you are dealing with `NULL` pointers.

But before we discuss their details, we need to understand the following terms related to reference ownership:

- **Passing of ownership**: Whenever we say that the function passes the ownership over a reference, it means that it has already increased the reference count and it is the responsibility of the caller to decrease the count when the reference to the object is no longer needed. Most of the functions that return the newly created objects, such as `Py_BuildValue`, are doing that. If that object is going to be returned from our function to another caller, then the ownership is passed again. We do not decrease the reference count in that case because it is no longer our responsibility. This is why the `fibonacci_py()` function does not call `Py_DECREF()` on the `result` variable.

- **Borrowed references**: The borrowing of references happens when the function receives a reference to some Python object as an argument. The reference count for such a reference should never be decreased in that function unless it was explicitly increased in its scope. In our `fibonacci_py()` function, the `self` and `args` arguments are such borrowed references and thus we do not call `PyDECREF()` on them. Some of the Python/C API functions may also return borrowed references. The notable examples are `PyTuple_GetItem()` and `PyList_GetItem()`. It is often said that such references are unprotected. There is no need to dispose of their ownership unless they will be returned as a function's return value. In most cases, extra care should be taken if we use such borrowed references as arguments of other Python/C API calls. It may be necessary in some circumstances to additionally protect such references with a separate `Py_INCREF()` call before using it as an argument to other functions and then calling `Py_DECREF()` when it is no longer needed. We'll see an example of such a situation at the end of the section.

- **Stolen references**: It is also possible for the Python/C API function to steal the reference instead of borrowing it when provided as a call argument. This is the case of exactly two functions—`PyTuple_SetItem()` and `PyList_SetItem()`. They fully take over the responsibility of the reference passed to them. They do not increase the reference count by themselves but will call `Py_DECREF()` when the reference is no longer needed.

Keeping an eye on the reference counts is one of the hardest things when writing complex extensions. Some of the non-obvious issues may not be noticed until the code is run in a multithreaded setup.

The other common problem is caused by the very nature of Python's object model and the fact that some functions return borrowed references. When the reference count goes to zero, the deallocation function is executed. For user-defined classes, it is possible to define a `__del__()` method that will be called at that moment.

This can be any Python code and it is possible that it will affect other objects and their reference counts. The official Python documentation gives the following example of code that may be affected by this problem:

```
void bug(PyObject *list) {
    PyObject *item = PyList_GetItem(list, 0);

    PyList_SetItem(list, 1, PyLong_FromLong(0L));
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```

It looks completely harmless, but the problem is in fact that we cannot know what elements the `list` object contains. When `PyList_SetItem()` sets a new value on the `list[1]` index, the ownership of the object that was previously stored at that index is disposed of. If it was the only existing reference, the reference count will become `0` and the object may be deallocated. It is possible that it was some user-defined class with a custom implementation of the `__del__()` method. A serious issue will occur if, in the result of such a `__del__()` execution, `item[0]` is removed from the list.

Note that `PyList_GetItem()` returns a borrowed reference! It does not call `Py_INCREF()` before returning a reference. So in that code, it is possible that `PyObject_Print()` will be called with a reference to an object that no longer exists. This will cause a segmentation fault and crash the Python interpreter.

The proper approach is to protect borrowed references for the whole time that we need them because there is a possibility that any call in between may cause the deallocation of that object. This can happen even if they are seemingly unrelated, as shown in the following code:

```
void no_bug(PyObject *list) {
    PyObject *item = PyList_GetItem(list, 0);

    Py_INCREF(item);
    PyList_SetItem(list, 1, PyLong_FromLong(0L));
    PyObject_Print(item, stdout, 0);
    Py_DECREF(item);
}
```

As you can see, writing Python extensions in C using the Python/C API can be a challenge. Especially if you are not experienced with C. It requires a lot of knowledge about CPython internals and precise memory management. But fortunately, there's an easier path to custom extensions. It is Cython, which is a special dialect of Python. We will discuss it in the next section.

Writing extensions with Cython

Cython is both an optimizing static compiler and the name of a programming language that is a superset of Python. It can be used to speed up Python applications by compiling them to machine code but can also be used as a "wrapping language" for code written in C or C++.

As a compiler, it performs the source-to-source compilation of native Python code and Cython dialect to Python C extensions using the Python/C API. It allows you to combine the power of Python and C without the need to manually deal with the Python/C API.

As a superset of Python, it offers the ability to use static typing, static linking of C libraries (as opposed to dynamic linking of shared libraries), the ability to interact with C header files, and direct control over CPython's GIL.

Let's first discuss Cython as a source-to-source compiler.

Cython as a source-to-source compiler

For extensions created using Cython, the major advantage you will get is using the superset language that it provides. Anyway, it is possible to create extensions from plain Python code using source-to-source compilation. This is the simplest approach to Cython because it requires almost no changes to the code and can give some significant performance improvements with very little effort.

To begin with, in order to build Cython extensions you will need the Cython package. It can be installed from PyPI using `pip`:

```
$ python3 -m pip install Cython
```

Cython provides a simple `cythonize` utility function that allows you to easily integrate the compilation process with the `setuptools` package. Let's assume that we would like to compile a pure Python implementation of our `fibonacci()` function to a C extension. If it is located in the `fibonacci.py` module, the minimal `setup.py` script could be as follows:

```python
from setuptools import setup
from Cython.Build import cythonize

setup(
    name='fibonacci',
    ext_modules=cythonize(['fibonacci.py'])
)
```

You can install such a module with `pip` the same way as you would do with a plain C extension:

```
$ python3 -m pip install -e .
Installing collected packages: fibonacci
   Running setup.py develop for fibonacci
Successfully installed fibonacci
```

The above command installs the package in editable mode so we can take a look at all files generated in the process. If you execute it in your own shell, you can see it creates some additional build artifacts:

```
$ ls -1ap
./
../
build/
fibonacci.c
fibonacci.cpython-39-darwin.so
fibonacci.egg-info/
fibonacci.py
setup.py
```

`fibonacci.c` in the preceding output is autogenerated C extension code. Cython translates the plain Python code into raw C code. During installation, this C code will be used to build the extension module library. In our case, it is the `fibonacci.cpython-39-darwin.so` file.

> You can take a look at the `fibonacci.c` file to see how much work Cython does behind the curtain. It is actually pretty long. For our simple `fibonacci.py` module, it can even be over 4000 lines long.

Cython, when used as a source compilation tool for the Python language, has another benefit. Source-to-source compilation to an extension can be a fully optional part of the source distribution installation process. If the environment where the package needs to be installed does not have Cython or any other building prerequisites, it can be installed as a normal pure Python package. The user should not notice any functional difference in the behavior of code distributed that way. A common approach for distributing extensions built with Cython is to include both Python/Cython sources and C code that would be generated from these source files.

This way, the package can be installed in the following three different ways, depending on the existence of building prerequisites:

- If the installation environment has Cython available, the extension C code is generated from the Python/Cython sources that are provided.
- If Cython is not available but there are available building prerequisites (C compiler, Python/C API headers), the extension is built from distributed pregenerated C files.
- If neither of the preceding is available but the extension is created from pure Python sources, the modules are installed like ordinary Python code, and the compilation step is skipped.

Note that the Cython documentation says that including generated C files as well as Cython sources is the recommended way of distributing Cython extensions. The same documentation says that Cython compilation should be disabled by default because the user may not have the required version of Cython in their environment, and this may result in unexpected compilation issues.

> You can read more about official guidelines on distribution Cython code at `https://cython.readthedocs.io/src/userguide/source_files_and_compilation.html`.

Anyway, with the advent of environment isolation, this seems to be a less worrying problem today. Also, Cython is a valid Python package that is available on PyPI, so it can easily be defined as your project requirement in a specific version. Including such a prerequisite is, of course, a decision with serious implications and should be considered very carefully. The safer solution is to leverage the power of the `extras_require` feature in the `setuptools` package and allow the user to decide whether they want to use Cython with a specific environment variable, for example:

```python
import os

from setuptools import setup, Extension

try:
    # cython source to source compilation
    # available only when Cython is available
    # and specific environment variable says
    # explicitly that Cython should be used
    # to generate C sources
```

```python
    USE_CYTHON = bool(os.environ.get("USE_CYTHON"))
    import Cython


except ImportError:
    USE_CYTHON = False

ext = '.pyx' if USE_CYTHON else '.c'

extensions = [Extension("fibonacci", ["fibonacci"+ext])]

if USE_CYTHON:
    from Cython.Build import cythonize
    extensions = cythonize(extensions)

setup(
    name='fibonacci',
    ext_modules=extensions,
    extras_require={
        # Cython will be set in that specific version
        # as a requirement if package will be installed
        # with '[with-cython]' extra feature
        'with-cython': ['cython==0.29.22']
    }
)
```

The `pip` installation tool supports the installation of packages with the `extras` option by adding the `[extra-name]` suffix to the package name. For the preceding example, the optional Cython requirement and compilation during the installation from local sources can be enabled using the following command:

```
$ USE_CYTHON=1 pip install .[with-cython]
```

The `USE_CYTHON` environment variable guarantees that `pip` will use Cython to compile `.pyx` sources to C and `[with-cython]` guarantees that the Cython compiler will be actually downloaded before installation.

Although you can use Cython to compile plain Python code, you will get the most benefit from using the Cython dialect. It has a few additional features that are not available in plain Python. We will take a closer look at Cython as a separate language in the next section.

Cython as a language

Cython is not only a compiler but also a superset of the Python language. Superset means that any valid Python code is allowed but it can be further enhanced with additional features, such as support for calling C functions or declaring C types on variables and class attributes. So, any code written in Python is also written in Cython but the reverse is not always true. This explains why ordinary Python modules can be so easily compiled to C using the Cython compiler.

But we won't stop at that simple fact. Instead of just saying that our reference `fibonacci()` function is also Cython code, we will try to improve it a bit. This won't be any real optimization because we still want to implement our Fibonacci sequence recursively. But we will do some minor updates that will allow it to benefit more from being written in Cython.

Cython sources use a different file extension. It is `.pyx` instead of `.py`. The content of the `fibonacci.pyx` file might look like this:

```python
"""Cython module that provides fibonacci sequence function."""


def fibonacci(unsigned int n):
    """Return nth Fibonacci sequence number computed recursively."""
    if n == 0:
        return 0
    if n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

As you can see, the only thing that has really changed is the signature of the `fibonacci()` function. Thanks to optional static typing in Cython, we can declare the n argument as `unsigned int` and this should slightly improve the way our function works. Additionally, it does a lot more than we did previously when writing extensions by hand. If the argument of the Cython function is declared with a static type, then the extension will automatically handle conversion and overflow errors by raising proper exceptions. The following is an example of an interactive session showing how our `fibonacci()` function written in Cython deals with conversion and overflow errors:

```pycon
>>> from fibonacci import fibonacci
>>> fibonacci(5)
5
>>> fibonacci(0)
0
```

```
>>> fibonacci(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "fibonacci.pyx", line 4, in fibonacci.fibonacci
    def fibonacci(unsigned int n):
OverflowError: can't convert negative value to unsigned int
>>> fibonacci(10 ** 10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "fibonacci.pyx", line 4, in fibonacci.fibonacci
    def fibonacci(unsigned int n):
OverflowError: value too large to convert to unsigned int
```

We already know that Cython compiles only source to source and the generated code uses the same Python/C API that we would use when writing C code for extensions by hand. Note that `fibonacci()` is a recursive function, so it calls itself very often. This will mean that although we declared a static type for the input argument, during the recursive call it will treat itself like any other Python function. So n-1 and n-2 will be packed back into the Python object and then passed to the hidden wrapper layer of the internal `fibonacci()` implementation that will again bring it back to the `unsigned int` type. This will happen again and again until we reach the final depth of recursion. This is not necessarily a problem but involves a lot more argument processing than is really required.

We can cut off the overhead of Python function calls and argument processing by delegating more of the work to the pure C function that does not know anything about Python structures. We did this previously when creating C extensions with pure C and we can do that in Cython too. We can use the `cdef` keyword to declare C-style functions that accept and return only C types as follows:

```
cdef long long fibonacci_cc(unsigned int n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    else:
        return fibonacci_cc(n - 1) + fibonacci_cc(n - 2)


def fibonacci(unsigned int n):
    """ Return nth Fibonacci sequence number computed recursively
    """
    return fibonacci_cc(n)
```

The `fibonacci_cc()` function will not be available to import in the final compiled `fibonacci` module. The `fibonacci()` function forms a façade to the low-level `fibonacci_cc()` implementation.

We can go even further. With a plain C example, we finally showed how to release GIL during the call of our pure C function, so that the extension was a bit nicer for multithreaded applications. In previous examples, we have used `Py_BEGIN_ALLOW_THREADS` and `Py_BEGIN_ALLOW_THREADS` preprocessor macros from Python/C API headers to mark a section of code as free from Python calls. The Cython syntax is a lot shorter and easier to remember. GIL can be released around the section of code using a simple `with nogil` statement like the following:

```python
def fibonacci(unsigned int n):
    """ Return nth Fibonacci sequence number computed recursively
    """
    with nogil:
        return fibonacci_cc(n)
```

You can also mark the whole C-style function as safe to call without GIL as follows:

```python
cdef long long fibonacci_cc(unsigned int n) nogil:
    if n < 2:
        return n
    else:
        return fibonacci_cc(n - 1) + fibonacci_cc(n - 2)
```

It is important to know that such functions cannot have Python objects as arguments or return types. Whenever a function marked as `nogil` needs to perform any Python/C API call, it must acquire GIL using the `with gil` statement.

We already know two ways of creating Python extensions: using plain C code with the Python/C API and using Cython. The first one gives you the most power and flexibility at the cost of quite complex and verbose code and the second one makes writing extensions easier but does a lot of magic behind your back. We also learned some potential advantages of extensions so it's time to take a closer look at some potential downsides.

Downsides of using extensions

To be honest, I started my adventure with Python only because I was tired of all the difficulty of writing software in C and C++. In fact, it is very common that programmers start to learn Python when they realize that other languages do not deliver what their users need.

Programming in Python, when compared to C, C++, or Java, is a breeze. Everything seems to be simple and well designed. You might think that there are no places where you can trip over and there are no other programming languages required anymore.

And of course, nothing could be more wrong. Yes, Python is an amazing language with a lot of cool features, and it is used in many fields. But it doesn't mean that it is perfect and doesn't have any downsides. It is easy to understand and write, but this easiness comes with a price. It is not as slow as many people think but will never be as fast as C. It is highly portable, but its interpreter is not available on as many architectures as compilers of some other languages are. We could go on with that list for a while.

One of the solutions to fix that problem is to write extensions. That gives us some ability to bring some of the advantages of good old C back to Python. And in most cases, it works well. The question is—are we really using Python because we want to extend it with C? The answer is no. This is only an inconvenient necessity in situations where we don't have any better options.

Extensions always come with a cost and one of the biggest downsides of using extensions is increased complexity.

Additional complexity

It is not a secret that developing applications in many different languages is not an easy task. Python and C are completely different technologies, and it is very hard to find anything that they have in common. It is also true that there is no application that is free of bugs. If extensions become common in your code base, debugging can become painful. Not only because the debugging of C code requires a completely different workflow and tools, but also because you will need to switch context between two different languages very often.

We are all human and we all have limited cognitive capabilities. There are, of course, people who can handle multiple layers of abstraction at the same time efficiently, but they seem to be a very rare specimen. No matter how skilled you are, there is always an additional price to pay for maintaining such hybrid solutions. This will either involve extra effort and time required to switch between C and Python, or additional stress that will make you eventually less efficient.

According to the TIOBE index, C is still one of the most popular programming languages. Despite this fact, it is very common for Python programmers to know very little or almost nothing about it. Personally, I think that C should be the *lingua franca* in the programming world, but my opinion is very unlikely to change anything in this matter.

Python also is so seductive and easy to learn, meaning that a lot of programmers forget about all their previous experiences and completely switch to the new technology. And programming is not like riding a bike. This particular skill erodes very fast if not used and polished sufficiently. Even programmers with a strong C background are risking gradually losing their previous C proficiency if they decide to dive into Python for too long.

All of the above leads to one simple conclusion—it is harder to find people who will be able to understand and extend your code. For open-source packages, this means fewer voluntary contributors. In closed source, this means that not all of your teammates will be able to develop and maintain extensions without breaking things. And debugging broken things is definitely harder in extensions than in plain Python code.

Harder debugging

When it comes to failures, the extensions may break very badly. One could think that static typing gives you a lot of advantages over Python and allows you to catch a lot of issues during the compilation step that would be hard to notice in Python. And that can happen even without a rigorous testing routine and full test coverage. But that's only one side of the coin.

On the other side, we have all the memory management that must be performed manually. And faulty memory management is the main reason for most programming errors in C. In the best-case scenario, such mistakes will result only in some memory leaks that will gradually eat all of your environment resources. The best case does not mean easy to handle. Memory leaks are really tricky to find without using proper external tools such as **Valgrind**. In most cases, the memory management issues in your extension code will result in a segmentation fault that is unrecoverable in Python and will cause the interpreter to crash without raising an exception that would explain the cause. This means that you will eventually need to arm up with additional tools that most Python programmers usually don't need to use. This adds complexity to your development environment and workflow.

The downsides of using extensions mean that they are not always the best tool to bridge Python with other languages. If the only thing you need to do is to interact with already built shared libraries, sometimes the best option is to use a completely different approach. The next section discusses ways of interacting with dynamic libraries without using extensions.

Interfacing with dynamic libraries without extensions

Thanks to `ctypes` (a module in the standard library) or `cffi` (an external package available on PyPI), you can integrate every compiled dynamic/shared library in Python, no matter what language it was written in. And you can do that in pure Python without any compilation step. Those two packages are known as **foreign function libraries**. They are interesting alternatives to writing your own extensions in C.

Although using foreign function libraries does not require writing C code, it does not mean you don't need to know anything about C to use them effectively. Both `ctypes` and `cffi` require from you a reasonable understanding of C and how dynamic libraries work in general. On the other hand, they remove the burden of dealing with Python reference counting and greatly reduce the risk of making painful mistakes. Also, interfacing with C code through `ctypes` or `cffi` is more portable than writing and compiling the C extension modules.

Let's first take a look at `ctypes`, which is a part of the Python standard library.

The ctypes module

The `ctypes` module is the most popular module to call functions from dynamic or shared libraries without the need to write custom C extensions. The reason for that is obvious. It is part of the standard library, so it is always available and does not require any external dependencies.

The first step to use code from a shared library is to load it. Let's see how to do that with `ctypes`.

Loading libraries

There are exactly four types of dynamic library loaders available in `ctypes` and two conventions to use them. The classes that represent dynamic and shared libraries are `ctypes.CDLL`, `ctypes.PyDLL`, `ctypes.OleDLL`, and `ctypes.WinDLL`. The differences between them are as follows:

- `ctypes.CDLL`: This class represents loaded shared libraries. The functions in these libraries use the standard calling convention and are assumed to return the `int` type. GIL is released during the call.

- `ctypes.PyDLL`: This class works like `ctypes.CDLL`, but GIL is not released during the call. After execution, the Python error flag is checked, and an exception is raised if the flag was set during the execution. It is only useful when the loaded library is directly calling functions from the Python/C API or uses callback functions that may be Python code.

- `ctypes.OleDLL`: This class is only available on Windows. Functions in these libraries use Windows' `stdcall` calling convention and return Windows-specific `HRESULT` code about call success or failure. Python will automatically raise an `OSError` exception after a result code indicating a failure.

- `ctypes.WinDLL`: This class is only available on Windows. Functions in these libraries use Windows' `stdcall` calling convention and return values of type `int` by default. Python does not automatically inspect whether these values indicate failure or not.

To load the library, you can either instantiate one of the preceding classes with proper arguments or call the `LoadLibrary()` function from the submodule associated with a specific class:

- `ctypes.cdll.LoadLibrary()` for `ctypes.CDLL`

- `ctypes.pydll.LoadLibrary()` for `ctypes.PyDLL`

- `ctypes.windll.LoadLibrary()` for `ctypes.WinDLL`

- `ctypes.oledll.LoadLibrary()` for `ctypes.OleDLL`

The main challenge when loading shared libraries is how to find them in a portable way. Different systems use different suffixes for shared libraries (`.dll` on Windows, `.dylib` on macOS, `.so` on Linux) and search for them in different places. The main offender in this area is Windows, which does not have a predefined naming scheme for libraries. Because of that, we won't discuss details of loading libraries with `ctypes` on this system and will concentrate mainly on Linux and macOS, which deal with this problem in a consistent and similar way.

> If you are interested in the Windows platform, refer to the official `ctypes` documentation, which has plenty of information about supporting that system. It can be found at `https://docs.python.org/3/library/ctypes.html`.

Both library loading conventions (the `LoadLibrary()` functions and specific library-type classes) require you to use the full library name. This means all the predefined library prefixes and suffixes need to be included. For example, to load the C standard library on Linux, you need to write the following:

```
>>> import ctypes
>>> ctypes.cdll.LoadLibrary('libc.so.6')
<CDLL 'libc.so.6', handle 7f0603e5f000 at 7f0603d4cbd0>
```

Here, for macOS, this would be the following:

```
>>> import ctypes
>>> ctypes.cdll.LoadLibrary('libc.dylib')
```

Fortunately, the `ctypes.util` submodule provides a `find_library()` function that allows you to load a library using its name without any prefixes or suffixes and will work on any system that has a predefined scheme for naming shared libraries:

```
>>> import ctypes
>>> from ctypes.util import find_library
>>> ctypes.cdll.LoadLibrary(find_library('c'))
<CDLL 'libc.so.6', handle 7f2e82f12000 at 0x7f2e8288e220>
>>> ctypes.cdll.LoadLibrary(find_library('bz2'))
<CDLL 'libbz2.so.1.0', handle 55fb3c2d1660 at 0x7f2e827e8af0>
```

So, if you are writing a `ctypes` package that is supposed to work both under macOS and Linux, always use `ctypes.util.find_library()`.

When your shared library is loaded, it is time to use its functions. Calling C functions using `ctypes` is explained in the next section.

Calling C functions using ctypes

When the dynamic/shared library is successfully loaded to the Python object, the common pattern is to store it as a module-level variable with the same name as the name of the loaded library. The functions can be accessed as object attributes, so calling them is like calling a Python function from any other imported module, for example:

```
>>> import ctypes
>>> from ctypes.util import find_library
>>> libc = ctypes.cdll.LoadLibrary(find_library('c'))
>>> libc.printf(b"Hello world!\n")
Hello world!
13
```

Unfortunately, all the built-in Python types except integers, strings, and bytes are incompatible with C datatypes and thus must be wrapped in the corresponding classes provided by the `ctypes` module. Here is the full list of compatible datatypes that come from the `ctypes` documentation:

ctypes **type**	C **type**	**Python type**
c_bool	_Bool	bool
c_char	char	1-character bytes
c_wchar	wchar_t	1-character string
c_byte	char	int
c_ubyte	unsigned char	int
c_short	short	int
c_ushort	unsigned short	int
c_int	int	int
c_uint	unsigned int	int
c_long	long	int
c_ulong	unsigned long	int
c_longlong	__int64 or long long	int
c_ulonglong	unsigned __int64 or long long	int
c_size_t	size_t	int
c_ssize_t	ssize_t or Py_ssize_t	int
c_float	float	float
c_double	double	float
c_longdouble	long double	float
c_char_p	char* (NULL-terminated)	bytes or None
c_wchar_p	wchar_t* (NULL-terminated)	string or None
c_void_p	void*	int or None

As you can see, the preceding table does not contain dedicated types that would reflect any of the Python collections as C arrays. The recommended way to create types for C arrays is to simply use the multiplication operator with the desired basic `ctypes` type as follows:

```
>>> import ctypes
>>> IntArray5 = ctypes.c_int * 5
>>> c_int_array = IntArray5(1, 2, 3, 4, 5)
>>> FloatArray2 = ctypes.c_float * 2
>>> c_float_array = FloatArray2(0, 3.14)
>>> c_float_array[1]
3.140000104904175
```

The above syntax works for every basic `ctypes` type.

Let's look at how Python functions are passed as C callbacks in the next section.

Passing Python functions as C callbacks

It is a very popular design pattern to delegate part of the work of function implementation to custom callbacks provided by the user. The most-known function from the C standard library that accepts such callbacks is a `qsort()` function that provides a generic implementation of the quicksort algorithm. It is rather unlikely that you would like to use this algorithm instead of the default `TimSort` implemented in the CPython interpreter, which is more suited for sorting Python collections. Anyway, `qsort()` seems to be a canonical example of an efficient sorting algorithm and a C API that uses the callback mechanism that is found in many programming books. This is why we will try to use it as an example of passing the Python function as a C callback.

The ordinary Python function type will not be compatible with the callback function type required by the `qsort()` specification. Here is the signature of `qsort()` from the BSD man page that also contains the type of accepted callback type (the `compar` argument):

```
void qsort(void *base, size_t nel, size_t width,
           int (*compar)(const void*, const void *));
```

So in order to execute `qsort()` from `libc`, you need to pass the following:

- `base`: This is the array that needs to be sorted as a `void*` pointer.
- `nel`: This is the number of elements as `size_t`.
- `width`: This is the size of the single element in the array as `size_t`.
- `compar`: This is the pointer to the function that is supposed to return `int` and accepts two `void*` pointers. It points to the function that compares the size of two elements that are being sorted.

We already know from the *Calling C functions using ctypes* section how to construct the C array from other `ctypes` types using the multiplication operator. `nel` should be `size_t` and that maps to Python `int`, so it does not require any additional wrapping and can be passed as `len(iterable)`. The `width` value can be obtained using the `ctypes.sizeof()` function once we know the type of our base array. The last thing we need to know is how to create the pointer to the Python function compatible with the `compar` argument.

The `ctypes` module contains a `CFUNCTYPE()` factory function that allows you to wrap Python functions and represent them as C callable function pointers. The first argument is the C return type that the wrapped function should return.

It is followed by the variable list of C types that the function accepts as the arguments. The function type compatible with the `compar` argument of `qsort()` will be as follows:

```python
CMPFUNC = ctypes.CFUNCTYPE(
    # return type
    ctypes.c_int,
    # first argument type
    ctypes.POINTER(ctypes.c_int),
    # second argument type
    ctypes.POINTER(ctypes.c_int),
)
```

> `CFUNCTYPE()` uses the `cdecl` calling convention, so it is compatible only with the `CDLL` and `PyDLL` shared libraries. The dynamic libraries on Windows that are loaded with `WinDLL` or `OleDLL` use the `stdcall` calling convention. This means that the other factory must be used to wrap Python functions as C callable function pointers. In `ctypes`, it is `WINFUNCTYPE()`.

To wrap everything up, let's assume that we want to sort a randomly shuffled list of integer numbers with a `qsort()` function from the standard C library. Here is the example script that shows how to do that using everything that we have learned about `ctypes` so far:

```python
from random import shuffle

import ctypes
from ctypes.util import import find_library

libc = ctypes.cdll.LoadLibrary(find_library('c'))

CMPFUNC = ctypes.CFUNCTYPE(
    # return type
    ctypes.c_int,
    # first argument type
    ctypes.POINTER(ctypes.c_int),
    # second argument type
    ctypes.POINTER(ctypes.c_int),
```

```python
)


def ctypes_int_compare(a, b):
    # arguments are pointers so we access using [0] index
    print(" %s cmp %s" % (a[0], b[0]))

    # according to qsort specification this should return:
    # * less than zero if a < b
    # * zero if a == b
    # * more than zero if a > b
    return a[0] - b[0]


def main():
    numbers = list(range(5))
    shuffle(numbers)
    print("shuffled: ", numbers)

    # create new type representing array with length
    # same as the length of numbers list
    NumbersArray = ctypes.c_int * len(numbers)
    # create new C array using a new type
    c_array = NumbersArray(*numbers)

    libc.qsort(
        # pointer to the sorted array
        c_array,
        # length of the array
        len(c_array),
        # size of single array element
        ctypes.sizeof(ctypes.c_int),
        # callback (pointer to the C comparison function)
        CMPFUNC(ctypes_int_compare)
    )
    print("sorted:   ", list(c_array))


if __name__ == "__main__":
    main()
```

The comparison function provided as a callback has an additional `print` statement, so we can see how it is being executed during the sorting process as follows:

```
$ python3 ctypes_qsort.py
shuffled:  [4, 3, 0, 1, 2]
 4 cmp 3
 4 cmp 0
 3 cmp 0
 4 cmp 1
 3 cmp 1
 0 cmp 1
 4 cmp 2
 3 cmp 2
 1 cmp 2
sorted:    [0, 1, 2, 3, 4]
```

Of course, using `qsort` in Python doesn't make a lot of sense because Python has its own specialized sorting algorithm. Anyway, passing Python functions as C callbacks is a very useful technique for integrating many third-party libraries.

The `ctypes` module is very popular among Python programmers because it is part of the standard library. Its downside is a lot of low-level type handling and a bit of boilerplate required to interact with loaded libraries. That's why some developers prefer using a third-party package, CFFI, that streamlines the usage of foreign function calls. We will take a look at it in the next section.

CFFI

CFFI is a foreign function interface for Python that is an interesting alternative to `ctypes`. It is not a part of the standard library, but it is easily available from PyPI as the `cffi` package. It is different from `ctypes` because it puts more emphasis on reusing plain C declarations instead of providing extensive Python APIs in a single module. It is way more complex and also has a feature that allows you to automatically compile some parts of your integration layer into extensions using the C compiler. This means it can be used as a hybrid solution that fills the gap between plain C extensions and `ctypes`.

Because it is a very large project, it is impossible to briefly introduce it in a few paragraphs. On the other hand, it would be a shame to not say something more about it. We have already discussed one example of integrating the `qsort()` function from the standard library using `ctypes`. So, the best way to show the main differences between these two solutions would be to reimplement the same example with `cffi`.

I hope that the following one block of code is worth more than a few paragraphs of text:

```python
from random import shuffle

from cffi import FFI

ffi = FFI()

ffi.cdef("""
void qsort(void *base, size_t nel, size_t width,
           int (*compar)(const void *, const void *));
""")
C = ffi.dlopen(None)


@ffi.callback("int(void*, void*)")
def cffi_int_compare(a, b):
    # Callback signature requires exact matching of types.
    # This involves less magic than in ctypes
    # but also makes you more specific and requires
    # explicit casting
    int_a = ffi.cast('int*', a)[0]
    int_b = ffi.cast('int*', b)[0]
    print(" %s cmp %s" % (int_a, int_b))

    # according to qsort specification this should return:
    # * less than zero if a < b
    # * zero if a == b
    # * more than zero if a > b
    return int_a - int_b


def main():
    numbers = list(range(5))
    shuffle(numbers)
    print("shuffled: ", numbers)

    c_array = ffi.new("int[]", numbers)

    C.qsort(
        # pointer to the sorted array
```

```
        c_array,
        # length of the array
        len(c_array),
        # size of single array element
        ffi.sizeof('int'),
        # callback (pointer to the C comparison function)
        cffi_int_compare,
    )
    print("sorted:   ", list(c_array))

if __name__ == "__main__":
    main()
```

The output will be similar to the one presented earlier when discussing the example of C callbacks in `ctypes`. Using CFFI to integrate `qsort` in Python doesn't make any more sense than using `ctypes` for the same purpose. Anyway, the preceding example should show the main differences between `ctypes` and `cffi` regarding handling datatypes and function callbacks.

Summary

This chapter explained one of the most complex topics in the book. We discussed the reasons and tools for building Python extensions as a way of bridging Python with other languages. We started by writing pure C extensions that depend only on the Python/C API and then reimplemented it with Cython to show how easy it can be if you only choose the proper tool.

There are still some reasons for doing things the hard way and using nothing more than the pure C compiler and the `Python.h` headers. Anyway, the best recommendation is to use tools such as Cython because this will make your code base more readable and maintainable. It will also save you from most of the issues caused by incautious reference counting and memory mismanagement.

Our discussion of extensions ended with the presentation of `ctypes` and CFFI as alternative ways to solve the problems of integrating shared libraries. Because they do not require writing custom extensions to call functions from compiled binaries, they should be your tools of choice for integrating closed-source dynamic/shared libraries—especially if you don't need to use custom C code.

In the last few chapters, we have discussed multiple complex topics. From advanced design patterns, through concurrency and event-driven programming to bridging Python with different languages. Now we will be moving on to the topic of maintaining Python applications: from testing and quality assurance to packaging, monitoring, and optimizing applications of any size.

One of the most important challenges of software maintenance is how to assure that the code we wrote is correct. As our software inevitably becomes more complex, it is harder to ensure that it is working properly without an organized testing regime. And as it will grow bigger, it will be impossible to effectively test it without any kind of automation. That's why in the next chapter, we will take a look at various Python tools and techniques that allow you to automate testing and quality processes.

10

Testing and Quality Automation

Software is complex. No matter what language you use, what frameworks you build on, and how elegant your coding style is, it is hard to verify software correctness just by reading the code. That's not only because non-trivial applications usually consist of large amounts of code. It is also because complete software is often composed of many layers and relies on many external or interchangeable components, such as operating systems, libraries, databases, caches, web APIs, or clients used to interact with your code (browsers, for instance).

The complexity of modern software means that the verification of its correctness often requires you to go beyond your code. You need to consider the environment in which your code runs, variations of components that can be replaced, and the ways your code can be interacted with. That's why developers of high-quality software often employ special testing techniques that allow them to quickly and reliably verify that the code they write meets desired acceptance criteria.

Another concern of complex software is its **maintainability**. This can be understood as how easy it is to sustain the ongoing development of a piece of software. And development is not only about implementing new features or enhancements but also diagnosing and fixing issues that will be inevitably discovered along the way. Maintainable software is software that requires little effort to introduce new changes and where there is a low risk of introducing new defects upon change.

As you can probably guess, maintainability is a product of many software aspects. Automated testing of course helps in reducing the risk of change by enforcing that known use cases are properly covered by existing and future code. But it is not enough to ensure that future changes will be easy to implement. That's why modern testing methodologies also rely on automated code quality measurement and testing to enforce specific coding conventions, highlight potentially erroneous code fragments, or scan for security vulnerabilities.

The modern testing landscape is vast. It is easy to get lost in a sea of testing methodologies, tools, frameworks, libraries, and utilities. That's why in this chapter we will review the most popular testing and quality automation techniques that are often employed by professional Python developers. This should give you a good overview of what's generally possible and also allow you to build your own testing routine. We will cover the following topics:

- The principles of test-driven development
- Writing tests with pytest
- Quality automation
- Mutation testing
- Useful testing utilities

We will use a lot of packages from PyPI, so let's start by considering the technical requirements for this chapter.

Technical requirements

The following are the Python packages that are mentioned in this chapter, which you can download from PyPI:

- `pytest`
- `redis`
- `coverage`
- `mypy`
- `mutmut`
- `faker`
- `freezegun`

Information on how to install packages is included in *Chapter 2*, *Modern Python Development Environments*.

The code files for this chapter can be found at `https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition/tree/main/Chapter%2010`.

The principles of test-driven development

Testing is one of the most important elements of the software development process. It is so important that there is even a software development methodology called **Test-Driven Development** (**TDD**). It advocates writing software requirements as tests as the first (and foremost) step in developing code.

The principle is simple: you focus on the tests first. Use them to describe the behavior of the software, verify it, and check for potential errors. Only when those tests are complete should you proceed with the actual implementation to satisfy the tests.

TDD, in its simplest form, is an iterative process that consists of the following steps:

1. **Write tests**: Tests should reflect the specification of a functionality or improvement that has not been implemented yet.

2. **Run tests**: At this stage all new tests should fail as the feature or improvement is not yet implemented.

3. **Write a minimal valid implementation**: The code should be dead simple but correct. It is OK if it does not look elegant or has performance issues. The main focus at this stage should be satisfying all the tests written in *step 1*. It is also easier to diagnose problems in code that is simple than in code that is optimized for performance.

4. **Run tests**: At this stage all tests should pass. That includes both new and preexisting tests. If any of them fail, the code should be revised until it satisfies the requirements.

5. **Hone and polish**: When all tests are satisfied, the code can be progressively refactored until it meets desired quality standards. This is the time for streamlining, refactoring, and sometimes obvious optimizations (if you have brute-forced your way through the problem). After each change, all tests should be rerun to ensure that no functionality was broken.

This simple process allows you to iteratively extend your application without worrying that new change will break some preexisting and tested functionality. It also helps to avoid premature optimization and guides you through the development with a series of simple and bite-sized steps.

TDD won't deliver the promised results without proper work hygiene. That's why it is important to follow some basic principles:

- **Keep the size of the tested unit small**: In TDD we often talk about units of code and unit tests. A unit of code is a simple autonomous piece of software that (preferably) should do only one thing, and a single unit test should exercise one function or method with a single set of arguments. This makes writing tests easier but also favors good development practices and patterns like the single responsibility principle and inversion of control (see *Chapter 5, Interfaces, Patterns, and Modularity*).

- **Keep tests small and focused**: It is almost always better to create many small and simple tests than one long and elaborate test. Every test should verify only one aspect/requirement of the intended functionality. Having granular tests allows for easier diagnosing of potential issues and better maintainability of the test suite. Small tests pinpoint problems better and are just easier to read.

- **Keep tests isolated and independent**: The success of one test should not rely on the specific order of execution of all tests within the test suite. If a test relies on a specific state of the execution environment, the test itself should ensure that all preconditions are satisfied. Similarly, any side effects of the test should be cleaned up after the execution. Those preparation and cleanup phases of every test are also known as *setup and teardown*.

These few principles will help you write tests that are easy to understand and maintain. And that's important because writing tests is just yet another activity that takes time and increases initial development cost. Still, when done right, this is an investment that pays off pretty quickly. A systematic and automated testing routine reduces the number of software defects that would otherwise reach the end users. It also provides a framework for the verification of known software bugs.

A rigorous testing routine and following a few basic principles is usually supported by a dedicated testing library or framework. Python programmers are really lucky because the Python standard library comes with two built-in modules created exactly for the purpose of automated tests. These are:

- `doctest`: A testing module for testing interactive code examples found in docstrings. It is a convenient way of merging documentation with tests. `doctest` is theoretically capable of handling unit tests but it is more often used to ensure that snippets of the code found in docstrings reflect the correct usage examples.

> You can read more about `doctest` in the official documentation found at `https://docs.python.org/3/library/doctest.html`.

- `unittest`: A full-fledged testing framework inspired by JUnit (a popular Java testing framework). It allows for the organization of tests into test cases and test suites and provides common ways for managing setup and teardown primitives. `unittest` comes with a built-in test runner that is able to discover test modules across the whole codebase and execute specific test selections.

> You can read more about `unittest` in the official documentation found at `https://docs.python.org/3/library/unittest.html`.

These two modules together can satisfy most of the testing needs of even the most demanding developers. Unfortunately, `doctest` concentrates on a very specific use case for tests (the testing of code examples) and `unittest` requires a rather large amount of boilerplate due to class-oriented test organization. Its runner also isn't as flexible as it could be. That's why many professional programmers prefer using one of the third-party frameworks available on PyPI.

One such framework is **pytest**. It is probably one of the best and most mature Python testing frameworks out there. It offers a more convenient way of organizing tests as flat modules with test functions (instead of classes) but is also compatible with the `unittest` class-based test hierarchy. It has also a truly superior test runner and comes with a multitude of optional extensions.

The above advantages of pytest are the reason why we are not going to discuss the details of `unittest` and `doctest` usage. They are still great and useful but pytest is almost always a better and more practical choice. That's why we are now going to discuss examples of writing tests using pytest as our framework of choice.

Writing tests with pytest

Now it's time to put the theory into practice. We already know the advantages of TDD, so we'll try to build something simple with the help of tests. We will discuss the anatomy of a typical test and then go over common testing techniques and tools that are often employed by professional Python programmers. All of that will be done with the help of the pytest testing framework.

In order to do that we will require some problems to solve. After all, testing starts at the very beginning of the software development life cycle—when the software requirements are defined. In many testing methodologies, tests are just a code-native way of describing software requirements in executable form.

It's hard to find a single convincing programming challenge that would allow for showcasing a variety of testing techniques and at the same time would fit into a book format. That's why we are going to discuss a few small and unrelated problems instead. We will also revisit some of the examples found in previous chapters of the book.

From a TDD perspective, writing tests for existing code is of course an unorthodox approach to testing as writing tests ideally should precede the implementation and not vice versa. But it is a known practice. For professional programmers it is not uncommon to inherit a piece of software that is poorly tested or has not been tested at all. In such a situation, if you want to test your software reliably, you will have to eventually do the missing work. In our case, writing some tests for preexisting code will also be an interesting opportunity to talk about the challenges of writing tests after the code.

The first example will be pretty simple. It will allow us to understand the basic anatomy of a test and how to use the pytest runner to discover and run tests. Our task will be to create a function that:

- Accepts an iterable of elements and a batch size
- Returns an iterable of sub-lists where every sub-list is a batch of consecutive elements from the source list. The order of elements should stay the same
- Each batch has the same size
- If the source list does not have enough elements to fill the last batch, that batch should be shorter but never empty

That would be a relatively small but useful function. It could for instance be used to process large streams of data without needing to load them fully into process memory. It could also be used for distributing chunks of work to process separate threads or process workers as we learned in *Chapter 6, Concurrency*.

Let's start by writing the stub of our function to find out what we are working with. It will be named `batches()` and will be hosted in a file called `batch.py`. The signature can be as follows:

```python
from typing import import Any, Iterable, List


def batches(
    iterable: Iterable[Any], batch_size: int
```

```
) -> Iterable[List[Any]]:
    pass
```

We haven't provided any implementation yet as this is something we will take care of once the tests are done. We can see typing annotations that constitute part of the contract between the function and the caller.

Once we have done this, we are able to import our function into the `test` module to write the tests. The common convention for naming test modules is `test_<module-name>.py`, where `<module-name>` is the name of the module whose contents we are going to test. Let's create a file named `test_batch.py`.

The first test will do a pretty common thing: provide input data to the function and compare the results. We will be using a plain literal list as input. The following is some example test code:

```python
from batch import batches

def test_batch_on_lists():
    assert list(batches([1, 2, 3, 4, 5, 6], 1)) == [
        [1], [2], [3], [4], [5], [6]
    ]
    assert list(batches([1, 2, 3, 4, 5, 6], 2)) == [
        [1, 2], [3, 4], [5, 6]
    ]
    assert list(batches([1, 2, 3, 4, 5, 6], 3)) == [
        [1, 2, 3], [4, 5, 6]
    ]
    assert list(batches([1, 2, 3, 4, 5, 6], 4)) == [
        [1, 2, 3, 4], [5, 6],
    ]
```

The `assert` statement is the preferred way in pytest to test for the pre- and post-conditions of tested code units. pytest is able to inspect such assertions, recognize their exceptions, and thanks to this, output detailed reports of the test failures in a readable form.

The above is a popular structure for tests for small utilities and is often just enough to ensure they work as intended. Still, it does not clearly reflect our requirements, so maybe it would be worth restructuring it a little bit.

The following is an example of two extra tests that more explicitly map to our predefined requirements:

```python
from itertools import chain

def test_batch_order():
    iterable = range(100)
    batch_size = 2

    output = batches(iterable, batch_size)

    assert list(chain.from_iterable(output)) == list(iterable)

def test_batch_sizes():
    iterable = range(100)
    batch_size = 2

    output = list(batches(iterable, batch_size))

    for batch in output[:-1]:
        assert len(batch) == batch_size
    assert len(output[-1]) <= batch_size
```

The `test_batch_order()` test ensures that the order of elements in batches is the same as in the source iterable. The `test_batch_sizes()` test ensures that all batches have the same size (with the exception of the last batch, which can be shorter).

We can also see a pattern unfolding in both tests. In fact, many tests follow a very common structure:

1. **Setup**: This is the step where the test data and all other prerequisites are prepared. In our case the setup consists of preparing `iterable` and `batch_size` arguments.

2. **Execution**: This is when the actual tested unit of code is put into use and the results are saved for later inspection. In our case it is a call to the `batches()` function.

3. **Validation**: In this step we verify that the specific requirement is met by inspecting the results of unit execution. In our case these are all the `assert` statements used to verify the saved output.

4. **Cleanup**: This is the step where all resources that could affect other tests are released or returned back to the state they were in before the setup step. We didn't acquire any such resources, so in our case this step can be skipped.

According to the testing process outlined in *the principles of test-driven development* section, at the moment our test should fail as we haven't provided any function implementation yet. Let's run the pytest runner and see how it goes:

```
$ pytest -v
```

The output we get may look as follows:

```
======================= test session starts =======================
platform darwin -- Python 3.9.2, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
-- .../Expert-Python-Programming-Fourth-Edition/.venv/bin/python
cachedir: .pytest_cache
rootdir: .../Expert-Python-Programming-Fourth-Edition/Chapter 10/01 -
Writing tests with pytest
collected 3 items

test_batch.py::test_batch_on_lists FAILED                    [ 33%]
test_batch.py::test_batch_order FAILED                       [ 66%]
test_batch.py::test_batch_sizes FAILED                       [100%]


============================ FAILURES =============================
_____ test_batch_on_lists _____

    def test_batch_on_lists():
>       assert list(batches([1, 2, 3, 4, 5, 6], 1)) == [
            [1], [2], [3], [4], [5], [6]
        ]
E       TypeError: 'NoneType' object is not iterable

test_batch.py:7: TypeError
_____ test_batch_order _____

    def test_batch_order():
        iterable = range(100)
        batch_size = 2

        output = batches(iterable, batch_size)

>       assert list(chain.from_iterable(output)) == list(iterable)
```

```
E        TypeError: 'NoneType' object is not iterable


test_batch.py:27: TypeError
_____ test_batch_sizes _____


    def test_batch_sizes():
        iterable = range(100)
        batch_size = 2

>       output = list(batches(iterable, batch_size))
E       TypeError: 'NoneType' object is not iterable


test_batch.py:34: TypeError
===================== short test summary info =====================
FAILED test_batch.py::test_batch_on_lists - TypeError: 'NoneType' ...
FAILED test_batch.py::test_batch_order - TypeError: 'NoneType' obj...
FAILED test_batch.py::test_batch_sizes - TypeError: 'NoneType' obj...
```

As we can see, the test run failed with three individual test failures and we've got a detailed report of what went wrong. We have the same failure in each test. `TypeError` says that the `NoneType` object is not iterable so it could not be converted to a list. This means that none of our three requirements have been met yet. That's understandable because the `batches()` function doesn't do anything meaningful yet.

Now it's time to satisfy those tests. The goal is to provide a minimal working implementation. That's why we won't do anything fancy and will provide a simple and naïve implementation based on lists. Let's take a look at our first iteration:

```python
from typing import Any, Iterable, List


def batches(
    iterable: Iterable[Any], batch_size: int
) -> Iterable[List[Any]]:
    results = []
    batch = []

    for item in iterable:
        batch.append(item)
        if len(batch) == batch_size:
            results.append(batch)
            batch = []

    if batch:
```

```
        results.append(batch)

    return results
```

The idea is simple. We create one list of results and traverse the input `iterable`, actively creating new batches as we go. When a batch is full, we add it to the list of results and start a new one. When we are done, we check if there is an outstanding batch and add it to the results as well. Then we return the results.

This is a pretty naïve implementation that may not work well with arbitrarily large results, but it should satisfy our tests. Let's run the `pytest` command to see if it works:

```
$ pytest -v
```

The test result should now be as follows:

```
======================= test session starts ========================
platform darwin -- Python 3.9.2, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
-- .../Expert-Python-Programming-Fourth-Edition/.venv/bin/python
cachedir: .pytest_cache
rootdir: .../Expert-Python-Programming-Fourth-Edition/Chapter 10/01 -
Writing tests with pytest
collected 3 items

test_batch.py::test_batch_on_lists PASSED                    [ 33%]
test_batch.py::test_batch_order PASSED                       [ 66%]
test_batch.py::test_batch_sizes PASSED                       [100%]


======================== 3 passed in 0.01s =========================
```

As we can see, all tests have passed successfully. This means that the `batches()` function satisfies the requirements specified by our tests. It doesn't mean the code is completely bug-free, but it gives us confidence that it works well within the area of conditions verified by the tests. The more tests we have and the more precise they are, the more confidence in the code correctness we have.

Our work is not over yet. We made a simple implementation and verified it works. Now we are ready to proceed to the step where we refine our code. One of the reasons for doing things this way is that it is easier to spot errors in tests when working with the simplest possible implementation of the code. Remember that tests are code too, so it is possible to make mistakes in tests as well.

If the implementation of a tested unit is simple and easy to understand, it will be easier to verify whether the tested code is wrong or there's a problem with the test itself.

The obvious problem with the first iteration of our `batches()` function is that it needs to store all the intermediate results in the `results` list variable. If the `iterable` argument is large enough (or even infinite), it will put a lot of stress on your application as it will have to load all the data into memory. A better way would be to convert that function into a generator that yields successive results. This can be done with only a little bit of tuning:

```python
def batches(
    iterable: Iterable[Any], batch_size: int
) -> Iterable[List[Any]]:
    batch = []
    for item in iterable:
        batch.append(item)

        if len(batch) == batch_size:
            yield batch
            batch = []

    if batch:
        yield batch
```

The other way around would be to make use of the iterators and the `itertools` module as in the following example:

```python
from itertools import islice

def batches(
    iterable: Iterable[Any], batch_size: int
) -> Iterable[List[Any]]:
    iterator = iter(iterable)

    while True:
        batch = list(islice(iterator, batch_size))

        if not batch:
            return

        yield batch
```

That's what is really great about the TDD approach. We are now able to easily experiment and tune existing function implementation with a reduced risk of breaking things. You can test it for yourself by replacing the `batches()` function implementation with one of those shown above and running the tests to see if it meets the defined requirements.

Our example problem was small and simple to understand and so our tests were easy to write. But not every unit of code will be like that. When testing larger or more complex parts of code, you will often need additional tools and techniques that allow you to write clean and readable tests. In the next few sections we will review common testing techniques often used by Python programmers and show how to implement them with the help of pytest. The first one will be test parameterization.

Test parameterization

Using a direct comparison of the received and expected function output is a common method for writing short unit tests. It allows you to write clear and condensed tests. That's why we used this method in our first `test_batch_on_lists()` test in the previous section.

One problem with that technique is that it breaks the classic pattern of setup, execution, verification, and cleanup stages. You can't see clearly what instructions prepare the test context, which function call constitutes unit execution, and which instructions perform result verification.

The other problem is that when the number of input-output data samples increases, tests become overly large. It is harder to read them, and potential independent failures are not properly isolated. Let's recall the code of the `test_batch_on_lists()` test to better understand this issue:

```python
def test_batch_on_lists():
    assert list(batches([1, 2, 3, 4, 5, 6], 1)) == [
        [1], [2], [3], [4], [5], [6]
    ]
    assert list(batches([1, 2, 3, 4, 5, 6], 2)) == [
        [1, 2], [3, 4], [5, 6]
    ]
    assert list(batches([1, 2, 3, 4, 5, 6], 3)) == [
        [1, 2, 3], [4, 5, 6]
    ]
    assert list(batches([1, 2, 3, 4, 5, 6], 4)) == [
        [1, 2, 3, 4], [5, 6],
    ]
```

Each `assert` statement is responsible for verifying one pair of input-output samples. But each pair can be constructed to verify different conditions of the initial requirements. In our case the first three statements could verify that the size of each output batch is the same. But the last `assert` verifies that the incomplete batch is also returned if the length of the `iterable` argument is not divisible by `batch_size`. The intention of the test isn't perfectly clear as it slightly breaks the "keep tests small and focused" principle.

We can slightly improve the test structure by moving the preparation of all the samples to the separate setup part of the test and then iterating over the samples in the main execution part. In our case this can be done with a simple dictionary literal:

```python
def test_batch_with_loop():
    iterable = [1, 2, 3, 4, 5, 6]
    samples = {
        # even batches
        1: [[1], [2], [3], [4], [5], [6]],
        2: [[1, 2], [3, 4], [5, 6]],
        3: [[1, 2, 3], [4, 5, 6]],
        # batches with rest
        4: [[1, 2, 3, 4], [5, 6]],
    }

    for batch_size, expected in samples.items():
        assert list(batches(iterable, batch_size)) == expected
```

> See how a small change to the test structure allowed us to annotate which samples are expected to verify a particular function requirement. We don't always have to use separate tests for every requirement. Remember: practicality beats purity.

Thanks to this change we have a more clear separation of the setup and execution parts of the test. We can now say that the execution of the `batch()` function is parameterized with the content of the `samples` dictionary. It is like running multiple small tests within a single test run.

Another problem with testing multiple samples within a single test function is that the test may break early. If the first `assert` statement fails, the test will stop immediately. We won't know whether subsequent `assert` statements would succeed or fail until we fix the first error and are able to proceed with further execution of the test. And having a full overview of all individual failures often allows us to better understand what's wrong with the tested code.

This problem cannot be easily solved in a loop-based test. Fortunately, pytest comes with native support for test parameterization in the form of the `@pytest.mark.parametrize` decorator. This allows us to move the parameterization of a test's execution step outside of the test body. pytest will be smart enough to treat each set of input parameters as a separate "virtual" test that will be run independently from other samples.

`@pytest.mark.parametrize` requires at least two positional arguments:

- `argnames`: This is a list of argument names that pytest will use to provide test parameters to the test function as arguments. It can be a comma-separated string or a list/tuple of strings.
- `argvalues`: This is an iterable of parameter sets for each individual test run. Usually, it is a list of lists or a tuple of tuples.

We could rewrite our last example to use the `@pytest.mark.parametrize` decorator as follows:

```python
import pytest

@pytest.mark.parametrize(
    "batch_size, expected", [
        # even batches
        [1, [[1], [2], [3], [4], [5], [6]]],
        [2, [[1, 2], [3, 4], [5, 6]]],
        [3, [[1, 2, 3], [4, 5, 6]]],
        # batches with rest
        [4, [[1, 2, 3, 4], [5, 6]]]
    ]
)
def test_batch_parameterized(batch_size, expected):
    iterable = [1, 2, 3, 4, 5, 6]
    assert list(batches(iterable, batch_size)) == expected
```

If we now execute all the tests that we've written so far with the `pytest -v` command, we will get the following output:

```
======================= test session starts =========================
platform darwin -- Python 3.9.2, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
-- .../Expert-Python-Programming-Fourth-Edition/.venv/bin/python
cachedir: .pytest_cache
rootdir: .../Expert-Python-Programming-Fourth-Edition/Chapter 10/01 -
Writing tests with pytest
collected 8 items
```

```
test_batch.py::test_batch_on_lists PASSED                  [ 12%]
test_batch.py::test_batch_with_loop PASSED                 [ 25%]
test_batch.py::test_batch_parameterized[1-expected0] PASSED   [ 37%]
test_batch.py::test_batch_parameterized[2-expected1] PASSED   [ 50%]
test_batch.py::test_batch_parameterized[3-expected2] PASSED   [ 62%]
test_batch.py::test_batch_parameterized[4-expected3] PASSED   [ 75%]
test_batch.py::test_batch_order PASSED                     [ 87%]
test_batch.py::test_batch_sizes PASSED                     [100%]


======================== 8 passed in 0.01s =========================
```

As you can see, the test report lists four separate instances of the `test_batch_parameterized()` test run. If any of those fails it won't affect the others.

Test parameterization effectively puts a part of classic test responsibility—the setup of the test context—outside of the test function. This allows for greater reusability of the test code and gives more focus on what really matters: unit execution and the verification of the execution outcome.

Another way of extracting the setup responsibility from the test body is through the use of reusable test fixtures. pytest already has great native support for reusable fixtures that is truly magical.

pytest's fixtures

The term "fixture" comes from mechanical and electronic engineering. It is a physical device that can take the form of a clamp or grip that holds the tested hardware in a fixed position and configuration (hence the name "fixture") to allow it to be consistently tested in a specific environment.

Software testing fixtures serve a similar purpose. They simulate a fixed environment configuration that tries to mimic the real usage of the tested software component. Fixtures can be anything from specific objects used as input arguments, through environment variable configurations, to sets of data stored in a remote database that are used during the testing procedure.

In pytest, a fixture is a reusable piece of setup and/or teardown code that can be provided as a dependency to the test functions. pytest has a built-in dependency injection mechanism that allows for writing modular and scalable test suites.

We've covered the topic of dependency injection for Python applications in *Chapter 5, Interfaces, Patterns, and Modularity*.

To create a pytest fixture you need to define a named function and decorate it with the `@pytest.fixture` decorator as in the following example:

```python
import pytest


@pytest.fixture
def dependency():
    return "fixture value"
```

pytest runs fixture functions before test execution. The return value of the fixture function (here `"fixture value"`) will be provided to the test function as an input argument. It is also possible to provide both setup and cleanup code in the same fixture function by using the following generator syntax:

```python
@pytest.fixture
def dependency_as_generator():
    # setup code
    yield "fixture value"
    # teardown code
```

When generator syntax is used, pytest will obtain the yielded value of the fixture function and keep it suspended until the test finishes its execution. After the test finishes, pytest will resume execution of all used fixture functions just after the `yield` statement regardless of the test result (failure or success). This allows for the convenient and reliable cleanup of the test environment.

To use a fixture within a test you need to use its name as an input argument of the test function:

```python
def test_fixture(dependency):
    pass
```

When starting a `pytest` runner, pytest will collect all fixture uses by inspecting the test function signatures and matching the names with available fixture functions. By default, there are a few ways that pytest will discover fixtures and perform their name resolution:

- **Local fixtures**: Tests are able to use all the fixtures that are available from the same module that they are defined in. These can be fixtures that are imported in the same module. Local fixtures always take precedence over **shared fixtures**.

- **Shared fixtures**: Tests are able to use fixtures available in the `conftest` module stored in the same directory as the test module or any of its parent directories. A test suite can have multiple `conftest` modules. Fixtures from `conftest` that are closer in the directory hierarchy take precedence over those that are further in the directory hierarchy. Shared fixtures always take precedence over **plugin fixtures**.

- **Plugin fixtures**: pytest plugins can provide their own fixtures. These fixture names will be matched last.

Last but not least, fixtures can be associated with specific scopes that decide the lifetime of fixture values. These scopes are extremely important for fixtures implemented as generators because they determine when cleanup code is executed. There are five scopes available:

- **"function" scope**: This is the default scope. A fixture function with the `"function"` scope will be executed once for every individual test run and will be destroyed afterward.

- **"class" scope**: This scope can be used for test methods written in the `xUnit` style (based on the `unittest` module). Fixtures with this scope are destroyed after the last test in a test class.

- **"module" scope**: Fixtures with this scope are destroyed after the last test in the test module.

- **"package" scope**: Fixtures with this scope are destroyed after the last test in the given test package (collection of test modules).

- **"session" scope**: This is kind of a global scope. Fixtures with this scope live though the entire runner execution and are destroyed after the last test.

Different scopes of fixtures can be used to optimize the test execution as a specific environment setup may sometimes take a substantial amount of time to execute. If many tests can safely reuse the same setup, it may be reasonable to expand the default `"function"` scope to `"module"`, `"package"`, or even `"session"`.

Moreover, `"session"` fixtures can be used to perform global setup for the whole test run as well as a global cleanup. That's why they are often used with the `autouse=True` flag, which marks a fixture as an automatic dependency for a given group of tests. The scoping of **autouse fixtures** is as follows:

- **Module-level for the test module fixture**: If a fixture with the `autouse` flag is included in the test module (a module with the `test` prefix), it will be automatically marked as a dependency of every test within that module.

- **Package-level for the test conftest module fixture**: If a fixture with the `autouse` flag is included in a `conftest` module of a given test directory, it will be automatically marked as a dependency of every test in every test module within the same directory. This also includes subdirectories.

The best way to learn about using fixtures in various forms is by example. Our tests for the `batch()` function from the previous section were pretty simple and so didn't require the extensive use of fixtures. Fixtures are especially useful if you need to provide some complex object initialization or the setup state of external software components like remote services or databases. In *Chapter 5*, *Interfaces, Patterns, and Modularity*, we discussed examples of code for tracking page view counts with pluggable storage backends, and one of those examples used Redis as a storage implementation. Testing those backends would be a perfect use case for pytest fixtures, so let's recall the common interface of the `ViewsStorageBackend` abstract base class:

```python
from abc import ABC, abstractmethod
from typing import Dict

class ViewsStorageBackend(ABC):
    @abstractmethod
    def increment(self, key: str): ...

    @abstractmethod
    def most_common(self, n: int) -> Dict[str, int]: ...
```

Abstract base classes or any other types of interface implementations, like `Protocol` subclasses, are actually great when it comes to testing. They allow you to focus on the class behavior instead of the implementation.

If we would like to test the behavior of any implementation of `ViewsStorageBackend`, we could test for a few things:

- If we receive an empty storage backend, the `most_common()` method will return an empty dictionary
- If we increment a number of page counts for various keys and request a number of most common keys greater or equal to the number of keys incremented, we will receive all tracked counts
- If we increment a number of page counts for various keys and request a number of most common keys greater than or equal to the number of keys incremented, we will receive a shortened set of the most common elements

We will start with tests and then go over actual fixture implementation. The first test function for the empty storage backend will be really simple:

```python
import pytest
import random
from interfaces import ViewsStorageBackend


@pytest.mark.parametrize(
    "n", [0] + random.sample(range(1, 101), 5)
)
def test_empty_backend(backend: ViewsStorageBackend, n: int):
    assert backend.most_common(n) == {}
```

This test doesn't require any elaborate setup. We could use a static set of n argument parameters, but additional parameterization with random values adds a nice touch to the test. The `backend` argument is a declaration of a fixture use that will be resolved by pytest during the test run.

The second test for obtaining a full set of increment counts will require more verbose setup and execution:

```python
def test_increments_all(backend: ViewsStorageBackend):
    increments = {
        "key_a": random.randint(1, 10),
        "key_b": random.randint(1, 10),
        "key_c": random.randint(1, 10),
    }

    for key, count in increments.items():
        for _ in range(count):
            backend.increment(key)

    assert backend.most_common(len(increments)) == increments
    assert backend.most_common(len(increments) + 1) == increments
```

The test starts with the declaration of a literal dictionary variable with the intended increments. This simple setup serves two purposes: the `increments` variable guides the further execution step and also serves as validation data for two verification assertions. As in the previous test, we expect the `backend` argument to be provided by a `pytest` fixture.

The last test is quite similar to the previous one:

```python
def test_increments_top(backend: ViewsStorageBackend):
    increments = {
        "key_a": random.randint(1, 10),
        "key_b": random.randint(1, 10),
        "key_c": random.randint(1, 10),
        "key_d": random.randint(1, 10),
    }

    for key, count in increments.items():
        for _ in range(count):
            backend.increment(key)

    assert len(backend.most_common(1)) == 1
    assert len(backend.most_common(2)) == 2
    assert len(backend.most_common(3)) == 3

    top2_values = backend.most_common(2).values()
    assert list(top2_values) == (
        sorted(increments.values(), reverse=True)[:2]
    )
```

The setup and execution steps are similar to the ones used in the `test_increments_all()` test function. If we weren't writing tests, we would probably consider moving those steps to separate reusable functions. But here it would probably have a negative impact on readability. Tests should be kept independent, so a bit of redundancy often doesn't hurt if it allows for clear and explicit tests. However, this is not a rule of course and always requires personal judgment.

Since all the tests are written down, it is time to provide a fixture. In *Chapter 5, Interfaces, Patterns, and Modularity*, we've included two implementations of backends: `CounterBackend` and `RedisBackend`. Ultimately, we would like to use the same set of tests for both storage backends. We will get to that eventually, but for now let's pretend that there's only one backend. It will simplify things a little bit.

Let's assume for now that we are testing only `RedisBackend`. It is definitely more complex than `CounterBackend` so we will have more fun doing that. We could write just one backend fixture but `pytest` allows us to have modular fixtures, so let's see how that works. We will start with the following:

```python
from redis import Redis
from backends import RedisBackend

@pytest.fixture
def backend(redis_client: Redis):
    set_name = "test-page-counts"
    redis_client.delete(set_name)

    return RedisBackend(
        redis_client=redis_client,
        set_name=set_name
    )
```

`redis_client.delete(set_name)` removes the key in the Redis data store if it exists. We will use the same key in the `RedisBackend` initialization. The underlying Redis key that stores all of our increments will be created on the first storage modification, so we don't need to worry about non-existing keys. This way we ensure that every time a fixture is initialized, the storage backend is completely empty. The default fixture session scope is `"function"`, and that means every test using that fixture will receive an empty backend.

> Redis is not a part of most system distributions so you will probably have to install it on your own. Most Linux distributions have it available under the `redis-server` package name in their package repositories. You can also use Docker and Docker Compose. The following is a short `docker-compose.yml` file that will allow you to quickly start it locally:
>
> ```yaml
> version: "3.7"
> services:
>   redis:
>     image: redis
>     ports:
>       - 6379:6379
> ```
>
> You can find more details about using Docker and Docker Compose in *Chapter 2*, *Modern Python Development Environments*.

You may have noticed that we didn't instantiate the Redis client in the `backend()` fixture and instead specified it as an input argument of the fixture functions.

The dependency injection mechanism in pytest also covers fixture functions. This means you can request other fixtures inside of a fixture.

The following is example of a `redis_client()` fixture:

```python
from redis import Redis


@pytest.fixture(scope="session")
def redis_client():
    return Redis(host='localhost', port=6379)
```

To avoid over-complicating things we have just hardcoded the values for the Redis `host` and `port` arguments. Thanks to the above modularity it will be easier to replace those values globally if you ever decide to use a remote address instead.

Save all the tests in the `test_backends.py` module, start the Redis server locally, and execute the pytest runner using the `pytest -v` command. You will get output that may look as follows:

```
======================= test session starts =========================
platform darwin -- Python 3.9.2, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
-- .../Expert-Python-Programming-Fourth-Edition/.venv/bin/python
cachedir: .pytest_cache
rootdir: .../Expert-Python-Programming-Fourth-Edition/Chapter 10/03 -
Pytest's fixtures
collected 8 items

test_backends.py::test_empty_backend[0] PASSED                  [ 12%]
test_backends.py::test_empty_backend[610] PASSED               [ 25%]
test_backends.py::test_empty_backend[611] PASSED               [ 37%]
test_backends.py::test_empty_backend[7] PASSED                  [ 50%]
test_backends.py::test_empty_backend[13] PASSED                 [ 62%]
test_backends.py::test_empty_backend[60] PASSED                 [ 75%]
test_backends.py::test_increments_all PASSED                    [ 87%]
test_backends.py::test_increments_top PASSED                    [100%]


======================= 8 passed in 0.08s =========================
```

All tests passing means that we have succeeded in verifying the `RedisBackend` implementation. It would be great if we could do the same for `CounterBackend`. The most naïve thing to do would be to copy the tests and rewrite the test fixtures to now provide a new implementation of the backend. But this is a repetition that we would like to avoid.

We know that tests should be kept independent. Still, our three tests referenced only the `ViewsStorageBackend` abstract base class. So they should always be the same regardless of the actual implementation of the tested storage backends. What we have to do is to find a way to define a parameterized fixture that will allow us to repeat the same test over various backend implementations.

The parameterization of the fixture functions is a bit different than the parameterization of the test functions. The `@pytest.fixture` decorator accepts an optional `params` keyword value that accepts an iterable of fixture parameters. A fixture with the `params` keyword must also declare the use of a special built-in `request` fixture that, among other things, allows access to the current fixture parameter:

```python
import pytest

@pytest.fixture(params=[param1, param2, ...])
def parmetrized_fixture(request: pytest.FixtureRequest):
    return request.param
```

We can use the parameterized fixture and the `request.getfixturevalue()` method to dynamically load a fixture depending on a fixture parameter. The revised and complete set of fixtures for our test functions can now look as follows:

```python
import pytest
from redis import Redis
from backends import RedisBackend, CounterBackend

@pytest.fixture
def counter_backend():
    return CounterBackend()

@pytest.fixture(scope="session")
def redis_client():
    return Redis(host='localhost', port=6379)

@pytest.fixture
def redis_backend(redis_client: Redis):
    set_name = "test-page-counts"
    redis_client.delete(set_name)

    return RedisBackend(
        redis_client=redis_client,
        set_name=set_name
```

```python
    )

@pytest.fixture(params=["redis_backend", "counter_backend"])
def backend(request):
    return request.getfixturevalue(request.param)
```

If you now run the same test suite with a new set of fixtures, you will see that the amount of executed tests just doubled. The following is some example output of the `pytest -v` command:

```
======================= test session starts ========================
platform darwin -- Python 3.9.2, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
-- .../Expert-Python-Programming-Fourth-Edition/.venv/bin/python
cachedir: .pytest_cache
rootdir: .../Expert-Python-Programming-Fourth-Edition/Chapter 10/03 -
Pytest's fixtures
collected 16 items

test_backends.py::test_empty_backend[redis_backend-0] PASSED     [  6%]
test_backends.py::test_empty_backend[redis_backend-72] PASSED    [ 12%]
test_backends.py::test_empty_backend[redis_backend-23] PASSED    [ 18%]
test_backends.py::test_empty_backend[redis_backend-48] PASSED    [ 25%]
test_backends.py::test_empty_backend[redis_backend-780] PASSED   [ 31%]
test_backends.py::test_empty_backend[redis_backend-781] PASSED   [ 37%]
test_backends.py::test_empty_backend[counter_backend-0] PASSED   [ 43%]
test_backends.py::test_empty_backend[counter_backend-72] PASSED  [ 50%]
test_backends.py::test_empty_backend[counter_backend-23] PASSED  [ 56%]
test_backends.py::test_empty_backend[counter_backend-48] PASSED  [ 62%]
test_backends.py::test_empty_backend[counter_backend-780] PASSED [ 68%]
test_backends.py::test_empty_backend[counter_backend-781] PASSED [ 75%]
test_backends.py::test_increments_all[redis_backend] PASSED      [ 81%]
test_backends.py::test_increments_all[counter_backend] PASSED    [ 87%]
test_backends.py::test_increments_top[redis_backend] PASSED      [ 93%]
test_backends.py::test_increments_top[counter_backend] PASSED    [100%]

======================= 16 passed in 0.08s ========================
```

Thanks to the clever use of fixtures, we have reduced the amount of testing code without impacting the test readability. We could also reuse the same test functions to verify classes that should have the same behavior but different implementations. So, whenever requirements change, we can be sure that we will be able to recognize differences between classes of the same interface.

> You need to be cautious when designing your fixtures as the overuse of dependency injection can make understanding the whole test suite harder. Fixture functions should be kept simple and well documented.

Using fixtures to provide connectivity for external services like Redis is convenient because the installation of Redis is pretty simple and does not require any custom configuration to use it for testing purposes. But sometimes your code will be using a remote service or resource that you cannot easily provide in the testing environment or cannot perform tests against without making destructive changes. This can be pretty common when working with third-party web APIs, hardware, or closed libraries/binaries, for instance. In such cases, a common technique is to use fake objects or mocks that can substitute for real objects. We will discuss this technique in the next section.

Using fakes

Writing unit tests presupposes that you can isolate the unit of code that is being tested. Tests usually feed the function or method with some data and verify its return value and/or the side effects of its execution. This is mainly to make sure that:

- Tests are concerned with an atomic part of the application, which can be a function, method, class, or interface
- Tests provide deterministic, reproducible results

Sometimes, the proper isolation of the program component is not obvious or easily done. In the previous section we discussed the example of a testing suite that, among other things, was verifying a piece of code that interacted with a Redis data store. We provided the connectivity to Redis using a pytest fixture and we saw that it wasn't that hard. But did we test only our code, or did we test also the behavior of Redis?

In this particular case, including connectivity to Redis was a pragmatic choice. Our code did only a bit of work and left most of the heavy lifting to the external storage engine. It couldn't work properly if Redis didn't work properly. In order to test the whole solution, we had to test the integration of our code and the Redis data store. Tests like that are often called **integration tests** and are commonly used in testing software that heavily relies on external components.

But safe integration tests are not always possible. Not every service you will use will be as easy to start locally as Redis. Sometimes you will be dealing with those "special" components that cannot be replicated outside of ordinary production use.

In such cases, you will have to substitute the dependency with a **fake** object that simulates a real-life component.

To better understand the typical use cases for using fakes in tests, let's consider the following imaginary story: We are building a scalable application that provides our customers with the ability to track page counts on their sites in real time. Unlike our competitors, we offer a highly available and scalable solution with very low latency and the ability to run with consistent results in many datacenters across the globe. The cornerstone of our product is a small counter class from the `backends.py` module.

Having a highly available distributed hash map (the data type we used in Redis) that would ensure low latency in a multi-region setup isn't something trivial. Surely one Redis instance won't do what we advertise to our customers. Thankfully, a cloud computing vendor—ACME Corp—reached out to us recently, offering one of their latest beta products. It is called ACME Global HashMap Service and it does exactly what we want. But there's a catch: it is still in beta, and thus ACME Corp by their policy does not provide a sandbox environment that we can use for testing purposes yet. Also, for some unclear legal reasons, we can't use the production service endpoint in our automated testing pipelines.

So, what could we do? Our code grows every day. The planned `AcmeStorageBackend` class will likely have the additional code that handles logging, telemetry, access control, and a lot of other fancy stuff. We definitely want to be able to test it thoroughly. Therefore, we've decided to use a fake substitute of the ACME Corp SDK that we were supposed to integrate into our product.

The ACME Corp Python SDK comes in the form of the `acme_sdk` package. Among other things, it includes the following two interfaces:

```python
from typing import Dict

class AcmeSession:
    def __init__(self, tenant: str, token: str): ...

class AcmeHashMap:
    def __init__(self, acme_session: AcmeSession): ...

    def incr(self, key: str, amount):
        """Increments any key by specific amount"""
        ...

    def atomic_incr(self, key: str, amount):
        """Increments any key by specific amount atomically"""
        ...
```

```python
def top_keys(self, count: int) -> Dict[str, int]:
    """Returns keys with top values"""
    ...
```

The `AcmeSession` session is an object that encapsulates the connection to ACME Corp services, and `AcmeHashMap` is the service client we want to use. We will most likely use the `atomic_incr()` method to increment page view counts. `top_keys()` will provide us with the ability to obtain the most common pages.

To build a fake, we simply have to define a new class that has an interface that is compatible with our use of `AcmeHashMap`. We can take the pragmatic approach and implement only those classes and methods that we plan to use. The minimal implementation of `AcmeHashMapFake` could be as follows:

```python
from collections import Counter
from typing import Dict

class AcmeHashMapFake:
    def __init__(self):
        self._counter = Counter()

    def atomic_incr(self, key: str, amount):
        self._counter[key] += amount

    def top_keys(self, count: int) -> Dict[str, int]:
        return dict(self._counter.most_common(count))
```

We can use `AcmeHashMapFake` to provide a new fixture in the existing test suite for our storage backends. Let's assume that we have an `AcmeBackend` class in the `backends` module that takes the `AcmeHashMapFake` instance as its only input argument. We could then provide the following two pytest fixture functions:

```python
from backends import AcmeBackend
from acme_fakes import AcmeHashMapFake

@pytest.fixture
def acme_client():
    return AcmeHashMapFake()

@pytest.fixture
def acme_backend(acme_client):
    return AcmeBackend(acme_client)
```

Splitting the setup into two fixtures prepares us for what may come in the near future. When we finally get our hands on the ACME Corp sandbox environment, we will have to modify only one fixture:

```python
from acme_sdk import AcmeHashMap, AcmeSession

@pytest.fixture
def acme_client():
    return AcmeHashMap(AcmeSession(..., ...))
```

To summarize, fakes provide the equivalent behavior for an object that we can't build during a test or that we simply don't want to build. This is especially useful for situations where you have to communicate with external services or access remote resources. By internalizing those resources, you gain better control of the testing environment and thus are able to better isolate the tested unit of code.

Building custom fakes can become a tedious task if you have to build many of them. Fortunately, the Python library comes with the `unittest.mock` module, which can be used to automate the creation of fake objects.

Mocks and the unittest.mock module

Mock objects are generic fake objects that can be used to isolate the tested code. They automate the building process of the fake object's input and output. There is a greater level of use of mock objects in statically typed languages, where monkey patching is harder, but they are still useful in Python to shorten the code that mimics external APIs.

There are a lot of mock libraries available in Python, but the most recognized one is `unittest.mock`, which is provided in the standard library.

> `unittest.mock` was initially created as a third-party mock package available on PyPI. After some time, it was included in the standard library as a provisional package. To learn more about provisional standard library packages, visit `https://docs.python.org/dev/glossary.html#term-provisional-api`.

Mocks can almost always be used in place of custom fake objects. They are especially useful for faking external components and resources that we don't have full control over during the test. They are also an indispensable utility when we have to go against the prime TDD principle—that is, when we have to write tests *after* the implementation has been written.

We already discussed the example of faking the connectivity layer to the external resource in the previous section. Now we will take a closer look at a situation when we have to write a test for an already existing piece of code that doesn't have any tests yet.

Let's say we have the following `send()` function that is supposed to send email messages over the SMTP protocol:

```python
import smtplib
import email.message


def send(
    sender, to,
    subject='None',
    body='None',
    server='localhost'
):
    """sends a message."""
    message = email.message.Message()
    message['To'] = to
    message['From'] = sender
    message['Subject'] = subject
    message.set_payload(body)

    client = smtplib.SMTP(server)
    try:
        return client.sendmail(sender, to, message.as_string())
    finally:
        client.quit()
```

It definitely doesn't help that the function creates its own `smtplib.SMTP` instance, which clearly represents an SMTP client connection. If we started with tests first, we would probably have thought of it in advance and utilized a minor inversion of control to provide the SMTP client as a function argument. But the damage is done. The `send()` function is used across our whole codebase and we don't want to start refactoring yet. We need to test it first.

The `send()` function is stored in a `mailer` module. We will start with a black-box approach and assume it doesn't need any setup. We create a test that naively tries to call the function and hope for success. Our first iteration will be as follows:

```python
from mailer import send

def test_send():
```

```
    res = send(
        'john.doe@example.com',
        'john.doe@example.com',
        'topic',
        'body'
    )
    assert res == {}
```

Unless you have an SMTP server running locally you will see the following output when running pytest:

```
$ py.test -v --tb line
====================== test session starts =======================
platform darwin -- Python 3.9.2, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
-- .../Expert-Python-Programming-Fourth-Edition/.venv/bin/python
cachedir: .pytest_cache
pytest-mutagen-1.3 : Mutations disabled
rootdir: .../Expert-Python-Programming-Fourth-Edition/Chapter 10/05 -
Mocks and unittest.mock module
plugins: mutagen-1.3
collected 1 item

test_mailer.py::test_send FAILED                          [100%]


============================ FAILURES ============================
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/socket.
py:831: ConnectionRefusedError: [Errno 61] Connection refused
====================== short test summary info ======================
FAILED test_mailer.py::test_send - ConnectionRefusedError: [Errno 61...
======================= 1 failed in 0.05s =========================
```

> The `--tb` parameter of the `py.test` command can be used to control the length of the traceback output on test failures. Here we used `--tb line` to receive one-line tracebacks. Other values are `auto`, `long`, `short`, `native`, and `no`.

There go our hopes. The `send` function failed with a `ConnectionRefusedError` exception. If we don't want to run the SMTP server locally or send real messages by connecting to a real SMTP server, we will have to find a way to substitute the `smtplib.SMTP` implementation with a fake object.

In order to achieve our goal, we will use two techniques:

- **Monkey patching**: We will modify the `smtplib` module on the fly during the test run in order to trick the `send()` function into using a fake object in place of the `smtplib.SMTP` class.
- **Object mocking**: We will create a universal mock object that can act as a fake for absolutely any object. We will do that just to streamline our work.

Before we explain both techniques in more detail, let's take a look at an example test function:

```python
from unittest.mock import patch
from mailer import send

def test_send():
    sender = "john.doe@example.com"
    to = "jane.doe@example.com"
    body = "Hello jane!"
    subject = "How are you?"

    with patch('smtplib.SMTP') as mock:
        client = mock.return_value
        client.sendmail.return_value = {}

        res = send(sender, to, subject, body)

        assert client.sendmail.called
        assert client.sendmail.call_args[0][0] == sender
        assert client.sendmail.call_args[0][1] == to
        assert subject in client.sendmail.call_args[0][2]
        assert body in client.sendmail.call_args[0][2]
        assert res == {}
```

The `unittest.mock.path` context manager creates a new `unittest.mock.Mock` class instance and substitutes it under a specific import path. When the `send()` function will try to access the `smtplib.SMTP` attribute, it will receive the mock instance instead of the `SMTP` class object.

Mocks are quite magical. If you try to access any attribute of a mock outside of the set of names reserved by the `unittest.mock` module, it will return a new mock instance. Mocks can also be used as functions and, when called, also return a new mock instance.

The `send()` function expects `stmptlib.SMTP` to be a type object, so it will use the `SMTP()` call to obtain an instance of the SMTP client object. We use `mock.return_value` (`return_value` is one of the reserved names) to obtain the mock of that client object and control the return value of the `client.sendmail()` method.

After the execution of the `send()` function, we used a couple of other reserved names (`called` and `call_args`) to verify whether the `client.sendmail()` method was called and to inspect the call arguments.

> Note that what we did here probably isn't a good idea, as we just retraced what the `send()` function's implementation does. You should avoid doing so in your own tests because there's no purpose in a test that just paraphrases the implementation of the tested function. Regardless, this was more to present the abilities of the `unittest.mock` module than to show how tests should be written.

The `patch()` context manager from the `unittest.mock` module is one way of dynamically monkey patching import paths during the test run. It can be also used as a decorator. It is quite an intricate feature, so it is not always easy to patch what you want. Also, if you want to patch several objects at once it will require a bit of nesting, and that may be quite inconvenient.

pytest comes with an alternative way to perform monkey patching. It comes with a built-in `monkeypatch` fixture that acts as a patching proxy. If we would like to rewrite the previous example with the use of the `monkeypatch` fixture, we could do the following:

```python
import smtplib
from unittest.mock import Mock
from mailer import send

def test_send(monkeypatch):
    sender = "john.doe@example.com"
    to = "jane.doe@example.com"
    body = "Hello jane!"
    subject = "How are you?"

    smtp = Mock()
    monkeypatch.setattr(smtplib, "SMTP", smtp)
    client = smtp.return_value
    client.sendmail.return_value = {}
```

```
    res = send(sender, to, subject, body)

    assert client.sendmail.called
    assert client.sendmail.call_args[0][0] == sender
    assert client.sendmail.call_args[0][1] == to
    assert subject in client.sendmail.call_args[0][2]
    assert body in client.sendmail.call_args[0][2]
    assert res == {}
```

The monkey patching and mocks can be easily abused. That happens especially often when writing tests after the implementation. That's why mocks and monkey patching should be avoided if there are other ways of reliably testing software. Otherwise, you may end up with a project that has a lot of tests that are just empty shells and do not really verify software correctness. And there is nothing more dangerous than a false sense of security. There is also a risk that your mocks will assume behavior that is different from reality.

In the next section we will discuss the topic of quality automation, which dovetails with TDD.

Quality automation

There is no arbitrary scale that can say definitely if some code's quality is bad or good. Unfortunately, the abstract concept of code quality cannot be measured and expressed in the form of numbers. Instead, we can measure various metrics of the software that are known to be highly correlated with the quality of code. The following are a few:

- The percentage of code covered by tests
- The number of code style violations
- The amount of documentation
- Complexity metrics, such as McCabe's cyclomatic complexity
- The number of static code analysis warnings

Many projects use code quality testing in their continuous integration workflows. A good and popular approach is to test at least the basic metrics (test coverage, static code analysis, and code style violations) and not allow the merging of any code to the main branch that scores poorly on these metrics.

In the following sections, we will discuss some interesting tools and methods that will allow you to automate the evaluation of select code quality metrics.

Test coverage

Test coverage, also known as **code coverage**, is a very useful metric that provides objective information on how well some given source code is tested. It is simply a measurement of how many and which lines of code are executed during the test run. It is often expressed as a percentage, and 100% coverage means that every line of code was executed during tests.

The most popular code coverage tool for measuring Python code is the `coverage` package, and it is freely available on PyPI. Its usage is very simple and consists only of two steps:

1. Running the test suite using the `coverage` tool
2. Reporting the `coverage` report in the desired format

The first step is to execute the `coverage run` command in your shell with the path to your script/program that runs all the tests. For pytest, it could be something like this:

```
$ coverage run $(which pytest)
```

> The `which` command is a useful shell utility that returns in standard output the path to the executable of the other command. The `$()` expression can be used in many shells as a subexpression to substitute the command output into the given shell statement as a value.

Another way to invoke `coverage run` is using the `-m` flag, which specifies the runnable module. This is similar to invoking runnable modules with `python -m`. Both the `pytest` package and the `unittest` module provide their test runners as runnable modules:

```
$ python -m pytest
$ python -m unittest
```

So, in order to run test suites under the supervision of the `coverage` tool, you can use the following shell commands:

```
$ coverage run -m pytest
$ coverage run -m unittest
```

By default, the `coverage` tool will measure the test coverage of every module imported during the test run. It may thus also include external packages installed in a virtual environment for your project. You usually want to only measure the test coverage of the source code of your own project and exclude external sources. The `coverage` command accepts the `--source` parameter, which allows you to restrict the measurement to specific paths as in the following example:

```
$ coverage run --source . -m pytest
```

> The `coverage` tool allows you to specify some configuration flags in the `setup.cfg` file. The example contents of `setup.cfg` for the above `coverage` run invocation would be as follows:
>
> ```
> [coverage:run]
> source =
>     .
> ```

During the test run, the `coverage` tool will create a `.coverage` file with the intermediate results of the coverage measurement. After the run you can review the results by issuing the `coverage report` command.

To see how coverage measurement works in practice, let's say that we decided to do an ad hoc extension of one of the classes mentioned in the *pytest's fixtures* section but we didn't bother to test it properly. We would add the `count_keys()` method to `CounterClass` as in the following example:

```python
class CounterBackend(ViewsStorageBackend):
    def __init__(self):
        self._counter = Counter()

    def increment(self, key: str):
        self._counter[key] += 1

    def most_common(self, n: int) -> Dict[str, int]:
        return dict(self._counter.most_common(n))

    def count_keys(self):
        return len(self._counter)
```

This `count_keys()` method hasn't been included in our interface declaration (the `ViewsStorageBackend` abstract base class), so we didn't anticipate its existence when writing our test suite.

Let's now perform a quick test run using the `coverage` tool and review the overall results. This is the example shell transcript showing what we could potentially see:

```
$ coverage run –source . -m pytest -q
................

[100%]
16 passed in 0.12s
$ coverage report -m
Name              Stmts   Miss  Cover   Missing
-----------------------------------------------
backends.py          21      1    95%   19
interfaces.py         7      0   100%
test_backends.py     39      0   100%
-----------------------------------------------
TOTAL                67      1    99%
```

> All parameters and flags after the `-m <module>` parameter in the `coverage run` command will be passed directly to the runnable module invocation. Here the `-q` flag is a pytest runner flag saying that we want to obtain a short (quiet) report of the test run.

As we can see, all the tests have passed but the coverage report showed that the `backends.py` module is `95%` covered by tests. This means that `5%` of lines haven't been executed during the test run. This highlights that there is a gap in our test suite.

The `Missing` column (thanks to the `-m` flag of the `coverage report` command) shows the numbers of lines that were missed during the test run. For small modules with high coverage, it is enough just to locate missing coverage gaps. When coverage is very low, you will probably want a more in-depth report.

The `coverage` tool comes with a `coverage html` command that will generate an interactive coverage report in HTML format:
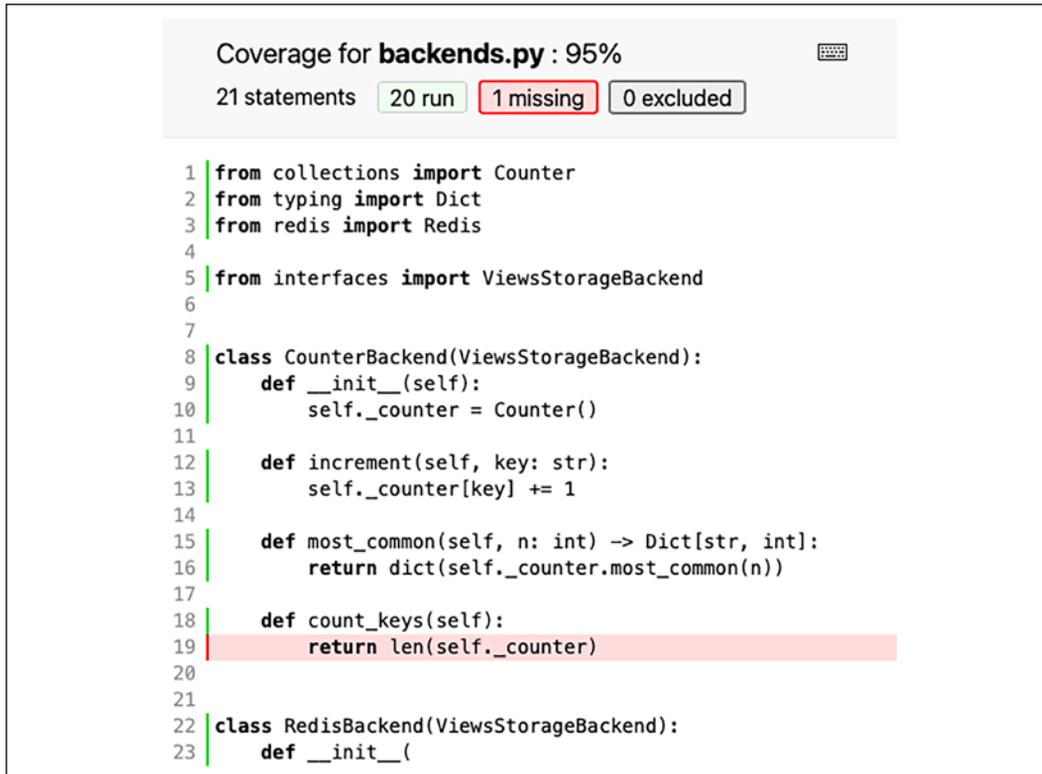
Figure 10.1: Example HTML coverage report highlighting coverage gaps

Test coverage is a very good metric that has high correlation with overall quality of code. Projects with low test coverage will statistically have more quality problems and defects. Projects with high coverage will usually have fewer defects and quality problems, assuming that tests are written according to the good practices highlighted in *the principles of test-driven development* section.

> Even projects with 100% coverage can behave unpredictably and be riddled with notorious bugs. In such situations, it may be necessary to use techniques that could validate the usefulness of existing test suites and uncover missed testing conditions. One such technique is mutation testing, discussed in the *Mutation testing* section.

Still, it is very easy to write meaningless tests that greatly increase test coverage. Always review the test coverage results of new projects with great care and don't treat the results as a definitive statement of project code quality.

Also, software quality is not only about how precisely software is tested but also about how easy it is to read, maintain, and extend. So, it is also about code style, common conventions, code reuse, and safety. Thankfully, measurement and validation of those programming areas can be automated to some extent.

> In this section we have used the `coverage` tool in a "classic" way. If you use pytest you can streamline the coverage measurement using the `pytest-cov` plugin, which can automatically add a `coverage` run to every test run. You can read more about `pytest-cov` at `https://github.com/pytest-dev/pytest-cov`.

Let's start with code style automation and linters, as those are the most common examples of quality automation used by professional Python programmers.

Style fixers and code linters

Code is simply harder to read than it is to write. That's true regardless of the programming language. It is less likely for a piece of software to be high quality if it is written inconsistently or with bizarre formatting or coding conventions. This is not only because it will be hard to read and understand but also because it will be hard to extend and maintain at a constant pace, and software quality concerns both the present state of code and its possible future.

To increase consistency across the codebase, programmers use tools that can verify the code style and various coding conventions. These tools are known as **linters**. Some such tools can also search for seemingly harmless but potentially problematic constructs, which include the following:

- Unused variables or `import` statements
- Access to protected attributes from outside of the hosting class
- Redefining existing functions
- Unsafe use of global variables
- Invalid order of `except` clauses
- Raising bad (non-exception) types

Although **linter** is an umbrella term used for any tool that flags style errors/ inconsistencies as well as suspicious and potentially dangerous code constructs, in recent years we have seen the ongoing specialization of linters. There are two main groups of linters commonly found in the Python community:

- **Style fixers**: These are linters that focus on coding conventions and enforce specific styling guidelines. For Python, this can be PEP 8 guidelines or any arbitrary coding conventions. Style fixers are able to find styling errors and apply fixes automatically. Examples of some popular Python style fixers are `black`, `autopep8`, and `yapf`. There are also highly specialized style fixers that focus only on one aspect of code style. A prime example is `isort`, which focuses on sorting `import` statements.

- **Classic linters**: These are linters that focus more on suspicious/dangerous constructs that may lead to bugs and/or undefined behaviors, although they can also include rulesets for specific style conventions. Classic linters usually work in a complain-only mode. They are able to flag issues but aren't able to fix those issues automatically. Examples of popular classic Python linters are `pylint` and `pyflakes`. A common style-only classic linter is `pycodestyle`.

> There are also some experimental hybrid linters focusing on auto-fixing suspicious code constructs, like `autoflake`. Unfortunately, due to the delicate nature of those suspicious constructs (like unused import statements or variables), it is not always possible to perform a safe fix without introducing side effects. Those fixers should be used with great care.

Both style fixers and classic linters are indispensable when writing high-quality software, especially for professional use. Popular classic linters like `pyflakes` and `pylint` have a plethora of rules for errors, warnings, and automatic recommendations, and the list of rules is ever-expanding.

A large collection of rules means that introducing one of these linters to a preexisting large project usually requires some tuning to match common coding conventions. You may find some of these rules quite arbitrary (like default line lengths, specific import patterns, or maximum function argument numbers) and decide to silence some of the default checks. That requires a bit of effort, but it is an effort that really pays off in the long term.

Anyway, configuring linters is quite a tedious task, so we won't dive any deeper into that. Both `pylint` and `pyflakes` have great documentation that describes their use clearly. Definitely more exciting than classic linters are style fixers. They usually require very little configuration or no configuration at all.

They can bring a lot of consistency to an existing codebase with just a single command execution. We will see how it works on the examples of the code bundle for this book.

All examples in this book are written usually according to the PEP 8 code style. Still being constrained by the medium, we had to make a few tweaks just to make sure the code samples are clear, concise, and read nicely on paper. These tweaks were as follows:

- **Using a single empty new line instead of two whenever possible**: There are places where PEP 8 recommends two (mostly for the separation of functions, methods, and classes). We decided to use one just to save space and avoid breaking longer code examples over two pages.

- **Using a lower characters-per-line amount**: The PEP 8 line width limit is 79 characters. It's not a lot, but it turns out that it's still too much for a book. Books typically have a portrait format and the standard 79-80 characters in monotype font would usually not fit a single line in print. Also, some readers may use ebook readers where the author has no control over the display of code samples. Using shorter lines makes it more likely that examples on paper and ebook readers will look the same.

- **Not grouping imports into sections:** PEP 8 suggests grouping imports for standard library modules, third-party modules, and local modules and separating them with a single newline. That makes sense for code modules but in a book format, where we rarely use more than two imports per example, it would only introduce noise and waste precious page space.

These small deviations from the PEP 8 guideline are definitely justified for the book format. But the same code samples are also available in the Git repository dedicated to the book. If you were to open those samples in your favorite IDE just to see the code unnaturally compacted to the book format, you would probably feel a bit uneasy. It was thus necessary to include the code bundle in the format that is best for computer display.

The book code bundle includes over 100 Python source files, and writing them in two styles independently would be error-prone and cost us a lot of time. So, what did we do instead? We worked on the book samples in the Git repository using the informal book format. Every chapter was reviewed by multiple editors, so some examples had to be updated a few times. When we knew that everything was correct and working as expected, we simply used the `black` tool to discover all style violations and automatically apply fixes.

The usage of the `black` tool is pretty straightforward. You invoke it with the `black <sources>` command, where `<sources>` is a path to the source file or directory containing the source files you want to reformat. To reformat all source files in the current working directory, you can use:

```
$ black .
```

When you run `black` over the code bundle for this book, you will see output like the following:

```
(...)
reformatted /Users/swistakm/dev/Expert-Python-Programming-Fourth-
Edition/Chapter 8/01 - One step deeper: class decorators/autorepr.py
reformatted /Users/swistakm/dev/Expert-Python-Programming-Fourth-
Edition/Chapter 6/07 - Throttling/throttling.py
reformatted /Users/swistakm/dev/Expert-Python-Programming-Fourth-
Edition/Chapter 8/01 - One step deeper: class decorators/autorepr_
subclassed.py
reformatted /Users/swistakm/dev/Expert-Python-Programming-Fourth-
Edition/Chapter 7/04 - Subject-based style/observers.py
reformatted /Users/swistakm/dev/Expert-Python-Programming-Fourth-
Edition/Chapter 8/04 - Using __init__subclass__ method as alternative
to metaclasses/autorepr.py
reformatted /Users/swistakm/dev/Expert-Python-Programming-Fourth-
Edition/Chapter 9/06 - Calling C functions using ctypes/qsort.py
reformatted /Users/swistakm/dev/Expert-Python-Programming-Fourth-
Edition/Chapter 8/04 - Using __init__subclass__ method as alternative
to metaclasses/autorepr_with_init_subclass.py
All done! ✨ 🍰 ✨
64 files reformatted, 37 files left unchanged.
```

> The actual output of the `black` command was a few times longer and we had to truncate it substantially.

What would otherwise have cost us many hours of mundane work, thanks to the `black` tool, was done in just a few seconds.

Of course, you can't rely on every developer in the project consistently running `black` for every change committed to the central code repository. That's why the `black` tool can be run in check-only mode using the `--check` flag. Thanks to this, `black` can also be used as a style verification step in shared build systems, providing the continuous integration of changes.

Tools like `black` definitely increase the quality of code by ensuring effortless and consistent formatting of code. Thanks to this, code will be easier to read and (hopefully) easier to understand. Another advantage is that it saves a lot of time that otherwise would be wasted on countless code formatting debates. But that's only a small aspect of the quality spectrum. There's no guarantee that consistently formatted code will have fewer bugs or will be visibly more maintainable.

When it comes to defect discovery and maintainability, classic linters are generally much better than any automated fixer. One subgroup of classic linters is especially great at finding potentially problematic parts of code. These are linters that are able to perform static type analysis. We will take a closer look at one of them in the next section.

Static type analysis

Python isn't statically typed but has voluntary type annotations. This single feature, with the help of highly specialized linters, can make Python code almost as type-safe as classical statically typed languages.

The voluntary nature of typing annotations has also another advantage. You can decide whether to use type annotations or not at any time. Typed arguments, variable functions, and methods are great for consistency and avoiding silly mistakes but can get in your way when you attempt to do something unusual. Sometimes you just need to add an extra attribute to a preexisting object instance using monkey patching, or hack through a third-party library that doesn't seem to want to cooperate. It is naturally easier to do so if the language doesn't enforce type checks.

The leading type of static checker for Python is currently `mypy`. It analyzes functions and variable annotations that can be defined using a type hinting hierarchy from the `typing` module. In order to work, it does not require you to annotate the whole code with types. This characteristic of `mypy` is especially great when maintaining legacy codebases as typing annotations can be introduced gradually.

> You can learn more about the typing hierarchy of Python by reading the *PEP 484 -- Type Hints* document, available at `https://www.python.org/dev/peps/pep-0484/`.

Like with other linters, the basic usage of `mypy` is pretty straightforward. Simply write your code (using type annotations or not) and verify its correctness with one `mypy <path>` command, where `<path>` is a source file or a directory containing multiple source files. `mypy` will recognize parts of the code that feature type annotations and verify that the usage of functions and variables matches the declared types.

Although `mypy` is an independent package available on PyPI, type hinting for the purpose of static analysis is fully supported by mainstream Python development in the form of a Typeshed project. Typeshed (available at `https://github.com/python/typeshed`) is a collection of library stubs with static type definitions for both the standard library and many popular third-party projects.

> You'll find more information about `mypy` and its command-line usage on the official project page at `http://mypy-lang.org`.

So far, we have discussed the topic of quality automation with regard to application code. We used tests as a tool to increase overall software quality and measured test coverage to get some understanding of how well tests have been written. What we haven't spoken about yet is the quality of tests, and this is as important as the quality of the code that is being tested. Bad tests can give you a false sense of security and software quality. This can be almost as harmful as a lack of any tests at all.

Generally, the basic quality automation tools can be applied to test code as well. This means that linters and style fixers can be used to maintain the test codebase. But those tools do not give us any quantitative measurements of how well tests can detect new and existing bugs. Measuring the effectiveness and quality of tests requires slightly different techniques. One of those techniques is mutation testing. Let's learn what that is.

Mutation testing

Having 100% test coverage in your project is indeed a satisfying thing. But the higher it is, the quicker you will learn that it is never a guarantee of bullet-proof software. Countless projects with high coverage discover new bugs in parts of the code that are already covered by tests. How does that happen?

Reasons for that vary. Sometimes requirements aren't clear, and tests do not cover what they were supposed to cover. Sometimes tests include errors. In the end, tests are just code and like any other code are susceptible to bugs.

But sometimes bad tests are just empty shells—they execute some units of code and compare some results but don't actually care about really verifying software correctness. And amazingly, it is easier to fall into this trap if you really care about quality and measure the test coverage. Those empty shells are often tests written in the last stage just to achieve perfect coverage.

One of the ways to verify the quality of tests is to deliberately modify the code in a way that we know would definitely break the software and see if tests can discover the issue. If at least one test fails, we are sure that they are good enough to capture that particular error. If none of them fails, we may need to consider revisiting the test suite.

As possibilities for errors are countless, it is hard to perform this procedure often and repeatedly without the aid of tools and specific methodologies. One such methodology is mutation testing.

Mutation testing works on the hypothesis that most faults of software are introduced by small errors like off-by-one errors, flipping comparison operators, wrong ranges, and so on. There is also the assumption that these small mistakes cascade into larger faults that should be recognizable by tests.

Mutation testing uses well-defined modification operators known as mutations that simulate small and typical programmer mistakes. Examples of those can be:

- Replacing the `==` operator with the `is` operator
- Replacing a `0` literal with `1`
- Switching the operands of the `<` operator
- Adding a suffix to a string literal
- Replacing a `break` statement with `continue`

In each round of mutation testing, the original program is modified slightly to produce a so-called mutant. If the mutant can pass all the tests, we say that it **survived the test**. If at least one of the tests failed, we say that it **was killed during the test**. The purpose of mutation testing is to strengthen the test suite so that it does not allow new mutants to survive.

All of this theory may sound a little bit vague at this point, so we will now take a look at a practical example of a mutation testing session. We will try to test an `is_prime()` function that is supposed to verify whether an integer number is a prime number or not.

A prime number is a natural number greater than 1 that is divisible only by itself and 1. We don't want to repeat ourselves, so there is no easy way to test the `is_prime()` function other than providing some sample data. We will start with the following simple test:

```
from primes import is_prime

def test_primes_true():
    assert is_prime(5)
```

```
        assert is_prime(7)

def test_primes_false():
        assert not is_prime(4)
        assert not is_prime(8)
```

We could use some parameterization, but let's leave that for later. Let's save that in the `test_primes.py` file and move to the `is_prime()` function. What we care about right now is simplicity, so we will create a very naïve implementation as follows:

```
def is_prime(number):
        if not isinstance(number, int) or number < 0:
            return False

        if number in (0, 1):
            return False

        for element in range(2, number):
            if number % element == 0:
                return False
        return True
```

It may not be the most performant implementation, but it's dead simple and so should be easy to understand. Only integers greater than 1 can be prime. We start by checking for type and against the values 0 and 1. For other numbers, we iterate over integers smaller than `number` and greater than 1. If `number` is not divisible by any of those integers it means it is a prime. Let's save that function in the `primes.py` file.

Now it's time to evaluate the quality of our tests. There are a few mutation testing tools available on PyPI. One that seems the simplest to use is **mutmut**, and we will use it in our mutation testing session. `mutmut` requires you to define a minor configuration that tells it how tests are run and how to mutate your code. It uses its own `[mutmut]` section in the common `setup.cfg` file. Our configuration will be the following:

```
[mutmut]
paths_to_mutate=primes.py
runner=python -m pytest -x
```

The `paths_to_mutate` variable specifies paths of the source files that `mutmut` is able to mutate. Mutation testing in large projects can take a substantial amount of time so it is crucial to guide `mutmut` on what it is supposed to mutate, just to save time.

The `runner` variable specifies the command that is used to run tests. `mutmut` is framework agnostic so it supports any type of test framework that has a runner executable as a shell command. Here we use pytest with the `-x` flag. This flag tells pytest to abort testing on the first failure. Mutation testing is all about discovering surviving mutants. If any of the tests fail, we will already know that the mutant hasn't survived.

Now it's time to start the mutation testing session. The `mutmut` tool's usage is very similar to that of the `coverage` tool, so our work starts with the `run` subcommand:

```
$ mutmut run
```

The whole run will take a couple of seconds. After `mutmut` finishes validation of the mutants, we will see the following summary of the run:

```
- Mutation testing starting -

These are the steps:
1. A full test suite run will be made to make sure we
   can run the tests successfully and we know how long
   it takes (to detect infinite loops for example)
2. Mutants will be generated and checked

Results are stored in .mutmut-cache.
Print found mutants with `mutmut results`.

Legend for output:
🎉 Killed mutants.   The goal is for everything to end up in this
bucket.
⏰ Timeout.          Test suite took 10 times as long as the baseline so
were killed.
🤥 Suspicious.       Tests took a long time, but not long enough to be
fatal.
🙁 Survived.         This means your tests needs to be expanded.
🔇 Skipped.          Skipped.

1. Running tests without mutations
⚡ Running...Done

2. Checking mutants
⚡ 15/15  🎉 8  ⏰ 0  🤥 0  🙁 7  🔇 0
```

The last line shows a short summary of the results. We can get a detailed view by running the `mutmut results` command. We got the following output in our session:

```
$ mutmut results
To apply a mutant on disk:
    mutmut apply <id>

To show a mutant:
    mutmut show <id>


Survived ☺ (7)

---- primes.py (7) ----

8-10, 12-15
```

The last line shows the identifiers of the mutants that survived the test. We can see that 7 mutants survived, and their identifiers are in the `8-10` and `12-15` ranges. The output also shows useful information on how to review mutants using the `mutmut show <id>` command. You can also review mutants in bulk using the source file name as the `<id>` value.

We're doing this only for illustration purposes, so we will review only two mutants. Let's take a look at the first one with an ID of 8:

```
$ mutmut show 8
--- primes.py
+++ primes.py
@@ -2,7 +2,7 @@
    if not isinstance(number, int) or number < 0:
        return False


-   if number in (0, 1):
+   if number in (1, 1):
        return False

    for element in range(2, number):
```

`mutmut` has modified the range values of our `if number in (...)` and our tests clearly didn't catch the issue. This means that we probably have to include those values in our testing conditions.

Let's now take a look at the last mutant with an ID of `15`:

```
$ mutmut show 15
--- primes.py
+++ primes.py
@@ -1,6 +1,6 @@
 def is_prime(number):
     if not isinstance(number, int) or number < 0:
-        return False
+        return True

     if number in (0, 1):
         return False
```

`mutmut` has flipped the value of the `bool` literal after the type and value range checks. The mutant survived because we included a type check but didn't test what happens when the input value has the wrong type.

In our case all those mutants could have been killed if we included more test samples in our tests. If we extend the test suite to cover more corner cases and invalid values, it would probably make it more robust. The following is a revised set of tests:

```python
from primes import is_prime

def test_primes_true():
    assert is_prime(2)
    assert is_prime(5)
    assert is_prime(7)

def test_primes_false():
    assert not is_prime(-200)
    assert not is_prime(3.1)
    assert not is_prime(0)
    assert not is_prime(1)
    assert not is_prime(4)
    assert not is_prime(8)
```

Mutation testing is a hybrid methodology because it not only verifies the testing quality but also can highlight potentially redundant code. For instance, if we implement the test improvements from the above example, we will still see two surviving mutants:

```
# mutant 12
--- primes.py
+++ primes.py
@@ -1,5 +1,5 @@
 def is_prime(number):
-    if not isinstance(number, int) or number < 0:
+    if not isinstance(number, int) or number <= 0:
         return False

     if number in (0, 1):
```

```
# mutant 13
--- primes.py
+++ primes.py
@@ -1,5 +1,5 @@
 def is_prime(number):
-    if not isinstance(number, int) or number < 0:
+    if not isinstance(number, int) or number < 1:
         return False

     if number in (0, 1):
```

Those two mutants survive because the two `if` clauses we used can potentially handle the same condition. It means that the code we wrote is probably overly complex and can be simplified. We will be able to kill those two outstanding mutants if we collapse two `if` statements into one:

```
def is_prime(number):
    if not isinstance(number, int) or number <= 1:
        return False

    for element in range(2, number):
        if number % element == 0:
            return False
    return True
```

Mutation testing is a really interesting technique that can strengthen the quality of tests. One problem is that it doesn't scale well. On larger projects, the number of potential mutants will be really big and in order to validate them, you have to run the whole test suite. It will take a lot of time to execute a single mutation session if you have many long-running tests. That's why mutation testing works well with simple unit tests but is very limited when it comes to integration testing. Still, it is a great tool for poking holes in those perfect coverage test suites.

In the past few sections, we have been focusing on systematic tools and approaches for writing tests and quality automation. These systematic approaches create a good foundation for your testing operations but do not guarantee that you will be efficient in writing tests or that testing will be easy. Testing sometimes can be tedious and boring. What makes it more fun is the large collection of utilities available on PyPI that allows you to reduce the boring parts.

Useful testing utilities

When it comes to efficiency in writing tests, it usually boils down to handling all those mundane or inconvenient problems like providing realistic data entries, dealing with time-sensitive processing, or working with remote services. Experienced programmers usually boost their effectiveness with the help of a large collection of small tools for dealing with all these small typical problems. Let's take a look at a few of them.

Faking realistic data values

When writing tests based on input-output data samples, we often need to provide values that have some meaning in our application:

- Names of people
- Addresses
- Telephone numbers
- Email addresses
- Identification numbers like tax or social security identifiers

The easiest way around that is to use hardcoded values. We've already done that in the example of our `test_send()` function in the *Mocks and unittest.mock module* section:

```python
def test_send():
    sender = "john.doe@example.com"
    to = "jane.doe@example.com"
    body = "Hello jane!"
    subject = "How are you?"
    ...
```

The advantage of doing that is that whoever reads the test will be able to visually understand the values, so it can also serve test documentation purposes. But the problem of using hardcoded values is that it does not allow tests to efficiently search through the vast space of potential errors. We've already seen in the *Mutation testing* section how a small set of testing samples can lead to low-quality tests and a false sense of security about your code quality.

We could of course solve this problem by parameterizing tests and using much more realistic data samples. But this is a lot of mundane repeatable work and many developers are not willing to do that on a larger scale.

One way around this monotony of sample data sets is using a readily available generator of data entries that could provide realistic values. One such generator is the `faker` package available on PyPI. `faker` comes with a built-in pytest plugin, which provides a `faker` fixture that can be easily used in any of your tests. The following is the modified part of the `test_send()` function that utilizes the `faker` fixture:

```python
from faker import Faker

def test_send(faker: Faker):
    sender = faker.email()
    to = faker.email()
    body = faker.paragraph()
    subject = faker.sentence()
    ...
```

On each run, `faker` will seed the test with different data samples. Thanks to this, you are more likely to discover potential issues. Also, if you want to run the same tests multiple times using various random values, you can use the following pytest parameterization trick:

```
import pytest

@pytest.mark.parametrize("iteration", range(10))
def test_send(faker: Faker, iteration: int):
    ...
```

`pytest` has dozens of data provider classes and each one has several data entry methods. Every method can be obtained directly through the `Faker` class instance. It also supports localization, so many provider classes are available in versions for different languages.

`faker` can also provide date and time entries in various standards. What it can't do is freeze time. But don't worry, we have a different package for that.

Faking time values

It may happen that for some reason you would like to change the way your application experiences the passage of time. This could be useful in testing time-sensitive processing like work scheduling or inspecting the automatically assigned creation timestamps of specific objects.

You can of course always pause your application. On POSIX systems you can pause the process with the `pause()` system call. In Python you can set a breakpoint using the `breakpoint()` function. But that doesn't affect the passage of time. Time still flows. Also, when the application is suspended, it cannot continue processing, so you can't continue testing.

What we need to do instead is to trick our code into thinking that time is moving at a different rate or is stopped at a single point without interfering with normal execution. There is a great `freezegun` package on PyPI that is capable of doing exactly that.

The usage of `freezegun` is quite simple. It offers a `@freeze_time` decorator that can be used on a test function to freeze time at a specific date and time:

```
from freeze_gun import freeze_time

@freeze_time("1988-02-05 05:10:00")
def test_with_time():
    ...
```

During the test, all calls to the standard library functions that return current time values will return the value specified with the `decorator` parameter. Among other things, this means that `time.time()` will return an epoch value and `datetime.datetime.now()` will return a `datetime` object, which are both located at the same point in time, namely `1988-02-05 05:10:00`.

The `freeze_time()` call can also be used as a context manager. It will return a special `FrozenDateTimeFactory` that allows you to precisely control the flow of time as in the following example:

```python
from datetime import timedelta
from freeze_gun import freeze_time

with freeze_time("1988-02-04 05:10:00") as frozen:
    frozen.move_to("1988-02-05 05:10:00")
    frozen.tick()
    frozen.tick(timedelta(hours=1))
```

The `move_to()` method moves the current time context to a designated point of time (string-formatted or `datetime` object) and `tick()` progresses the time by a specified interval (1 second by default).

Freezing time should of course be done really carefully. If your application actively checks the current time with `time.time()` and waits until a certain time passes, you could easily lock it in indefinite sleep.

Summary

The most important thing in developing software with TDD is always starting with tests. That's the only way you can ensure that code units are easily testable. This also naturally encourages good design practices like the single responsibility principle or the inversion of control. Sticking to those principles helps in writing good and maintainable code. And we've already seen how hard it is to test code reliably when tests are just an afterthought.

But caring about software correctness and maintainability does not end with testing and quality automation. These two allow us to verify the requirements we know about and fix bugs we have discovered. We can of course deepen the testing suite, and we've learned that mutation testing is an effective technique to discover potential testing blindspots, but this approach has its limits.

What follows next is usually the constant monitoring of the application and listening to bug reports submitted by the users. You probably don't want to treat your users as a free workforce and swarm them with countless bugs to discover, but in the long run they will be the best source of insights you can get. That's both due to their scale and the fact that they are the ones who have the most interest in having a working piece of software.

But before your users will be able to get their hands on your application, you need to package and ship it. That will be the sole topic of the next chapter. We will learn how to prepare a Python package for distribution on PyPI and also discuss common patterns for releasing web-based software and desktop applications.

11

Packaging and Distributing Python Code

In this chapter, we will focus on ways of packaging and shipping various types of Python packages. We will consider complete applications intended for end users as well as libraries that are typically consumed only by software developers.

Everyone that writes software does so for a reason. You may be a hobbyist that makes applications for fun and wants to share them with friends for their amusement. You may be a scientist or researcher that solves an important problem and wants to share code with other people to make their lives easier. Or you may be a professional that writes code for a living and you want to make your application or service available for paying customers.

Every reason for writing code is good but each one usually comes with its own preferable way of distributing the software. In this chapter, we will discuss three main scenarios:

- Packaging and distributing libraries
- Packaging applications and services for the web
- Creating standalone executables

At first, we will focus on packaging and distributing libraries as this is the scenario that may support the development of other packaging and distribution flows. But before we continue on this topic, let's first consider the technical requirements for this chapter.

Technical requirements

The following are Python packages that are mentioned in this chapter that you can download from PyPI:

- `twine`
- `wheel`
- `cx_Freeze`
- `py2exe`
- `pyinstaller`

Information on how to install packages is included in *Chapter 2*, *Modern Python Development Environments*.

The code files for this chapter can be found at `https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition/tree/main/Chapter%2011`.

Packaging and distributing libraries

A software library is a reusable piece of code that can be used as a component of a larger application or another library. Libraries usually focus on solving limited sets of problems of a specific technical area, but there is no limit for library size. For the purpose of this chapter, we will consider frameworks to be libraries too. That's because frameworks also can be understood as components of an application, although on a larger and more generic scale.

Libraries in Python are distributed in the form of packages (or modules). We've been using them throughout the book already. Most of the packages that we've obtained from PyPI in previous chapters can in fact be considered libraries. Most of the open-source Python libraries are distributed through PyPI and that's why we will discuss this topic through the prism of distributing open-source packages.

You should know how to create packages even if you are not interested in distributing your code as open-source. Knowing how to make your own packages will give you more insight into the packaging ecosystem and will help you to work with third-party code that is available on PyPI (which you are probably using already).

Python packaging can be a bit overwhelming at first. The main reason for that is the confusion about proper tools for creating Python packages. Anyway, once you create your first package, you will see that this is not as hard as it looks. Also, knowing proper, state-of-the-art packaging tools helps a lot.

But before we get to the state-of-the-art tools, let's take a closer look at the anatomy of a Python package.

The anatomy of a Python package

The minimal distributable piece of Python code is a module, which is a single source file ending with the `.py` extension. A collection of modules is called a **package**. While you could theoretically distribute your Python packages and modules as a raw source code bundle and let your users use it through the Python interpreter, it would be really problematic for non-technical people. Even developers expect some amount of minimal packaging that would allow them to install your application or library using Python packaging tools like `pip` or `Poetry`.

There are several possible layouts of the source tree for a Python package that has to be distributed on PyPI. There are a few recurring patterns and almost every package has a few common files. It's hard to tell which layout is best so let's simply consider the following layout, which is the authors' favorite:

```
.
├── packagename/
│   └── __init__.py
├── tests/
│   ├── __init__.py
│   └── conftest.py
├── bin/
├── data/
├── docs/
├── README.md
├── LICENSE
├── setup.py
├── setup.cfg
├── MANIFEST.in
└── CHANGELOG.md
```

The main structure of the project sources is dictated by the sub-directory layout. Each one has its own role:

- `packagename/`: This is the directory holding the Python sources of the package. This is the core of what is distributed on PyPI. Preferably, this has exactly the same name as the name under which the package is registered on PyPI, although many developers use dashes instead of underscores in the PyPI registration. Usually, there's only one top-level package in the source tree.

- `tests/`: This is the test package directory. It holds test modules and (optionally) test sub-packages. In the example above, we see the `conftest` module, which is a special test module of the `pytest` framework that usually contains test fixtures and optional `pytest` plugins. This directory usually isn't distributed on PyPI because the `tests` name is pretty common, and your test package would likely conflict with other test packages in the `site-packages` directory after the installation. If you want to distribute tests with your package, you should namespace it by nesting it within the main package directory (here, the `packagename/` directory).

> Some developers prefer to put a package sources directory and test package directory inside of an additional top-level `src/` directory. This doesn't change a lot and is rather a matter of personal preference.

- `bin/`: This is a directory for shell scripts and utilities that may be helpful in package development. It can hold, for instance, scripts for building documentation, custom linters, or utilities aiding in the package distribution process. These scripts are not distributed on PyPI.

> If a package has to distribute some actual shell scripts, the common convention is to put them in the `scripts/` directory.

- `data/`: This is a directory for essential data files that have to be included in package distribution. An example could be pre-trained machine learning models, images, or translation files.

- `docs/`: This is a directory for package documentation. Documentation can take any form, but many developers use automated documentation building systems like Sphinx or MkDocs. In such cases, the `docs/` directory holds documentation sources and configuration for those systems but not the rendered documentation files. This directory often isn't distributed on PyPI.

> Sphinx is a documentation generator that is used to build official Python documentation. You can learn more about Sphinx at `https://www.sphinx-doc.org`.
>
> Sphinx is powerful but quite heavyweight. Sometimes (especially for smaller packages) a more lightweight tool can be a better alternative. MkDocs is a popular static site generator that is specifically designed for building project documentation. You can learn more about MkDocs at `https://www.mkdocs.org`.

Files outside of the above directories usually provide configuration tools or hold metadata of the package. The suggested layout lists six files that are the essential minimum for an open-source package:

- `README.md`: This file contains a minimal description and/or documentation of the package. The `.md` extension denotes the **Markdown** markup language, which is a popular choice with developers. The use of dedicated markup language is fully optional and common alternative names for this file are `README` or `README.txt`. It is a good practice to include this file in package distribution.

> Another popular markup choice for documenting a Python project is **reStructuredText** (denoted by the `.rst` file extension). It is the default markup language of the Sphinx engine. You can read more about reStructuredText at `https://docutils.sourceforge.io/rst.html`.

- `LICENSE`: This file contains a software license for package users. It is usually a plain-text file without any specific markup language. Package distribution should include this file.

- `setup.py`: This is a Python package distribution script. It is used to build package distributions and upload them to the package registry. Among other things, it contains package metadata and definitions of extensions (if the package provides any). It is included only in **source distributions** (we will discuss them in the *Types of package distributions* section).

- `setup.cfg`: This is an optional Python package configuration file (INI-style). It may include package metadata and default options for `setup.py` script subcommands. Many Python development tools (test frameworks, linters) use dedicated sections in this file as their own configuration too.

- `MANIFEST.in`: This is the template file for the package file manifest. It can be used to tell the `setup.py` script which of the non-source files should be included in the package distribution.

- `CHANGELOG.md`: This is an optional file with a log of all changes made to the package up to the current release. It is a good practice to include it in the package distribution. Short changelogs can also be included in the `README` file, although for projects with frequent releases, it is usually better to have a dedicated file for that purpose.

> Many developers choose to maintain a log of changes in a more convenient form outside of the source tree. A popular example is the project's *Releases* section on GitHub. Still, it is a good practice to include at least a minimal log of changes with package distribution as well.

Some of those files have a very specific syntax or structure, which we will discuss shortly. Let's take a closer look at the most important one—the `setup.py` script.

setup.py

The root directory of a project that has a distributable Python package contains a `setup.py` script. It provides essential package metadata like version number, description, authors, license type, or required dependency. Package metadata is expressed as arguments to the `setuptools.setup()` function.

> Python provides the built-in `distutils` module for the purpose of code packaging, but it is actually recommended to use the `setuptools` instead. The `setuptools` package provides a layer of multiple enhancements over the standard `distutils` module. Also, starting from Python 3.10, the `distutils` package will be officially deprecated and the `setuptools` codebase is now independent of the `distutils` module. That's why we will be discussing the behavior of the `setuptools` package in this chapter.

Therefore, the minimum content for the `setup.py` file is as follows:

```python
from setuptools import setup

setup(
    name='mypackage',
)
```

Note that using a bare `name` argument is just enough to register the package in the package registry but it still does not allow you to create functional distributions. In order to create functional distributions, you will have to provide a little more metadata that will allow the `setuptools` package to properly collect source files. We will discuss the most important metadata entries later, in the *Essential package metadata* section.

The name argument defines the full name of the package distribution. If you decide to publish your package in a registry like PyPI, it will be registered under this exact name. From there, the script provides several commands that can be listed with the `--help-commands` option. The following is an example output:

```
$ python3 setup.py --help-commands
Standard commands:
  build             build everything needed to install
  clean             clean up temporary files from 'build' command
  install           install everything from build directory
  sdist             create a source distribution (tarball, zip file,
etc.)
  register          register the distribution with the Python package
index
  bdist             create a built (binary) distribution
  check             perform some checks on the package
  upload            upload binary package to PyPI

Extra commands:
  bdist_wheel       create a wheel distribution
  alias             define a shortcut to invoke one or more commands
  develop           install package in 'development mode'

usage: setup.py [global_opts] cmd1 [cmd1_opts] [cmd2 [cmd2_opts] ...]
   or: setup.py --help [cmd1 cmd2 ...]
   or: setup.py --help-commands
   or: setup.py cmd --help
```

> The actual list of commands is longer and can vary depending on the available `setuptools` extensions. It was truncated to show only those that are most important and relevant to this chapter.

Standard commands are the built-in commands provided by `distutils`, whereas **extra commands** are the ones provided by third-party packages, such as `setuptools` or any other package that defines and registers a new command. Here, one such extra command registered by another package is `bdist_wheel`, provided by the `wheel` package.

setup.cfg

The `setup.cfg` file contains default options for commands of the `setup.py` script. This is very useful if the process for building and distributing the package is more complex and requires many optional arguments to be passed to the `setup.py` script commands. This `setup.cfg` file allows you to store such default parameters together with your source code on a per-project basis. This will make your distribution flow bound to the project and also provides transparency about how your package was built/distributed to the users or your team members.

The syntax for the `setup.cfg` file is the same as provided by the built-in `configparser` module so it is similar to the popular Microsoft Windows INI files. Here is an example of the `setup.cfg` configuration file that provides some global defaults as well as defaults for `sdist` and `bdist_wheel` commands:

```
[global]
quiet=1

[sdist]
formats=tar,zip

[bdist_wheel]
universal=1
```

The above configuration will ensure that source distributions (the `sdist` section) will always be created in two formats (ZIP and TAR) and the built wheel distributions (the `bdist_wheel` section) will be created as universal wheels that are independent of the Python version. Also, most of the output will be suppressed on every command by the global `--quiet` switch.

> Note that the global `quiet` option is included here only for demonstration purposes and it may not be a sensible choice to suppress the output for every command by default. You can also provide a global personal configuration file named `.pydistutils.cfg` in your home directory.

MANIFEST.in

When building a source distribution with the `sdist` command, the `setuptools` module browses the package directory looking for files to include in the archive. By default, `setuptools` will include the following files based on arguments of the `setup()` function:

- All Python source files implied by the `py_modules` and `packages` arguments
- All extension source files listed in the `ext_modules` argument
- All scripts specified by the `scripts` argument
- All files specified by the `package_data` and `data_files` arguments
- The license files specified by the `license_file` and `license_files` arguments
- Files that match the glob pattern `test/test*.py`
- Files named `setup.py`, `pyproject.toml`, `setup.cfg`, and `MANIFEST.in`
- Files named `README`, `README.txt`, `README.rst`, and `README.md`

Besides that, if your package is versioned with a version control system such as Subversion, Mercurial, or Git, there is the possibility to auto-include all version-controlled files using additional `setuptools` extensions such as `setuptools-svn`, `setuptools-hg`, and `setuptools-git`. Integration with other version control systems is also possible through other custom extensions.

No matter if it uses the default built-in file collection strategy or one defined by a custom extension, `sdist` will create a `MANIFEST` file that lists all files and will include it in the final archive.

Although the `setup()` function arguments allow you to list any type of file to be included in the package distribution, listing them one by one may not be the most convenient option. Also, using the extensions for a specific version control system may capture some files that you may not want to include in your package distribution. In both cases, you can use the `MANIFEST.in` template to provide an extra manifest template to automatically include or exclude files based on the file name pattern.

Let's say you are not using any extra extensions, and you need to include in your package distribution some files that are not captured by default. You can define a template called `MANIFEST.in` in your package root directory (the same directory as the `setup.py` file). This template directs the `sdist` command on which files to include.

The `MANIFEST.in` template defines one inclusion or exclusion rule per line. The following is an example of the `MANIFEST.in` template that enables the inclusion of the `LICENSE` file, extra textual information found in `.txt` files, and all Markdown-formatted files:

```
include HISTORY.txt
include README.txt
include CHANGES.txt
include CONTRIBUTORS.txt
include LICENSE
recursive-include *.md
```

> The full list of the MANIFEST.in commands can be found in
> the official distutils documentation available at https://
> packaging.python.org/guides/using-manifest-
> in/#manifest-in-commands.

Essential package metadata

The most important argument of the setup() function is name. Without it, the
setuptools package will assume the UNKNOWN name, which won't allow you to easily
distinguish different package distributions.

Using just the name argument is of course not enough to provide proper and
functional packaging for your code. The most important arguments that the setup()
function can receive are as follows:

- version: This is the current version specifier of the package.

- description: This includes a short description of the package. It is usually
  one sentence that explains the purpose of the package.

- long_description: This includes a full description that can be in
  reStructuredText (default) or other supported markup languages.

- long_description_content_type: This defines the MIME type of the long
  description; it is used to tell the package repository what kind of markup
  language is used for the package description.

- keywords: This is a list of keywords that define the package and allow for
  better indexing in the package repository.

- author: This is the name of the package author or organization that takes
  care of it.

- author_email: This is the contact email address of the package author.

- install_requires: This lists the packages and their versions that are required
  dependencies of your package. For instance, if your package requires some
  other packages available on PyPI in order to work, you put their names
  (and their version requirements) here.

- url: This is the project URL. It is often the URL to the site where project
  sources and/or documentation are hosted.

- license: This is the name of the license (GPL, LGPL, and so on) under which
  the package is distributed.

- py_modules: A list of Python modules to include in the distribution. It can be
  used for simple projects that have only top-level modules that do not share
  a common package namespace.

- **packages**: This is a list of all package names in the package distribution; `setuptools()` provides a helpful function called `find_packages()` that can automatically find package names to include.

- **namespace_packages**: This is a list of namespace packages within a package distribution.

The above arguments are essential metadata entries that will allow you to properly build package distributions but also attribute your code to you. Pay attention to license information and all addresses (email and URLs) that will allow users to gain more information about your package and terms of use or to reach you for help.

> The `setuptools` package provides a few more metadata entries that we didn't list here. The detailed description of all package metadata entries is described in the **PEP 345** document available at `https://www.python.org/dev/peps/pep-0345/`.

One of the important but not essential arguments is `classifiers`. It allows you to categorize your application using a standardized set of software categories known as **trove classifiers**. This feature is especially useful if you want to publish your application on PyPI. Let's take a closer look at it.

Trove classifiers

PyPI provides a solution for categorizing applications with the set of classifiers called **trove classifiers**. All trove classifiers form a tree-like structure. Each classifier string defines a list of nested namespaces where every namespace is separated by the `::` substring. Their list is provided to the package definition as a `classifiers` argument of the `setup()` function.

Here is an example list of classifiers taken from the `solrq` project available on PyPI:

```python
from setuptools import setup

setup(
    name="solrq",
    # (...)

    classifiers=[
        'Development Status :: 4 - Beta',
        'Intended Audience :: Developers',
        'License :: OSI Approved :: BSD License',
        'Operating System :: OS Independent',
        'Programming Language :: Python',
```

```
        'Programming Language :: Python :: 2',
        'Programming Language :: Python :: 2.6',
        'Programming Language :: Python :: 2.7',
        'Programming Language :: Python :: 3',
        'Programming Language :: Python :: 3.2',
        'Programming Language :: Python :: 3.3',
        'Programming Language :: Python :: 3.4',
        'Programming Language :: Python :: Implementation :: PyPy',
        'Topic :: Internet :: WWW/HTTP :: Indexing/Search',
    ],
)
```

Trove classifiers are completely optional in the package definition but provide a useful extension to the basic metadata available in the setup() interface. Among others, trove classifiers may provide information about supported Python versions, supported operating systems, the development stage of the project, or the license under which the code is released. Many PyPI users search and browse the available packages by categories so a proper classification helps packages to reach their target.

Trove classifiers serve an important role in the whole packaging ecosystem and should never be ignored. There is no organization that verifies package classification, so it is your responsibility to provide proper classifiers for your packages and not introduce chaos to the whole package index.

At the time of writing this book, there are 756 classifiers available on PyPI that are grouped into the following major categories:

- Development status
- Environment
- Framework
- Intended audience
- License
- Natural language
- Operating system
- Programming language
- Topic
- Typing

This list is ever-growing, and new classifiers are added from time to time. It is thus possible that the total count of them will be different at the time you read this. The full list of currently available trove classifiers is available at `https://pypi.org/classifiers/` and can be accessed in Python code via the `trove-classifiers` package available at `https://github.com/pypa/trove-classifiers`.

We know what the typical anatomy of a Python package is. Now it's time to discuss various types of package distributions supported by standard Python packaging tools.

Types of package distributions

Package distribution is a packaging artifact that wraps Python package sources, metadata, and any additional files into a single-file archive that can be distributed to other developers either in raw form or through the package repository.

There are generally two types of distributions for Python packages:

- Source distributions
- Built (binary) distributions

Source distributions are the simplest and most platform-independent. For pure Python packages, it is a no-brainer. Such a distribution contains only Python sources, and these should already be highly portable.

A more complex situation is when your package introduces some extensions written, for example, in C. Source distributions will still work provided that the package user has the proper development toolchain in their environment. This consists mostly of the compiler and proper C header files. For such cases, the built distribution format may be better suited because it can provide already built extensions for specific platforms.

Creating source distributions is handled by the `sdist` command of the `setup.py` script. That's why they are also commonly referred to as **sdist distributions**. They are the easiest to create so let's take a look at them first.

sdist distributions

The `sdist` command is the simplest of the `setup.py` script distribution commands. It creates a release tree and copies everything that is needed to run the package to it. This tree is then archived in one or many archive files (often, it just creates one tarball). The archive is basically a copy of the source tree.

This command is the easiest way to distribute a package that would be independent of the target system. It creates a `dist/` directory for storing the archives to be distributed. Before you create the first distribution, you have to provide a `setup()` call with a version number. If you don't, the `setuptools` module will assume the default value of `0.0.0`.

To see how it works in action, let's consider the following example of the `setup.py` script:

```python
from setuptools import setup

setup(name='acme.sql', version='0.1.1')
```

Let's now run the `sdist` command for the `acme.sql` package in the `0.1.1` version:

```
$ python setup.py sdist
```

You should see the following output:

```
running sdist
...
creating dist
tar -cf dist/acme.sql-0.1.1.tar acme.sql-0.1.1
gzip -f9 dist/acme.sql-0.1.1.tar
removing 'acme.sql-0.1.1' (and everything under it)
```

If we now list the contents of the `dist/` directory, we should see the following output:

```
$ ls dist/
acme.sql-0.1.1.tar.gz
```

> On Windows, the default archive type will be ZIP.

The version specifier is used in the name of the archive. Now the archive can be distributed and installed on any system that has Python. In the `sdist` distribution, if the package contains C libraries or extensions, the target system is responsible for compiling them. This is very common for Linux-based systems or macOS because they commonly provide a compiler. But it is less usual to have it working out of the box under Windows.

If a package with extensions is intended to be used on several platforms, it should always be distributed with a prebuilt distribution format as well.

Prebuilt distributions are created with a different set of `setup.py` script commands. Let's take a look at them.

bdist and wheel distributions

To be able to distribute a prebuilt distribution, `setuptools` provides the `build` command. This command compiles the package in the following four steps:

- `build_py`: This builds pure Python modules by byte-compiling them and copying them into the build folder.
- `build_clib`: This builds C libraries, when the package contains any, using the Python compiler and creating a static library in the build folder.
- `build_ext`: This builds C extensions and puts the result in the build folder like `build_clib`.
- `build_scripts`: This builds the modules that are marked as scripts. It also changes the interpreter path when the first line was set (using the `!#` prefix) and fixes the file mode so that it is executable.

Each of these steps is a command that can also be invoked independently. The result of the compilation process is a `build/` folder that contains everything needed for the package to be installed. There's no cross-compilation option in the `setuptools` package. This means that the result of the command is always specific to the system it was built on.

When some C extensions have to be created, the build process uses the default system compiler and the Python header file (`Python.h`). For a packaged Python distribution, an extra system package for your system distribution is probably required. At least in popular Linux distributions, it is often named `python-dev` or `python3-dev`. It contains all the necessary header files for building Python extensions.

The C compiler used in the build process is the compiler that is the default for your operating system. For a Linux-based system or macOS, this would be `gcc` or `clang` respectively. For Windows, Microsoft Visual C++ can be used (there's a free command-line version available). The open-source project MinGW can be used as well. The compiler choice can also be configured through `setuptools`.

The `build` command is used by the `bdist` command to build a binary distribution. It invokes `build` and all the dependent commands and then creates an archive in the same way as `sdist` does.

Let's create a binary distribution for `acme.sql` as follows:

```
$ python setup.py bdist
```

If run on macOS, the output could be as follows:

```
running bdist
running bdist_dumb
running build
...
running install_scripts
tar -cf dist/acme.sql-0.1.1.macosx-10.3-fat.tar .
gzip -f9 acme.sql-0.1.1.macosx-10.3-fat.tar
removing 'build/bdist.macosx-10.3-fat/dumb' (and everything under it)
```

If we now list the contents of the `dist/` directory, we should see the following output:

```
$ ls dist/
acme.sql-0.1.1.macosx-10.3-fat.tar.gz     acme.sql-0.1.1.tar.gz
```

Notice that the newly created archive's name contains the name of the system and the distribution it was built on (macOS 10.3). The same command invoked on Windows will create a different system-specific distribution archive:

```
C:\acme.sql> python.exe setup.py bdist
...

C:\acme.sql> dir dist
25/02/2008  08:18    <DIR>          .
25/02/2008  08:18    <DIR>          ..
25/02/2008  08:24             16 055 acme.sql-0.1.1.win32.zip
              1 File(s)         16 055 bytes
              2 Dir(s)  22   2222        2 D free
```

> If a package contains C code, apart from a source distribution, it's important to release as many different binary distributions as possible. At the very least, a Windows binary distribution is important for those who most probably don't have a C compiler installed.

A binary release contains all resources required to use the package on the intended system. It mainly contains a folder that is copied into Python's `site-packages` folder. It may also contain cached bytecode files (the `__pycache__/*.pyc` files).

The other kind of build distributions are wheels provided by the `wheel` package. When installed (for example, using `pip`), the `wheel` package adds a new `bdist_wheel` command to the `setup.py` script. It allows the creation of platform-specific distributions (currently only for Windows, macOS, and Linux) that are better alternatives to normal `bdist` distributions. It was designed to replace another distribution format introduced earlier by `setuptools`, called **eggs**. Eggs are now obsolete, so won't be featured in the book. The list of advantages of using wheels is quite long. Here are the ones that are mentioned on the Python Wheels page available at `http://pythonwheels.com/`:

- Faster installation for pure Python and native C extension packages.
- Avoids arbitrary code execution for installation (avoids `setup.py`).
- Installation of a C extension does not require a compiler on Windows, macOS, or Linux.
- Allows better caching for testing and continuous integration.
- Creates `.pyc` files as part of the installation to ensure they match the Python interpreter used.
- More consistent installs across platforms and machines.

According to **Python Packaging Authority** (**PyPA**) recommendations, wheels should be your default distribution format. For a very long time, the binary wheels for Linux were not supported, but that has changed, fortunately. Binary wheels for Linux are called **manylinux wheels**.

> PyPA is a community formed to bring back order and organization to the Python packaging ecosystem. The Python Packaging User Guide (`https://packaging.python.org`), maintained by PyPA, is the authoritative source of information about the latest packaging tools and best practices.

The process of building manylinux wheels is unfortunately not as straightforward as for Windows and macOS binary wheels. For this kind of wheel, PyPA maintains special Docker images that serve as a ready-to-use build environment. You can find sources of these images and detailed information on how to use them on the project's GitHub page available at `https://github.com/pypa/manylinux`.

Registering and publishing packages

Packages would be useless without an organized way to store, upload, and download them. The Python Package Index is the main source of open-source packages in the Python community. Anyone can freely upload new packages and the only requirement is to register on the PyPI site at `https://pypi.python.org/pypi`.

> Packages are bound to the user, so, by default, only the user that registered the name of the package is its admin and can upload new distributions. This could be a problem for bigger projects, so there is an option to mark other users as package maintainers so that they are able to upload new distributions too.

You are not, of course, limited to only this index and all Python packaging tools support the usage of alternative package repositories. This is especially useful for distributing closed-source code among internal organizations or for deployment purposes. Here we focus mainly on open-source uploads to PyPI, with only a brief mention of how to specify alternative repositories.

The easiest way to upload a package is to use the following `upload` command of the `setup.py` script:

```
$ python setup.py <dist-commands> upload
```

Here, `<dist-commands>` is a list of commands that creates distributions to upload. Only distributions created during the same `setup.py` execution will be uploaded to the repository. So, if you want to upload the source distribution, built distribution, and wheel package all at once, then you need to issue the following command:

```
$ python setup.py sdist bdist bdist_wheel upload
```

When uploading using `setup.py`, you cannot reuse distributions that were already built during previous distribution command executions and you are instead forced to rebuild them on every upload. This may be inconvenient for large or complex projects where the creation of the actual distribution may take a considerable amount of time. Notable examples are packages leveraging Python/C API extensions (see *Chapter 9*, *Bridging Python with C and C++*).

Another problem with `setup.py` upload is that it could use plain-text HTTP or unverified HTTPS connections on some older Python versions or if your system is not configured properly. This is why Twine is recommended as a secure replacement for the `setup.py` upload command.

Twine is the utility for interacting with PyPI that currently serves only one purpose—securely uploading packages to the repository. It supports any packaging format and always ensures that the connection is secure. It also allows you to upload files that were already created, so you are able to test distributions before the release. The following example usage of Twine still requires invoking the `setup.py` script for building distributions:

```
$ python setup.py sdist bdist_wheel
$ twine upload dist/*
```

Twine of course won't guess your credentials and you need to provide them in the special `.pypirc` file. The `.pypirc` file is a configuration file that stores information about Python package repositories. It should be located in your home directory. The format for this file is as follows:

```
[distutils]
index-servers =
    pypi
    other

[pypi]
repository: <repository-url>
username: <username>
password: <password>

[other]
repository: https://example.com/pypi
username: <username>
password: <password>
```

The `distutils` section should have the `index-servers` variable that lists all sections describing all the available repositories and credentials for them. There are only the following three variables that can be modified for each repository section:

- `repository`: This is the URL of the package repository (it defaults to `https://pypi.org/`).
- `username`: This is the username for authentication in the given repository.
- `password`: This is the user password for authentication in the given repository (in plain text).

Note that storing your repository password in plain text may not be the wisest security choice. You can always leave it blank. Twine will prompt you for credentials when it needs them.

> Another option for the safe handling of your PyPI credentials is to use the `keyring` package. It will allow Twine to interact with your system keyring service, like Keychain for macOS or Windows Credential Locker. You can read more about this feature at `https://twine.readthedocs.io/en/latest/index.html#keyring-support`.

The `.pypirc` file should be respected by every packaging tool built for Python. While this may not be true for every packaging-related utility out there, it is supported by the most important ones, such as `pip`, `twine`, `distutils`, and `setuptools`.

The danger of using the `.pypirc` file with Twine is that Twine is by default set to publish packages on PyPI. That may be a problem if you're working with closed-source code and want to publish your package in a private package index. If you forget to use the proper registry argument (the `-r` flag) and actually have your `.pypirc` file configured to work with PyPI, you may accidentally make your closed code accessible to the public.

> One of the tools that solves multiple problems of packaging Python code is Poetry. It doesn't require providing custom distribution scripts (the `setup.py` scipts are replaced with the `pyproject.toml` configuration file), is fully interactive, and allows you to specify a dedicated package registry together with the source code of your project. Usually, distributing packages with Poetry is as simple as running two commands:
>
> ```
> $ poetry build
> $ poetry publish
> ```
>
> You can learn more about building and publishing packages with Poetry at `https://python-poetry.org/docs/cli/#publish`.

Package versioning and dependency management

If you have your package published on the package registry, chances are that you will want to modify it at some point and publish a new version of it. In order to allow developers to decide whether they want to use a new release of the package or not, we use **version specifiers** to tag consecutive releases of the package.

A version specifier generally takes the form of a string composed of numbers separated by dots (like `1.0`, `3.6.5`, or `4.0.0`). That's why version specifiers are also commonly referred to as **version numbers**. This allows for easy sorting of the version specifiers. By convention, a higher version means a newer release. This convention is assumed by almost every package versioning tool and allows for straightforward updates of outdated packages to their newer version. For instance, with `pip` you can install a newer package version using the `-U` switch as in the following example:

```
$ pip install -U pip
Collecting pip
  Using cached pip-21.0.1-py3-none-any.whl (1.5 MB)
Installing collected packages: pip
  Attempting uninstall: pip
    Found existing installation: pip 20.2.4
    Uninstalling pip-20.2.4:
      Successfully uninstalled pip-20.2.4
Successfully installed pip-21.0.1
```

In the above example, we've used `pip` to update itself (it is distributed as a package). The output shows that the currently installed `pip` version was `20.2.4`. At the time of running this command, the most recent `pip` version on PyPI was `21.0.1`. `pip` compared those two version specifiers and decided that the one available on PyPI is a higher version number. It uninstalled the old version and installed the new one in the current environment.

Although package versions are usually composed only of numbers, Python allows you to use letters in version specifiers. This allows you, for instance, to tag specific versions as pre-releases, development releases, or post-release. Those extra version specifier components are usually included as the last version specifier segment just after the numeric segments.

The PEP 440 document (*Version Identification and Dependency Specification*) is the official standard for versioning packages that, among other things, specifies the following conventions for those special release tags:

- `{a|b|rc}N`: Designates the pre-release version (alpha, beta, or release-candidate). These tags designate versions at various stages of development. Alpha releases are the earliest stages and release-candidates are close to being the final versions. A package can have multiple versions in any pre-release stages and they are distinguished by raising the `N` number. An example progression of pre-release versions could be: `1.0.0a1`, `1.0.0a2`, `1.0.0b1`, and `1.0.0rc`. Versions without pre-release tags are considered final and always take precedence over pre-releases with the same number prefix.

> `pip` does not install pre-releases and development versions by default. If you want to install a pre-release version, you need to use the `--pre` option of the `pip install` command.

- `postN`: Designates a post-release version. Post-releases are often used to release an update that does not constitute a functional fix or enhancement. Examples could be updates to package metadata or documentation (if it is included in the package distribution). The same version number can have multiple post releases and they are distinguished by raising the `N` number. Post-releases can also be added on top of pre-releases. Example post-release version specifiers could be `1.0.0-post1`, `1.0.0a1.post1`, and `1.0.0.a1.post2`.

- `devN`: Designates a developmental release. Some package maintainers choose to publish packages as part of continuous integration systems and those developmental versions can be used to distinguish consecutive builds of the package. The same version number can have multiple developmental releases distinguished by raising the `N` number. Developmental releases can also be added on top of pre-releases and post-releases, although this practice is strongly discouraged on general-purpose public package indexes.

> You can access the full version of the PEP 440 document at `https://www.python.org/dev/peps/pep-0440/`.

Pre-releases, post-releases, and dev-releases add some complexity to the package versioning and thus are not used by many package maintainers. Anyway, at least pre-releases can be a useful tool to give developers the ability to preview and evaluate a future release of the package in their own environment.

What matters most is the final version number of the package. There are two popular versioning strategies to decide what number to assign to the new package release:

- **Semantic versioning**: This strategy assumes that each numeric component has a semantic value that allows package consumers to infer the amount and scope of changes between two versions.

- **Calendar versioning**: This strategy assumes that selected numeric components are derived from the date on which the new release was crafted (or was supposed to be crafted). This allows users to infer the amount of development time that has passed between two versions.

To make things easier, the community has come up with with two standards for those versioning strategies to ease their adoption. Let's take a closer look at them.

The SemVer standard for semantic versioning

The **SemVer** standard assumes that a version specifier consists of, at most, three numerical segments:

- The `MAJOR` segment: Changing the `MAJOR` segment is a sign of a backward-incompatible change. Users updating between two major versions should expect that their code may no longer be working properly.

- The `MINOR` segment: Changing the `MINOR` segment is a sign of new backward-compatible feature upgrades. Users updating between two minor versions (within the same major version) should not expect their code to become invalid but may receive new functional enhancements.

- The `PATCH` segment: Changing the `PATCH` segment is a sign of bug fixes. Users updating between two patch versions (within the same major and minor versions) should expect some issues to be fixed but should not expect any other enhancements or new features.

A proper SemVer version always includes all three segments in the following order:

```
MAJOR.MINOR.PATCH
```

For instance, version `20.2.4` of a package would mean it is on its 20th major update, with 2 minor updates and 4 patches. According to SemVer versioning principles, users updating from version `20.2.0` or `20.1.0` should not expect any braking changes.

The full specification also covers usage of pre-release versions and build numbers and provides guidance on communicating API changes and handling feature deprecation policies. You can access the full specification text at `https://semver.org`.

CalVer for calendar versioning

CalVer is more of a versioning blueprint than a full-fledged standard (especially when compared to SemVer). It assumes that a version specifier is composed of segments corresponding to elements of the date associated with a particular release.

The site explaining the CalVer convention lists the following common date-based segments:

- `YYYY`: Full year: 2006, 2016, 2106
- `YY`: Short year: 6, 16, 106
- `0Y`: Zero-padded year: 06, 16, 106
- `MM`: Short month: 1, 2 ... 11, 12
- `0M`: Zero-padded month: 01, 02 ... 11, 12
- `WW`: Short week (since start of year): 1, 2, 33, 52
- `0W`: Zero-padded week: 01, 02, 33, 52
- `DD`: Short day: 1, 2 ... 30, 31
- `0D`: Zero-padded day: 01, 02 ... 30, 31

> All CalVer segments are based on the Gregorian calendar.

This convention is best suited for projects that have a well-defined release schedule or are somehow time-sensitive. Example time-sensitive projects are `certify` (a bundle of Mozilla-curated lists of trusted root certificates that changes regularly) and `tzdata` (a bundle of IANA time zone databases, see *Chapter 3*, *New Things in Python*).

There's no common format of CalVer versions and users of CalVer have to decide on their own which version segments to use. The deciding factor is usually the release cadence of the project. This convention can also be mixed to some extent with semantic versioning. The `pip` project, for instance, used a versioning scheme composed of `YY.MINOR.PATCH` segments.

The official site for the CalVer convention isn't as thorough as the SemVer specification but provides some interesting case studies and guidelines for calendar versioning. You can find it at `https://calver.org`.

Installing your own packages

Working with `setuptools` is mostly about building and distributing packages. However, you still need to use `setuptools` to install packages directly from project sources. And the reason for that is simple. It is a good habit to test if our packaging code works properly before submitting your package to PyPI. And the simplest way to test it is by installing it. If you send a broken package to the repository, then in order to re-upload it, you need to increase the version number.

Testing if your code is packaged properly before the final distribution saves you from unnecessary version number inflation and obviously from wasting your time.

Installing packages directly from sources

Installation directly from your own sources using `setuptools` may be essential when working on multiple related packages at the same time:

```
setup.py install
```

The `install` command installs the package in your current Python environment. It will try to build the package if no previous build was made and then inject the result into the filesystem directory where Python is looking for installed packages. If you have an archive with a source distribution of some package, you can decompress it in a temporary folder and then install it with this command. The `install` command will also install dependencies that are defined in the `install_requires` argument. Dependencies will be installed from PyPI.

When installing a package, an alternative to the `setup.py` script is to use `pip`. Since it is a tool that is recommended by PyPA, you should use it even when installing a package in your local environment just for development purposes. In order to install a package from local sources, run the following command:

```
pip install <project-path>
```

If you want to install a package from the distribution archive, this command becomes:

```
pip install <path-to-archive>
```

Amazingly, the `setup.py` script lacks the `uninstall` command. Fortunately, it is possible to uninstall any Python package using `pip` as follows:

```
pip uninstall <package-name>
```

Uninstalling can be a dangerous operation when attempted on system-wide packages. This is another reason why it is so important to use virtual environments for any development.

Installing packages through the `setup.py` script or the `pip install` command copies the sources of the package (or contents of the distribution) to your `site-packages` directory. But sometimes we want to make package sources available in a specific environment without copying them. This method of installation is called **editable-mode installation** and is especially useful when working on multiple related packages that have independent source trees.

Installing packages in editable mode

Packages installed with `setup.py install` are copied to the `site-packages` directory of your current Python environment. This means that whenever you make a change to the sources of that package, you are required to reinstall it. This is often a problem during intensive development because it is very easy to forget about the need to perform the installation again.

This is why `setuptools` provides an extra `develop` command that allows you to install packages in the development mode. This command creates a special link to project sources in the deployment directory (`site-packages`) instead of copying the whole package there. Package sources can be edited without the need for reinstallation and are available in `sys.path` as if they were installed normally.

`pip` also allows you to install packages in such a mode. This installation option is called **editable mode** and can be enabled with the `-e` parameter in the `install` command as follows:

```
pip install -e <project-path>
```

Once you install the package in your environment in editable mode, you can freely modify the installed package in place and all the changes will be immediately visible without the need to reinstall the package.

Using editable mode helps when you need to work with multiple related packages without the need to reinstall them continuously. Another practice that is helpful in projects composed of multiple related packages is using namespace packages.

Namespace packages

The *Zen of Python* says the following about namespaces:

> *Namespaces are one honking great idea – let's do more of those!*

And this can be understood in at least two ways. The first is a namespace in the context of the language. We all use the following namespaces without even knowing:

- The global namespace of a module
- The local namespace of the function or method invocation
- The class namespace

The other kind of namespaces can be provided at the packaging level. These are namespace packages. This is often an overlooked feature of Python packaging that can be very useful in structuring the package ecosystem in your organization or in a very large project.

Namespace packages can be understood as a way of grouping related packages, where each of these packages can be installed independently.

Namespace packages are especially useful if you have components of your application developed, packaged, and versioned independently but you still want to access them from the same namespace. This also helps to make clear to which organization or project every package belongs. For instance, for some imaginary Acme company, the common namespace could be `acme`. Therefore, this organization could create the general `acme` namespace package that could serve as a container for other packages from this organization. For example, if someone from Acme wants to contribute to this namespace with, for example, an SQL-related library, they can create a new `acme.sql` package that registers itself in the `acme` namespace.

It is important to know what's the difference between normal and namespace packages and what problem they solve. Normally (without namespace packages), you would create a package called `acme` and a `sql` sub-package with the following file structure:

```
acme/
├── acme
│   ├── __init__.py
│   └── sql
│       └── __init__.py
└── setup.py
```

Whenever you want to add a new sub-package, let's say `templating`, you are forced to include it in the source tree of `acme` as follows:

```
acme/
├── acme
│   ├── __init__.py
│   ├── sql
│   │   └── __init__.py
│   └── templating
│       └── __init__.py
└── setup.py
```

Such an approach makes the independent development of `acme.sql` and `acme.templating` almost impossible. The `setup.py` script will also have to specify all dependencies for every sub-package. It is impossible (or at least very hard) to have an optional installation of some of the `acme` components. Also, with enough sub-packages, it may be hard to avoid dependency conflicts.

With namespace packages, you can store the source tree for each of these sub-packages independently as follows:

```
acme.sql/
├── acme
│   └── sql
│       └── __init__.py
└── setup.py

acme.templating/
├── acme
│   └── templating
│       └── __init__.py
└── setup.py
```

And you can also register them independently in PyPI or any package index you use. Users can choose which of the sub-packages they want to install from the `acme` namespace, but they never install the general `acme` package (it doesn't even have to exist). Example `pip` usage would be as follows:

```
$ pip install acme.sql acme.templating
```

Note that the `setuptools.find_packages()` function does not find namespace packages. If you want your `setup.py` script to collect namespace packages automatically instead of listing them individually, you need to use the `setuptools.find_namespace_packages()` instead.

This function will automatically discover namespace packages in directory structures as presented in the previous example.

Packages and namespace packages are concerned mostly with sharing code between projects that run in various environments. If you install such a package in a given environment, it will be immediately available for imports. But that's not the only purpose of Python packaging. Many Python projects provide shell utilities, commands, or even applications with graphical interfaces. A great example is the `pip` command distributed with the `pip` package. You can use the Python packaging infrastructure to surface your application scripts and executable modules in the target installation environment the same way the `pip` package does. Let's see how to do this.

Package scripts and entry points

Every Python module can be executed as if it were a program using the `python -m` command. This includes standard library modules as well as modules from packages installed by `pip`. For instance, the following is an invocation of the `json.tool` module from the standard library that allows you to format JSON text in your shell:

```
$ echo '{"name": "John Doe", "age": 42}' | python -m json.tool
{
    "name": "John Doe",
    "age": 42
}
```

That's a simple way to execute any module from the installed package, but not the most convenient one. Most of all, users of your package will have to know what the structure of modules is inside of your application and know which modules are supposed to be run in the shell. Also, users will have to type the `python -m` command, which adds a bit of redundancy to their scripts. That's why when using `pip`, we'd rather invoke the `pip` command than `python -m pip`.

When writing your own Python packages, you can do the same as what the `pip` package does and provide your own custom shell command that will be installed together with your package. There are two ways to do that:

- Through the `scripts` argument of the `setuptools.setup()` function
- Through the `entry_points` argument of the `setuptools.setup()` function

The `scripts` argument is the most basic method of providing shell commands through your package. The argument is already supported by the `distutils` module (the standard library module that `setuptools` is based on) so it is quite simple. It accepts a list of script file paths that are to be distributed with your package. After package installation, these scripts become available in one of the `PATH` directories associated with your Python environment.

To see how it works, we will reuse the example of the script that finds imports within Python sources from *Chapter 3, New Things in Python*. The full code and detailed explanation can be found in that chapter. We will start by creating the `findimports.py` file with the following contents:

```python
import os
import re
import sys

import_re = re.compile(r"^\s*import\s+\.{0,2}((\w+\.)*(\w+))\s*$")
import_from_re = re.compile(
    r"^\s*from\s+\.{0,2}((\w+\.)*(\w+))\s+import\s+(\w+|\*)+\s*$"
)

def main():
    if len(sys.argv) != 2:
        print(f"usage: {os.path.basename(__file__)} file-name")
        sys.exit(1)

    with open(sys.argv[1]) as file:
        for line in file:
            if match := import_re.match(line):
                print(match.groups()[0])

            if match := import_from_re.match(line):
                print(match.groups()[0])

if __name__ == "__main__":
    main()
```

From there, we will create the following `setup.py` script with some basic metadata and the `scripts` argument:

```python
from setuptools import setup

setup(
```

```
    name="findimports",
    version="0.0.0",
    py_modules=["findimports.py"],
    scripts=["findimports"],
)
```

Now you are able to install the package either in editable mode using one of the following commands:

```
$ pip install -e .
$ python setup.py develop
```

Or if you prefer, you can install the package in normal mode:

```
$ pip install .
$ python setup.py install
```

Once we've installed the package, the `findimports` module will be available as a shell command. On macOS or Linux, we can use `compgen` and `grep` to search through all discoverable commands and see that it is now indeed available in your shell:

```
$ compgen -c | grep findimports
findimports.py
```

As you see, the `findimports.py` script is now available under the name that is exactly the same as the script file name. If you really want to omit the `.py` extension from the shell command, you have one of two options:

- **Remove the .py extension from the module file name**: You will have to update the `setup.py` script accordingly. The drawback of this approach is that you will no longer be able to distribute the `findimports` module as an importable Python module (the `py_modules` argument). It would also make unit-testing of the script module harder.

- **Create a wrapper script for** `findimports.py`: The `scripts` argument allows you to distribute any type of script including shell scripts. Here, we could create a wrapper shell script with a name without an extension (for instance, `scripts/findimports`) and specify it as the target of the `scripts` argument. The file could be as simple as the following:

  ```
  #!/usr/bin/env sh
  python -m findimports
  ```

The problems with script file extensions and wrapper scripts in `distutils` can be avoided thanks to the `entry_points` extension offered by the `setuptools` module. It is a standardized way to provide application entry points (like shell scripts) via the configuration in the `setup.py` distribution script. It allows you to target any function within your package sources to be distributed as a shell script. This greatly simplifies the management of application entry points because you don't need to create dedicated runnable modules.

There are various types of entry points possible but the most common is `console_scripts`, which allows you to register the module or function as a target of the autogenerated script command. The following is an example of the console entry point we could provide for our `findimports` script:

```python
from setuptools import setup

setup(
    name="findimports",
    version="0.0.0",
    py_modules=["findimports"],
    entry_points={
        "console_scripts": ["findimports=findimports:main"]
    }
)
```

The usage of console entry points is more flexible when it comes to the naming of commands and the selection of what to exactly run when a command is invoked. On the left side of the = sign, we have the desired name of the command. In our case, it is simply `findimports`. On the right side, we have a module import path (`findimports` again) together with the name of the function (the `main()` function) to execute.

The `entry_points` argument allows for better naming of commands as well as packing multiple commands into a single Python module. But it doesn't mean that the `scripts` argument becomes useless. You can't, for instance, package shell scripts (like Bash) with `entry_points` but you can do that with the `scripts` argument.

> The feature of entry points in the `setuptools` package is in fact a generic method of advertising hooks between packages. Every package can query for existing entry points of other packages. This feature can be used, for instance, to create a plugin mechanism. The `pytest` unit-testing framework is an example package that uses the mechanism of entry points for its plugin system. You can learn more about writing `pytest` plugins at `https://docs.pytest.org/en/stable/writing_plugins.html`.

Python packaging, thanks to binary wheels and features that allow packaging scripts, can be a method of distributing complete applications. If you use virtual environments, you can ensure a sensible amount of dependency isolation between various applications.

Unfortunately, Python packaging and virtual environments don't solve all environment isolation problems. You cannot, for instance, shield your applications from changes in shared system libraries through virtual Python environments. Also, not every Python dependency you use will be distributed in a binary wheel format. Python extensions written in C, C++, or Cython are amazingly popular, which means that for complex applications, an on-site compilation may often be required. Lack of pure dependency isolation and a common need for on-site compilation are the main reasons why Python packages often aren't reliable distribution artifacts for specific use cases. One such use case is packaging applications and services for the web.

Packaging applications and services for the web

The distribution of software is a process that traditionally requires two parties. Someone (the distributor) has to make the software release available to be consumed. In the past, it required physical mediums like floppy disk or CD, but nowadays it is usually done through the internet. Someone else (the consumer) needs to consciously obtain the software and install it on their own computer. It's not always the same for software updates as many applications offer automated updates. Still, these updates usually require the user's consent in order to be installed.

With the advent of **Software as a Service** (**SaaS**), less and less software is distributed in a form that would allow it to be installed on the user's own computer. We see that classic programs are gradually being replaced by their SaaS counterparts:

- Traditional desktop applications are being replaced with web-based software
- Traditional software libraries are being replaced by web APIs

Web-based software isn't distributed to its users the same way as a traditional web desktop application. Users of web-based applications usually interact with them through a standard web browser or a dedicated client that acts as a mere shell for your code that lives on some server or cluster of servers. It has to indeed be distributed to these servers anyway but the whole process is usually opaque to the end users and they rarely are aware of this process.

That's why many developers often prefer the term **shipping** in the context of web-based applications: consumers consciously sign up as users of the software but have very limited control over how and when it will be delivered. Also, potential updates are just shoved through their door, and cannot be easily rejected or discarded.

Web-based applications are increasingly popular. Even applications that are primarily intended for desktop use often provide web-based capabilities like automated updates, cloud synchronization, or online collaboration. It means that it is worth knowing the basics of shipping those web applications even if the web is not your thing.

In this section, we will discuss good practices and tools for building and distributing web applications together with some Python-specific tips and tricks.

The Twelve-Factor App manifesto

Being able to distribute software only to your own servers removes one important factor from the distribution process: users. You don't need to care if they are able to download your application and handle the installation process. You don't have to care about their operating system (although you may need to care about their browser). And also, most of the time, you don't have to ask for permission to perform the update. You can do whatever you want. But should you?

Web-based software comes with a lot of advantages. You have full control of it. You can do as many updates as you want and whenever you want. But that is a double-edged sword. Users of web applications will expect frequent updates and almost immediate fixes for problems they submit. Also, if your software becomes successful, you will have to rely on a large fleet of servers to support the growing scale of your user base. And a large user base is usually the goal of web-based applications.

That's why it is extremely important to build your software in a way that will enable its growth at a sustainable pace. Your application should be easily configurable and decoupled from its dependencies (like external services and the operating system) to ensure easy maintenance and straightforward, repeatable deployments of new versions. It should also be as easy to deploy in a production environment as to run locally for development (and vice versa).

That's of course not easy to do without some operational knowledge. If you have not got much experience working with software on a large scale, you will definitely make a lot of mistakes that will cost you a lot of time, resources, and money (server costs for instance). That's why it is a good idea to follow a set of good, proven practices.

The **Twelve-Factor App** manifesto is a good set of such practices. It is a general language-agnostic methodology for building SaaS apps. One of its purposes is making applications easier to deploy, but it also highlights other topics such as maintainability or making applications easier to scale.

As the name says, the Twelve-Factor App consists of 12 rules:

1. **Codebase**: One codebase tracked in a revision control system and many deploys
2. **Dependencies**: Explicitly declare and isolate dependencies
3. **Config**: Store configurations in the environment
4. **Backing services**: Treat backing services as attached resources
5. **Build, release, run**: Strictly separate build and run stages
6. **Processes**: Execute the app as one or more stateless process
7. **Port binding**: Export services via port binding
8. **Concurrency**: Scale out via the process model
9. **Disposability**: Maximize robustness with fast startup and graceful shutdown
10. **Dev/prod parity**: Keep development, staging, and production as similar as possible
11. **Logs**: Treat logs as event streams
12. **Admin processes**: Run administration/management tasks as one-off processes

> You can access the full text of the Twelve-Factor App manifesto at `https://12factor.net`.

We won't discuss every factor in detail as the Twelve-Factor App website provides a great explanation and rationale for all of them. We will anyway zoom into specific rules because some of them can be employed using popular tools, techniques, or libraries that are popular in the Python ecosystem.

Leveraging Docker

We've already introduced Docker in *Chapter 2*, *Modern Python Development Environments*, as a lightweight virtualization tool that can provide great development environment isolation.

It simply packages all your code and its runtime dependencies (modules, packages, shared libraries) into container images that can be executed as isolated containers in given environments.

Moreover, Docker containers are stateless. This means that two containers started from the same image will have the same initial state. Every filesystem modification done within a container stays inside of the container. Part of the filesystem inside of a container can be of course exported outside by mounting a dedicated volume but this is always explicit and never happens by accident. A container that has finished its work (the main process exited, either gracefully or due to abrupt termination) is out of use, the same as its internal state.

> In fact, Docker containers do not vanish by default after they exit. The automatic removal of exited containers is ensured with the `--rm` flag of the `docker run` command. It is possible to resume working with the container after it has finished, although this should be used only for inspection and not as a default means of operation.

The way Docker containers and their images are defined, run, and managed already ticks multiple checkboxes of the Twelve-Factor App manifesto:

- **Dependencies**: To create a new Docker image, you need to define a Dockerfile, which is a declarative statement of all the preparation steps. This includes all shared libraries, packages, and your own code. Moreover, the multi-stage Docker builds allow you to separate build-time dependencies from runtime builds. Dependencies are isolated. You can have multiple containers from different Docker images running on the same host systems and their dependencies will never conflict.

- **Build, release, run**: Docker images are usually built outside of their dedicated runtime environment. It can be a dedicated build server or even your own computer used for development. Images are usually stored in a dedicated image repository. From there, Docker daemons running in target environments can pull the latest image version. Moreover, the tagging of images with descriptive labels allows you to easily keep track of their versions and even designation for a specific environment.

- **Processes**: Docker containers are stateless. Moreover, a container looks like a single process from the perspective of the operating system that hosts it. It sandboxes all threads or subprocesses that may be running within a container as well as all resources it may use (memory, for instance).

- **Dev/prod parity**: Packaging software into containers allows you to reduce the gap between production and development environments because it isolates a lot of dependencies from the operating system. Also, Docker Compose allows you to compose whole applications from multiple containers and use the same versions of backing services (databases, caches, reverse proxies, and so on) as the ones used in the production environment.

The great thing about Docker is the portability of the applications. As long as your target system can run the Docker daemon, it will be able to run your containers.

If you operate your own cluster of servers (physical or virtual), you will have to provision them with the Docker daemon and also provide some configuration and/or scripting that will ensure your containers are always up and running. But this is something you would have to do anyway with any kind of software. Docker may make your life easier because every application will have the same type of deliverable—a container image—and will not require a complex installation process. The management of containers alone can be done, for instance, with `systemd`, a common system and service manager found in most Linux distributions.

> We've discussed the topic of creating Docker images using Dockerfiles in *Chapter 2, Modern Python Development Environments.* You can learn more about best practices for writing Dockerfiles at `https://docs.docker.com/develop/develop-images/dockerfile_best-practices/`.

But not all organizations are willing to support their own infrastructure. Fortunately, many cloud providers offer various services that can take a lot of the operation burden from Docker users. For a larger scale, you can use dedicated container orchestration systems like **Kubernetes** (**k8s**). Kubernetes is a container orchestration system designed by Google. It organizes collections of application containers that should run on the same cluster node into groups called **Pods**. Kubernetes can manage container volumes, configuration maps, control automated scaling of services, and manage communication within the cluster as well as incoming traffic.

> You can learn more about Kubernetes at `https://kubernetes.io`.

Kubernetes can handle a range of container orchestration needs, from managed Kubernetes clusters where you can decide how many worker nodes you need and how to configure them, to fully **serverless** offerings where you simply provide Docker images with their configuration and the cloud provider takes care of scaling the infrastructure for you. Flexible on-demand pricing often means you pay only for allocated resources. This allows you to avoid large upfront infrastructure costs and to "scale as you grow."

Docker is of course not the only way for applications to be portable between hosts or service providers. But regardless of the packaging format, your application won't be portable if it isn't configurable in a system- and application-agnostic way. Let's take a look at typical configuration options for applications.

Handling environment variables

Every application will require configuration values that will vary between environments. Examples can be:

- Connection strings (URLs), hostnames, and ports of backing services like caches, databases, proxy servers, or web APIs
- Credentials to those services
- Other secrets like encryption keys and client certificates
- Per-environment values like feature toggles or resource limits

These configuration values should always be separated from the application code and definitely shouldn't be stored as constants in modules. That's especially important for values that have to be kept secret. There are multiple reasons for that:

- The first one is security. If the code contains information about secrets and credentials, whoever gets access to the code will know them all. And if someone gets access to the code repository, they will know all and also past secrets. That poses a real security risk.

- Another reason for decoupling configuration from applications is the volatility of environments—they come and go. On one day, you may work with just a few environments, but on another day, you may want to create more of them. What if you want to create a new short-lived environment for every feature branch you work on? What if you would like to do the same for every team member on the project. Do you really want to keep all those configurations in the same project repository?

- Last but not least, the configuration should be language- and framework-agnostic. You will eventually use different technologies to run your software. You may change your framework or maybe even move from Python to a completely different language. You may also want to migrate from one infrastructure to another at some point in time. Today it may be a simple application running in a virtual environment on one host but tomorrow it may be a Docker container in a Kubernetes cluster. Or even some serverless function managed by your cloud provider. You never know how your application will evolve so you need to be sure that the way you provide configuration to your application is as generic as possible.

The most universal way to provide configuration to your application is through **environment variables**. This is a simple key-value mapping that should be supported by every operating system and every programming language. They can be easily changed without any code or file modification. They are stored only in a running process environment (which is ephemeral) so are also better suited for providing secret values to your application.

The biggest advantage of using environment variables for configuration is that they can be completely decoupled from application source code. Thanks to this, you will be able to use the same deployable artifact (like a Docker container image or Python package) in various environments and tune it just by providing new environment variable values on application startup. This approach reduces the version drift between environments and allows you to avoid the bundling of secret variables in your application packages. Also, you may eventually decide to use code written in various frameworks or even languages. Environment variables allow the same configuration medium across different technologies (as opposed to dedicated configuration files or modules).

Using environment variables is easy. If you're working on Linux, macOS, or another POSIX-compliant system, you can set a new environment variable value using the `export` command as in the following example:

```
$ export MY_VARIABLE="my-value"
```

In those systems, you can also set specific variables just for the scope of one command invocation. You do that by prepending a series of variables to the command:

```
$ VARIABLE_1="value-1" command
```

On Windows, if you're using PowerShell, you can set an environment variable value through the special `$env` variable:

```
$ $env:TEAMS="my-value"
```

If you use CMD on Windows, you can also use the `set` command:

```
$ set MY_VARIABLE="my-value"
```

> Environment variable names on Linux and macOS are case-sensitive but on Windows are case-insensitive. That's why a good convention is to use the uppercase naming convention for environment variables, the same as you would do for constants in code.

As you can see, depending on the environment, there are different ways to set the environment variables. Moreover, for container-orchestration systems like Kubernetes or provider-specific cloud services, you won't be interacting with the system shell directly. You will usually be setting the desired environment values through dedicated service manifest files or the provider API.

What doesn't change between those environments is the way you read those variables. Environment variables in Python are exposed in the `environ` variable in the built-in `os` module. It is a dict-like object that allows access to and the modification of environment variables.

`os.environ` can be accessed at any time but the common convention is to create a single module in your application that accesses all environment variables. Thanks to this, you get a good overview of all configuration options supported by the application and are in control of all value processing and validation.

The example configuration for a small application could be as follows:

```python
import os

DATABASE_URI = os.environ["DATABASE_URI"]
ENCRYPTION_KEY = os.environ["ENCRYPTION_KEY"]

BIND_HOST = os.environ.get("BIND_HOST", "localhost")
BIND_PORT = int(os.environ.get("BIND_PORT", "80"))

SCHEDULE_INTERVAL = timedelta(
    seconds=int(os.environ.get("SHEDULE_INTERVAL_SECONDS", 50))
)
```

As you can see, `os.environ` has a common dictionary protocol. If a given variable does not exist, item access through the `[key]` syntax will raise a `KeyError` exception. This is a common way to specify environment variables that are required and without which the application will not work.

Analogously, the `os.environ.get()` method allows you to specify environment variables that are optional or can have a default value. Using defaults is a convenient way to reduce the amount of configuration required for an individual environment. Good targets for defaults are configuration values that usually stay the same for most environments but need to be overridden in specific use cases (a testing environment, for instance). From a security standpoint, defaults should reflect production values rather than development values. That prevents accidental misconfiguration in the most critical environment. Defaults should of course never store secret values.

Last but not least, some values may need conversion to specific data types. That's because environment variable values in the `os.environ` object are always strings. If you need a specific data type that would be more useful in your code, you need to parse and transform the string value. In the previous example, we see the `BIND_PORT` value parsed to integer format and `SHEDULE_INTERVAL_SECONDS` transformed into a `timedelta` object.

If the amount of environment variables grows, it may be sensible to pack them into a common configuration object that can automate value parsing and bring more structure to the configuration. The Python standard library lacks such a feature but there are plenty of utilities on PyPI that help with handling environment variables.

One such utility is the `environ-config` package. It allows for automatic prefixing of environment variables and grouping them in descriptive sections. It offers easy validation and transformation of the values. The core of the `environ-config` package is the `environ.config()` class decorator and `environ.var()` descriptor. They are used to define configuration classes that can read values directly from the `os.environ` object. The following is a reimplementation of the previous configuration module with the usage of the `environ-config` package:

```python
from datetime import timedelta
import environ


@environ.config(prefix="")
class Config:
    @environ.config()
    class Bind:
        host = environ.var(default="localhost")
        port = environ.var(default="80", converter=int)
```

```
    bind = environ.group(Bind)
    database_uri = environ.var()
    encryption_key = environ.var()

    schedule_interval = environ.var(
        name="SCHEDULE_INTERVAL_SECONDS",
        converter=lambda value: timedelta(seconds=int(value)),
        default=50
    )
```

In order to actually create a configuration object, you can use `Config.from_environ()` as in the following example:

```
>>> config = Config.from_environ()
>>> config.bind
Config.Bind(host='localhost', port=80)
>>> config.bind.host
'localhost'
>>> config.schedule_interval
datetime.timedelta(seconds=50)
```

The configuration classes decorated with the `environ.config()` decorator will automatically look for environment variables by transforming their attribute names into uppercase. So the `config.database_uri` attribute is related directly to the `DATABASE_URI` environment variable. But sometimes you may want to use a specific name instead of an auto-generated one. You can do that easily by providing the `name` keyword argument to the `environ.var()` descriptor. We see an example of such usage in the definition of the `schedule_interval` attribute.

The definition of the `Config.Bind` class and usage of the `environ.group()` descriptor show how configurations can be nested. The `environ-config` package is smart enough to prefix requested environment variable names with the name of the group attribute. It means that the `Config.bind.host` attribute relates to the `BIND_HOST` environment variable and the `Config.bind.port` attribute relates to the `BIND_PORT` environment variable.

But the most useful feature of the `environment-config` module is the ability to conveniently handle the conversion and validation of environment variables. That can be done with the `converter` keyword argument. It can be either a type constructor as in the `Config.bind.port` example or a custom function that takes one positional string argument.

The common technique is to use one-off lambda functions as in the `Config.schedule_interval` example. Usually, the `converter` argument is just enough to ensure that the variable has the correct type and value. If that's not enough, you can provide an additional `validator` keyword argument. It should be a callable that receives the output of the `converter` function and returns the final result.

The role of environment variables in application frameworks

The role of environment variables within application frameworks that have a dedicated configuration files or modules layout can be unclear. A prime example of such frameworks is the Django framework, which comes with the popular `settings.py` module. The `settings.py` module in Django is a module of every application that contains a collection of various runtime configuration variables. It serves two purposes:

- **Statement of application structure within the framework**: Django applications are a composition of various components: apps, views, middlewares, templates, context processors, and so on. The `settings.py` file is a manifest of all installed apps, used components, and a declaration of their configuration. Most of this configuration is independent from the environment in which the application runs. In other words, it is an integral part of the application.

- **Definition of runtime configuration**: The `settings.py` module is a convenient way to provide environment-specific values that need to be accessed by application components during the application runtime. It is thus a common medium for application configuration.

Having the framework-specific statement of application structure inside the code repository of your application code is something normal. It is indeed part of the application code. Problems arise when this `settings.py` file holds explicit values for the actual environments where an application is supposed to be deployed.

The common convention among some Django developers is to define multiple settings modules to store project configuration. Those settings modules can be quite large, so usually there is one base `settings.py` file that holds common configurations and multiple per-environment modules that override specific values (see *Figure 11.1*).
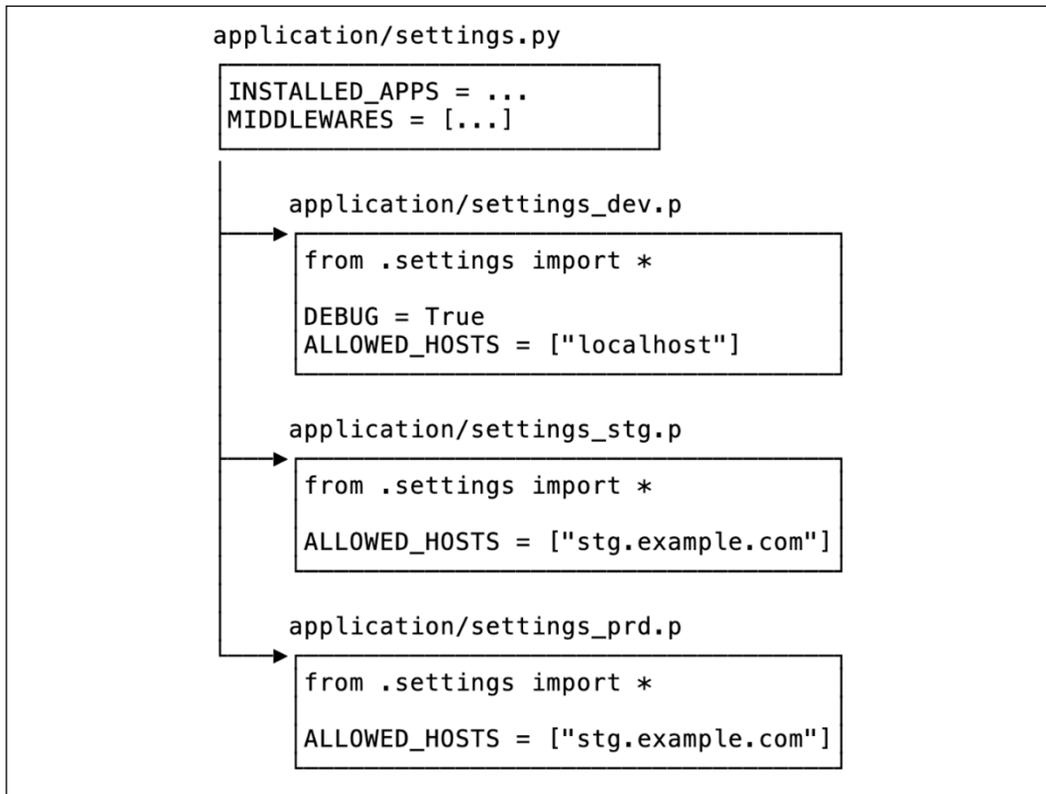
Figure 11.1: Typical layout of settings modules in many Django applications

This design is quite simple, and Django actually supports it out of the box. The Django application will read the value of the `DJANGO_SETTINGS_MODULE` environment variable on startup to decide which settings module to import. That's why this pattern is so popular.

Although using multiple per-environment settings modules is simple and popular, it has multiple drawbacks:

- **Configuration indirection**: Every settings module has to either preserve a copy of common values or import values from a shared common file. Usually, it is the latter. Then if you want to check what is the actual configuration of a specific environment, you have to read both modules.

In rare situations, developers decide to import parts of configuration between specific environments. In such situations, inspecting the configuration becomes a nightmare.

- **Adding a new environment requires code change**: Settings modules are Python code and thus will be tied to the application code. Whenever you need to create a completely new environment, you will have to modify the code.

- **Modifying configuration requires a new repackaging of application**: Whenever you modify the code for a configuration change, you need to create a new deployable artifact. A common practice in deployment methodologies is to promote every new version of an application through multiple environments. The common progression is:

  development → testing → staging → production

  With multiple settings modules, a single change for one environment configuration may necessitate redeployments in unaffected environments. This creates operational overhead.

- **A single application holds configurations for all environments**: This can pose a security risk if one environment is less secure than others. For instance, an attacker obtaining access to a development environment may gain more information about the possible attack surface in the production environment. This becomes even more problematic if secret values are stored in the configuration.

- **The problem of secret values**: Secrets should not be stored on a filesystem and definitely should not be put into the code. Django applications using per-environment settings modules usually read secrets from environment variables anyway (or communicate with dedicated password managers).

We used Django as an example of an application framework because it is extremely popular. But it's not the only framework that has the notion of settings modules and not the only framework where the pattern of multiple per-environment settings modules occurs.

Those frameworks often can't run without their settings modules. That's because settings modules are not only about environment-specific configuration but also about the composition of your application. It means that you cannot easily replace them with a set of environment variables. It would also be very inconvenient as many application-defining values often have to be provided as lists, dictionaries, or specialized data types.

But there is some middle ground. You can have an application that has a dedicated settings module but is still able to satisfy the twelve-factor rule about storing configuration in an environment. This can be achieved by following a few basic principles:

- **Use only one settings module**: A settings module should be a statement of the application structure and default behavior (timeout values, for instance) that is completely independent of the environment. In other words, if a specific value never changes between environments, you can put it safely in the settings module.

- **Use environment variables for environment-specific values**: If a value changes between environments, it can be exposed as a variable within the settings module, but it should always be read from environment variables. You can still be pragmatic and use defaults in situations where a value needs to be overridden only in very specific circumstances. An example could be a debugging flag that usually is enabled in development environments but rarely in others.

- **Use production defaults**: If a configuration variable has a default value, it is easy to miss it when configuring a specific environment. If you decide to use default values for specific configuration variables, always make sure that default values are the ones that can be safely used in a production environment. Examples of values that should be considered with great care are authentication/authorization settings or feature toggles that enable/disable experimental features. By using production defaults, you are shielding your environment from accidental misconfiguration.

- **Never put secrets into settings modules**: Secrets can be exposed as variables via a settings module (for instance, by reading them from environment variables) but should never be put there in plain-text format.

- **Do not expose an environment label to an application**: An application should be aware of its environment only through the qualities it can experience—specific configuration variables. It should never decide how to behave based on a specific label (development, staging, production, and so on) that you attach to the environment. The only acceptable use case for providing an environment label to the application is providing context to logging and telemetry utilities.

> We will talk more about logging and telemetry (including environment labeling) in *Chapter 12*, *Observing Application Behavior and Performance*.

- **Avoid per-environment .env files in your repository**: There's a common practice of writing down environment variables into so-called `.env` files. Those variables can be later exported through a shell script or read directly inside of the settings module. Simply avoid the urge to follow the practice to provide per-environment `.env` files inside of your code repository. It has all the downsides of per-environment settings modules and only increases the amount of configuration indirection.

> There's one acceptable use case for `.env` files. It is providing a configuration template for local development purposes that developers could use to quickly set up their own unique local development environment. Tools for local development like Docker Compose can understand `.env` files and export their values to an application container. Still, this practice should never be expanded to other environments. Also, it is better to use a scripting layer (or Docker Compose support) and to export `.env` files as real environment variables than to use dedicated libraries that could read those files directly from the filesystem.

The above set of principles is a pragmatic tradeoff between pure environment-based configuration and classic settings modules. Environment variables can be conveniently read through the `os.environ` object, `environ-config` package, or any other dedicated utility.

This methodology of course requires some experience in deciding which values would be environment-specific. It thus doesn't guarantee that you will never have to modify the code just to reconfigure a specific environment. The need to modify the code just for the sake of configuration change will definitely happen more often if you decide to heavily rely on default values. That's why it is usually better to avoid defaults if the value for a specific variable can be different in at least one environment.

Packaging applications that need to run on remote servers concentrates on isolation, configurability, and repeatability. Usually, we have full control over the servers and infrastructure where our code runs and can build dedicated architectures, such as container orchestration systems, to support and simplify the whole packaging process. However, things change dramatically when you are not the owner or administrator of the target environment and need your users to install or run your application themselves. This is a common case for desktop applications that are installed on users' personal computers. In such a situation, we usually build standalone executables that operate like any other standalone application. Let's see how to build such executables for Python.

Creating standalone executables

Creating standalone executables is a commonly overlooked topic in materials that cover the packaging of Python code. This is mainly because Python lacks proper tools in its standard library that could allow programmers to create simple executables that could be run by users without the need to install the Python interpreter.

Compiled languages have a big advantage over Python in that they allow you to create an executable application for the given system architecture that could be run by users in a way that does not require them to have any knowledge of the underlying technology. Python code, when distributed as a package, requires the Python interpreter in order to be run. This creates a big inconvenience for users who do not have enough technical proficiency.

Developer-friendly operating systems, such as macOS or most Linux distributions, come with the Python interpreter preinstalled. So, for their users, the Python-based application still could be distributed as a source package that relies on a specific interpreter directive in the main script file that is popularly called **shebang**. For most Python applications, this takes the following form:

```
#!/usr/bin/env python
```

Such a directive when used as the first line of script will mark it to be interpreted in the default Python version for the given environment. This can, of course, take a more detailed form that requires a specific Python version such as `python3.9`, `python3`, `python2`, and so on. Note that this will work in most popular POSIX systems but isn't portable at all. This solution relies on the existence of specific Python versions and also the availability of an `env` executable exactly at `/usr/bin/env`. Both of these assumptions may fail on some operating systems. Also, shebang will not work on Windows at all. Additionally, the bootstrapping of the Python environment on Windows can be a challenge even for developers, so you cannot expect nontechnical users to be able to do that by themselves.

The other thing to consider is the simple user experience in the desktop environment. Users usually expect applications to be run from the desktop by simply double-clicking on the executable file or the shortcut to the application. Not every desktop environment will support that with Python applications distributed in source form.

So, it would be best if we are able to create a binary distribution that would work as any other compiled executable. Fortunately, it is possible to create an executable that has both the Python interpreter and our project embedded. This allows users to open our application without caring about Python or any other dependency.

Let's see some specific use cases for standalone executables.

When standalone executables are useful

Standalone executables are useful in situations where the simplicity of the user experience is more important than the user's ability to interfere with the application's code.

Note that the fact that you are distributing applications as executables only makes code reading or modification harder, not impossible. It is not a way to secure application code and should only be used as a way to make interacting with the application simpler.

Standalone executables should be a preferred way of distributing applications for non-technical end users and also seems to be the only reasonable way of distributing any Python application for Windows.

Standalone executables are usually a good choice for the following:

- Applications that depend on specific Python versions that may not be easily available on the target operating systems
- Applications that rely on modified precompiled CPython sources
- Applications with graphical interfaces
- Projects that have many binary extensions written in different languages
- Games

Creating Python executables may not be straightforward but there are some tools that may ease the process. Let's take a look at some popular choices.

Popular tools

Python does not have any built-in support for building standalone executables. Fortunately, there are some community projects solving that problem with a varied amount of success. The following four are the most notable:

- `PyInstaller`
- `cx_Freeze`
- `py2exe`
- `py2app`

Each one of them is slightly different in use and also all have slightly different limitations. Before choosing your tool, you need to decide which platform you want to target, because every packaging tool can support only a specific set of operating systems.

It is best if you make such a decision at the very beginning of the project's life. Although none of these tools requires complex integration in your code, if you start building standalone packages early, you can automate the whole process and definitely save some future development time. If you leave this for later, you may find yourself in a situation where the project is built in such a sophisticated way that none of the available tools will work out of the box. Providing a standalone executable for such a project will be problematic and will take a lot of effort.

Let's take a look at PyInstaller in the next section.

PyInstaller

PyInstaller is by far the most advanced program to freeze Python packages into standalone executables. It provides the most extensive multiplatform compatibility among every available solution at the moment, so it is the most highly recommended one. PyInstaller supports the following platforms:

- Windows (32-bit and 64-bit)
- Linux (32-bit and 64-bit)
- macOS (32-bit and 64-bit)
- FreeBSD, Solaris, and AIX

> The documentation for PyInstaller can be found at `http://www.pyinstaller.org/`.

At the time of writing, the latest version of PyInstaller supports all Python versions from `3.5` to `3.9`. It is available on PyPI, so it can be installed in your working environment using `pip`. If you have problems installing it this way, you can always download the installer from the project's page.

Unfortunately, cross-platform building (cross-compilation) is not supported, so if you want to build your standalone executable for a specific platform, then you need to perform building on that platform. This is not a big problem today with the advent of many virtualization tools. If you don't have a specific system installed on your computer, you can always use VirtualBox or a similar system virtualization tool, which will provide you with the desired operating system as a virtual machine.

Usage for simple applications is pretty straightforward. Let's assume our application is contained in the script named `myscript.py`. This is a simple hello world application. We want to create a standalone executable for Windows users, and we have our sources located under `D://dev/app` in the filesystem. Our application can be bundled with the following short command:

```
$ pyinstaller myscript.py
```

The output you will see may be as follows:

```
2121 INFO: PyInstaller: 3.1
2121 INFO: Python: 3.9.2
2121 INFO: Platform: Windows-7-6.1.7601-SP1
2121 INFO: wrote D:\dev\app\myscript.spec
2137 INFO: UPX is not available.
2138 INFO: Extending PYTHONPATH with paths ['D:\\dev\\app', 'D:\\dev\\
app']
2138 INFO: checking Analysis
2138 INFO: Building Analysis because out00-Analysis.toc is non existent
2138 INFO: Initializing module dependency graph...
2154 INFO: Initializing module graph hooks...
2325 INFO: running Analysis out00-Analysis.toc
(...)
25884 INFO: Updating resource type 24 name 2 language 1033
```

PyInstaller's standard output is quite long, even for simple applications, so it has been truncated in the preceding example for the sake of brevity. On Windows, the resulting structure of directories and files created by PyInstaller may look as follows:

```
project/
├── myscript.py
├── myscript.spec
├──build/
│    └──myscript/
│        ├── myscript.exe
│        ├── myscript.exe.manifest
│        ├── out00-Analysis.toc
│        ├── out00-COLLECT.toc
│        ├── out00-EXE.toc
│        ├── out00-PKG.pkg
│        ├── out00-PKG.toc
│        ├── out00-PYZ.pyz
│        ├── out00-PYZ.toc
```

```
|        └── warnmyscript.txt
└──dist/
    └──myscript/
        ├── bz2.pyd
        ├── Microsoft.VC90.CRT.manifest
        ├── msvcm90.dll
        ├── msvcp90.dll
        ├── msvcr90.dll
        ├── myscript.exe
        ├── myscript.exe.manifest
        ├── python39.dll
        ├── select.pyd
        ├── unicodedata.pyd
        └── _hashlib.pyd
```

The `dist/myscript` directory contains the built application that can now be distributed to users. Note that the whole directory must be distributed. It contains all the additional files that are required to run our application (DLLs, compiled extension libraries, and so on). A more compact distribution can be obtained with the `--onefile` switch of the `pyinstaller` command as follows:

```
$ pyinstaller --onefile myscript.py
```

The resulting file structure will then look as follows:

```
project/
├── myscript.py
├── myscript.spec
├──build
│   └──myscript
│       ├── myscript.exe
│       ├── myscript.exe.manifest
│       ├── out00-Analysis.toc
│       ├── out00-COLLECT.toc
│       ├── out00-EXE.toc
│       ├── out00-PKG.pkg
│       ├── out00-PKG.toc
│       ├── out00-PYZ.pyz
│       ├── out00-PYZ.toc
│       └── warnmyscript.txt
└──dist/
    └── myscript.exe
```

When built with the `--onefile` option, the only file you need to distribute to other users is the single executable found in the `dist` directory (here, `myscript.exe`). For small applications, this is probably the preferred option.

One of the side effects of running the `pyinstaller` command is the creation of the `*.spec` file. This is an auto-generated Python module containing the specification on how to create executables from your sources. This is the example specification file created automatically for `myscript.py` code:

```python
# -*- mode: python -*-

block_cipher = None


a = Analysis(['myscript.py'],
             pathex=['D:\\dev\\app'],
             binaries=None,
             datas=None,
             hiddenimports=[],
             hookspath=[],
             runtime_hooks=[],
             excludes=[],
             win_no_prefer_redirects=False,
             win_private_assemblies=False,
             cipher=block_cipher)
pyz = PYZ(a.pure, a.zipped_data,
             cipher=block_cipher)
exe = EXE(pyz,
          a.scripts,
          a.binaries,
          a.zipfiles,
          a.datas,
          name='myscript',
          debug=False,
          strip=False,
          upx=True,
          console=True )
```

This `.spec` file contains all the `pyinstaller` arguments specified earlier. This is very useful if you have performed a lot of customizations on your build. Once created, you can use it as an argument to the `pyinstaller` command instead of your Python script as follows:

```
$ pyinstaller.exe myscript.spec
```

Note that this is a real Python module, so you can extend it and perform more complex customizations to the build procedure. Customizing the `.spec` file is especially useful when you are targeting many different platforms. Also, not all of the `pyinstaller` options are available through the command-line interface. The `.spec` file allows you to use every possible PyInstaller feature.

PyInstaller is an extensive tool, which is suitable for the great majority of programs. Anyway, thorough reading of its documentation is recommended if you are interested in using it as a tool to distribute your applications.

Let's take a look at `cx_Freeze` in the next section.

cx_Freeze

cx_Freeze is another tool for creating standalone executables. It is a simpler solution than PyInstaller, but also supports the following three major platforms:

- Windows
- Linux
- macOS

> Documentation for `cx_Freeze` can be found at `https://cx-freeze.readthedocs.io`.

At the time of writing, the latest version of `cx_Freeze` supports all Python versions from `3.6` to `3.9`. It is available on PyPI, so it can be installed in your working environment using `pip`.

Similar to PyInstaller, `cx_Freeze` does not allow you to perform cross-platform builds, so you need to create your executables on the same operating system you are distributing to. The major disadvantage of `cx_Freeze` is that it does not allow you to create real single-file executables. Applications built with it need to be distributed with related DLL files and libraries.

Let's assume that we want to package a Python application for Windows with `cx_Freeze`. The minimal example usage is very simple and requires only one command:

```
$ cxfreeze myscript.py
```

The output you will see may be as follows:

```
copying C:\Python39\lib\site-packages\cx_Freeze\bases\Console.exe ->
D:\dev\app\dist\myscript.exe
copying C:\Windows\system32\python39.dll ->
D:\dev\app\dist\python39.dll
writing zip file D:\dev\app\dist\myscript.exe
(...)
copying C:\Python39\DLLs\bz2.pyd -> D:\dev\app\dist\bz2.pyd
copying C:\Python39\DLLs\unicodedata.pyd -> D:\dev\app\dist\
unicodedata.pyd
```

The resulting structure of the files may be as follows:

```
project/
├── myscript.py
└── dist/
     ├── bz2.pyd
     ├── myscript.exe
     ├── python39.dll
     └── unicodedata.pyd
```

Instead of providing its own format for build specification (like PyInstaller does), `cx_Freeze` extends the `distutils` package. This means you can configure how your standalone executable is built with the familiar `setup.py` script. This makes `cx_Freeze` very convenient if you already distribute your package using `setuptools` or `distutils` because additional integration requires only small changes to your `setup.py` script. Here is an example of such a `setup.py` script using `cx_Freeze`. `setup()` for creating standalone executables on Windows:

```python
import sys
from cx_Freeze import setup, Executable

# Dependencies are automatically detected,
# but it might need fine tuning.
build_exe_options = {"packages": ["os"], "excludes": ["tkinter"]}

setup(
```

```
        name="myscript",
        version="0.0.1",
        description="My Hello World application!",
        options={
            "build_exe": build_exe_options
        },
        executables=[Executable("myscript.py")]
    )
```

With such a file, the new executable can be created using the new `build_exe` command added to the `setup.py` script as follows:

```
$ python setup.py build_exe
```

The usage of `cx_Freeze` may seem a bit more Pythonic than PyInstaller, thanks to the `distutils` integration. Unfortunately, this project may cause some trouble for inexperienced developers due to the following reasons:

- Installation using `pip` may be problematic under Windows
- The official documentation is very brief and lacking in some places

`cx_Freeze` is not the only tool for creating Python executables that integrates with `distutils`. Two notable examples are `py2exe` and `py2app`, which are described in the next section.

py2exe and py2app

`py2exe` (`http://www.py2exe.org/`) and `py2app` (`https://py2app.readthedocs.io/en/latest/`) are two complementary programs that integrate with Python packaging either via `distutils` or `setuptools` in order to create standalone executables. Here they are mentioned together because they are very similar in both usage and their limitations. The major drawback of `py2exe` and `py2app` is that they target only a single platform:

- `py2exe` allows building Windows executables.
- `py2app` allows building macOS apps.

> The documentation for `py2exe` can be found at `https://www.py2exe.org` and the documentation for `py2app` can be found at `https://py2app.readthedocs.io`.

Because the usage is very similar and requires only modification of the `setup.py` script, these packages complement each other. The documentation of the py2app project provides the following example of the `setup.py` script, which allows you to build standalone executables with the right tool (either py2exe or py2app) depending on the platform used:

```python
import sys
from setuptools import setup

mainscript = 'MyApplication.py'

if sys.platform == 'darwin':
    extra_options = dict(
        setup_requires=['py2app'],
        app=[mainscript],
        # Cross-platform applications generally expect sys.argv to
        # be used for opening files.
        options=dict(py2app=dict(argv_emulation=True)),
    )
elif sys.platform == 'win32':
    extra_options = dict(
        setup_requires=['py2exe'],
        app=[mainscript],
    )
else:
    extra_options = dict(
        # Normally unix-like platforms will use "setup.py install"
        # and install the main script as such
        scripts=[mainscript],
    )

setup(
    name="MyApplication",
    **extra_options
)
```

With such a script, you can build your Windows executable using the `python setup.py py2exe` command and macOS app using `python setup.py py2app`. Cross-compilation is, of course, not possible.

Despite py2app and py2exe having obvious limitations and offering less elasticity than PyInstaller or cx_Freeze, it is always good to be familiar with them. In some cases, PyInstaller or cx_Freeze might fail to build the executable for the project properly. In such situations, it is always worth checking whether other solutions can handle your code.

Security of Python code in executable packages

It is important to know that standalone executables do not make the application code secure by any means. In fact, there is no reliable way to secure applications from decompilation with the tools available today, and while it is not an easy task to decompile embedded code from executable files, it is definitely doable. What is even more important is that the results of such decompilation (if done with the proper tools) might look strikingly similar to original sources.

Still, there are some ways to make the decompilation process harder.

> It's important to note that harder does not mean less probable. For some programmers, the hardest challenges are the most tempting ones. And the eventual prize in this challenge is very high—the code that you tried to keep secret.

Usually, the process of decompilation consists of the following steps:

1. Extracting the project's binary representation of bytecode from standalone executables
2. Mapping a binary representation to the bytecode of a specific Python version
3. Translating bytecode to AST
4. Re-creating sources directly from AST

Providing the exact solutions for deterring developers from such reverse engineering of standalone executables would be pointless for obvious reasons—they will do it anyway. So here are only some ideas for hampering the decompilation process or devaluing its results:

- Removing any code metadata available at runtime (docstrings) so the eventual results will be a bit less readable.
- Modifying the bytecode values used by the CPython interpreter, so conversion from binary to bytecode, and later to AST, requires more effort.

- Using a version of CPython sources modified in such a complex way that even if decompiled sources of the application are available, they are useless without decompiling the modified CPython binary.

- Using obfuscation scripts on sources before bundling them into an executable, which will make sources less valuable after the decompilation.

Such solutions make the development process a lot harder. Some of the preceding ideas require a very deep understanding of the Python runtime, and each one of them is riddled with many pitfalls and disadvantages. Mostly, they only delay the inevitable. Once your trick is broken, it renders all your additional efforts a waste of time and effort. This fact means standalone Python executables are not a viable solution for closed-source projects where leaking of the application code could harm the organization.

The only reliable way to not allow your closed code to leak outside of your application is to not ship it directly to users in any form. And this is only possible if other aspects of your organization's security stay airtight (using strong multi-factor authentication, encrypted traffic, and a VPN to start with). So, if your whole business can be copied simply by copying the source code of your application, then you should think of other ways to distribute the application. Maybe providing software as a service would be a better choice for you.

Summary

In this chapter, we have discussed various ways of packaging Python libraries and applications including applications for SaaS/cloud environments as well as desktop applications. Now you should have a general idea about possible packaging tools and strategies for distributing your project. You should also know popular techniques for common problems and how to provide useful metadata to your project.

On our way, we've learned about the importance of the packaging ecosystem and details of publishing Python package distributions on package indexes. We've seen that standard distribution scripts (the `setup.py` files) can be useful even when not publishing code directly to PyPI.

The real fun begins when your code is made available to its users. No matter how well it is tested and how well it is designed, you will find that your application does not always behave as expected. People will report problems. You will have performance issues. Some things will inevitably go wrong.

To solve those issues, you will need a lot of information to replicate user errors and understand what has really happened. Wise developers are always prepared for the unexpected and know how to actively collect data that helps in diagnosing problems and allows you to anticipate future failures. That will be the topic of the next chapter.

12

Observing Application Behavior and Performance

With every new version of our software, we feel that **release thrill**. Did we manage to finally fix all those pesky problems we've been working on lately? Will it work or will it break? Will users be satisfied, or will they complain about new bugs or performance issues?

We usually employ various quality assurance techniques and automated testing methodologies to increase our confidence in software quality and validity. But these techniques and methodologies just increase our expectation that things will run smoothly with each new release. But how can you make sure that an application is running smoothly for your users? Or conversely, how can you know if something is going wrong?

In this chapter we will discuss the topic of **application observability**. Observability is a property of a software system that allows you to explain and understand the application's state based on its outputs. If you know the state of the system and understand how it got there, you will know if the state is correct. By employing various observability techniques, we will learn about:

- Capturing errors and logs
- Instrumenting code with custom metrics
- Distributed application tracing

Most observability techniques can be applied to both desktop applications installed on users' own computers and distributed systems running on remote servers or cloud services. However, in the case of desktop applications, observability options are often limited due to privacy concerns. That's why we will focus mainly on observing the behavior and performance of code running on your own infrastructure.

Great observability cannot be achieved without proper tools, so let's first consider the technical requirements for this chapter.

Technical requirements

The following are the Python packages that are mentioned in this chapter that you can download from PyPI:

- `freezegun`
- `sentry-sdk`
- `prometheus-client`
- `jaeger-client`
- `Flask-OpenTracing`
- `redis_opentracing`

Information on how to install packages is included in *Chapter 2, Modern Python Development Environments*.

The code files for this chapter can be found at `https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition/tree/main/Chapter%2012`.

Capturing errors and logs

Standard output is the cornerstone of observability. That's because one of the simplest things every application can do is to read information from standard input and print information to standard output. That's why the first thing every programmer learns is usually how to print "Hello world!"

Despite the fact that standard input and output are so important, users of modern software rarely know anything about their existence. Desktop applications usually aren't invoked from the terminal and users often interact with them using graphical interfaces instead.

Web-based software usually runs on remote servers and users interact with it using web browsers or custom client software. In both cases, standard input and output are hidden from the user.

But although users don't see standard output, that doesn't mean it does not exist. Standard output is often used for logging detailed information about internal application state, warnings, and errors that happen during program execution. Moreover, standard output can be easily directed to a filesystem for easy storage and later processing. The simplicity and versatility of standard output make it one of the most flexible observability enablers. It also provides the most basic way to capture and inspect information about error details.

Although you could handle application output using bare `print()` function calls, good logging requires some consistent structure and formatting. Python comes with a built-in `logging` module that provides a basic yet powerful logging system. Before we dive deeper into good practices for capturing logs and errors, let's get to grips with some of the basics of the Python logging system.

Python logging essentials

The usage of the `logging` module is pretty straightforward. The first thing to do is to create a named `logger` instance and configure the logging system:

```python
import logging

logger = logging.getLogger("my_logger")
logging.basicConfig()
```

> A common idiom when defining loggers is to use module name as the logger's name:
>
> ```python
> logger = logging.getLogger(__name__)
> ```
>
> This pattern helps in managing logger configuration in a hierarchical way. See the *Logging configuration* section for more details on configuring loggers.

Every logger is named. If you don't provide the name argument, `logging.getLogger()` will return a special "root" logger that serves as a basis for other loggers. `logging.basicConfig()` allows the specification of additional logging options like logging capture level, message formatters, and log handlers.

Using your `logger` instance, you can now record log messages at a specified log level using either the `log()` method:

```
logger.log(logging.CRITICAL, "this is critical message")
```

Or various convenience methods associated with specific log levels:

```
logger.error("This is info message")
logger.warning("This is warning message")
```

The log level is a positive integer value where greater values correspond to messages of higher importance. If `logger` is configured with a specific level, it will ignore messages of lower value. Python logging provides the following preconfigured levels with their respective integer values:

- `CRITICAL` and `FATAL` (50): Reserved for messages indicating errors due to which the program is unlikely to be able to continue its operation. A practical example would be exhaustion of resources (like disk space) or an inability to connect with a crucial backing service (a database, for instance).

- `ERROR` (40): Reserved for messages indicating serious errors that make the program unable to perform a specific task or function. A practical example would be an inability to parse user input or an intermediary network timeout.

- `WARNING` or `WARN` (30): Reserved for messages indicating abnormal situations that the program has recovered from or situations that may lead to more serious problems in the near future. A practical example would be fixing malformed user input using a fallback value or an indication of low disk space.

- `INFO` (20): Reserved for messages confirming that the program works as expected. It can be used, for instance, to output details about successful operations performed during program execution.

- `DEBUG` (10): Reserved for very detailed debugging messages, allowing us to trace application behavior during debugging sessions.

- `NOTSET` (0): A pseudo-level that captures all possible log levels.

As you can see, the default logging level values are defined in intervals of 10. This allows you to define custom levels in between existing levels if you need finer granularity.

Every predefined log level except `NOTSET` has a dedicated convenience method in the `logger` instance:

- `critical()` for the `CRITICAL` and `FATAL` levels
- `error()` or `exception()` for the `ERROR` level (the latter automatically prints a traceback of the currently captured exception)
- `warning()` for the `WARNING` and `WARN` levels
- `info()` for the `INFO` level
- `debug()` for the `DEBUG` level

The default configuration captures messages up to the `logging.WARNING` level and outputs them to standard output. The default format includes a textual representation of the error level, the logger name, and a message, as in the following example:

```
ERROR:my_logger:This is error message
WARNING:my_logger:This is warning message
CRITICAL:my_logger:This is critical message
```

The format and output of loggers can be modified by specifying two types of logging components: log handlers and formatters.

Logging system components

The Python logging system consists of four main components:

- **Loggers**: These are the entry points of the logging system. Application code uses loggers to produce messages for the logging system.
- **Handlers**: These are the receivers of the logging system. Handlers are attached to loggers and are supposed to emit information to the desired destination (usually outside of the application). One logger can have multiple handlers.
- **Filters**: These allow loggers or handlers to reject specific messages based on their contents. Filters can be attached to either loggers or handlers.
- **Formatters**: These transform raw logging messages into the desired format. Formatters can produce either human- or machine-readable messages. Formatters are attached to message handlers.

In essence, Python log messages go in one direction—from the application to the desired destination through loggers and handlers (see *Figure 12.1*). Both loggers and filters can terminate message propagation either through the mechanism of filters or by specifying the `enabled=False` flag.
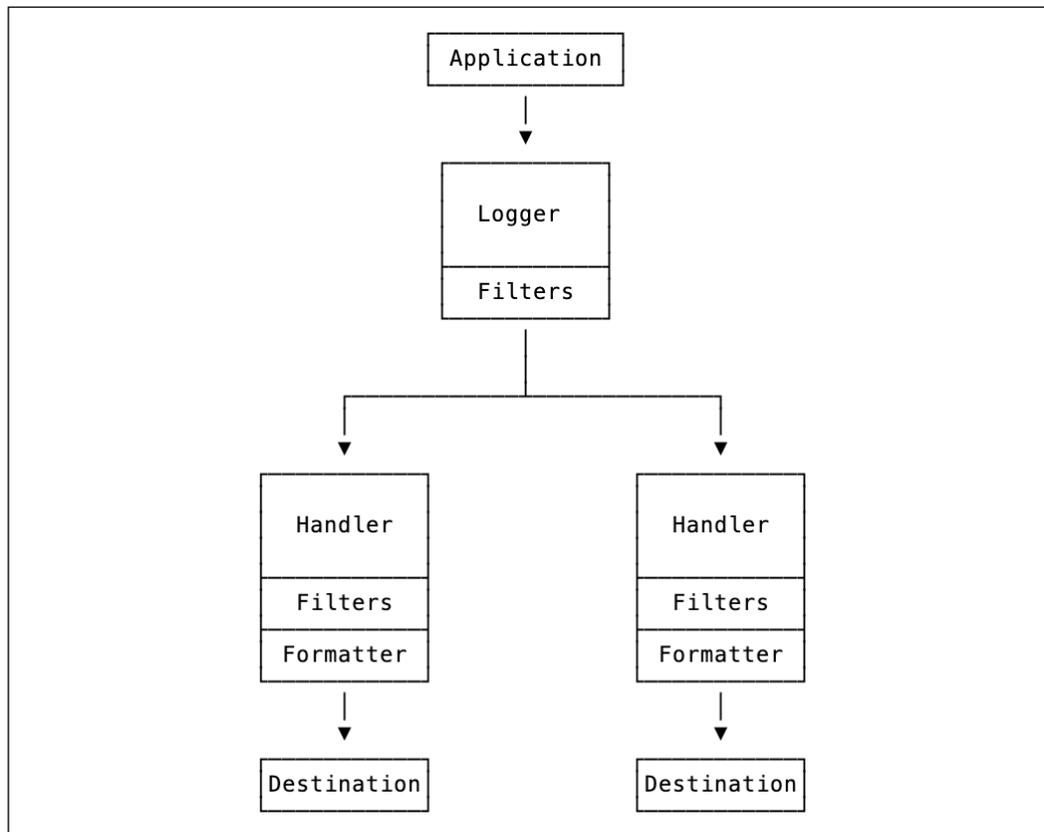
Figure 12.1: Topology of the Python logging system

The basic configuration of the logging system enabled with an empty `logging.basicConfig()` sets up the following hierarchy of components:

- **A root logger with the `logging.WARNING` level**: All messages with a severity level lower than `logging.WARNING` will be ignored by default.

- **A single console handler attached to the root logger**: This emits log messages to the standard error stream.

- **A simple formatter with `"%(levelname)s:%(name)s:%(message)s"` style**: Every message emitted to the given output will contain the severity level name, the name of the logger, and some plain message text separated by a colon.

These default settings are a good starting point. The simple format allows for easy parsing and the standard console handler will work well with every console application. However, some applications require more structured message representation and/or the handling of custom message destinations. Such customization can be achieved by overriding the log handlers and formatters.

The standard logging library comes with three built-in logging handlers:

- `NullHandler`: This handler does nothing. It can be used similarly as a `/dev/null` device on POSIX systems to discard all messages from the logger. It is often used as a default handler in libraries that use the Python logging system. In such a case, the library user is responsible for overriding the default `NullHandler` using their own logging configuration.

- `StreamHandler`: This handler emits messages to a given stream (a file-like object). If no stream is provided, `StreamHandler` will by default emit messages to the `sys.stderr` stream. After every message, the stream object will be flushed if it supports the `flush()` method.

- `FileHandler`: This handler is a subclass of `StreamHandler` that emits messages to the file specified by the `filename` argument. It will take care of opening and closing the file. By default, it uses the `append` mode of a file and default system encoding.

This is a very modest set of log handlers, but there's a `logging.handlers` module that offers over a dozen more advanced handlers. The following are the most important ones:

- `RotatingFileHandler`: This is a file handler that can rotate the logfiles whenever the current logfile exceeds a certain size limit. This is a good extension of the basic `FileHandler` that is especially useful for verbose applications that can produce a lot of log messages in a short period of time. `RotatingFileHandler` can be configured to keep only a specified number of past log backup files and thus reduces the risk of logfiles overflowing the disk.

- `TimedRotatingFileHandler`: This is similar to `RotatingFileHandler` but performs logfile rollovers at specified time intervals instead of monitoring the logfile size. It can be used to avoid disk overflow for applications that produce log messages at a rather constant and predictable pace. The advantage of `TimedRotatingFileHandler` is that past logfiles can be easily browsed by the date of their creation. The default interval of `TimedRotatingFileHandler` is one hour.

- `SysLogHandler`: This handler emits logs to the syslog server. Syslog is a popular logging standard and many Linux distributions come with a syslog server running locally by default. Many applications and services support logging to the syslog server, so `SysLogHandler` can be used to unify log collections across many programs running on the same host. Also, using syslog allows you to offload the responsibility of logfile rotation and compaction to a single system logging facility.

- `SMTPHandler`: This log handler emits a single SMTP email for every log message. It is commonly used together with the `logging.ERROR` severity level to deliver information about errors and exceptions to a specified email address. Such errors can be later reviewed at any time in the recipient's inbox. Using `SMTPHandler` is one of the simplest methods of error monitoring and allows programmers to be notified about issues even if they don't have direct access to the application runtime environment.

- `SocketHandler`, `DatagramHandler`, and `HTTPHandler`: These are simple network handlers that allow you to deliver log messages over the network. `SocketHandler` emits log messages over a socket connection, `DatagramHandler` emits logs messages as UDP datagrams, and `HTTPHandler` emits logs messages as HTTP requests. These handlers can be used to build your own custom distributed log delivery mechanism, although it will work well only for low volumes of logs that do not require good delivery guarantees. For larger log volumes, it is advised to use a specialized log delivery mechanism such as syslog or another modern distributed log system.

> We will discuss an example of a modern distributed log system in the *Distributed logging* section.

To see how one of those specialized log handlers works, let's assume we are building a small desktop application. Let's pretend our program runs with a graphical user interface and the user usually does not run it from the shell. We will be logging information about warnings and errors discovered during program execution. If there is a problem, the user will be able to find the logfile and send us a reference to it.

We don't know how many messages our application will produce. In order to avoid disk overflow, we will provide a rotating file handler. Files will be rotated daily, and we will keep a history of the last 30 days. Our logging handler configuration can be as simple as this:

```python
import logging.handlers
from datetime import timedelta, datetime


root_logger = logging.getLogger()
root_logger.setLevel(logging.INFO)
root_logger.addHandler(
    logging.handlers.TimedRotatingFileHandler(
        filename="application.log",
        when="D",
        backupCount=30,
    )
)
```

The `logging.getLogger()` call (without a specific logger name) allows us to obtain a special root logger. The purpose of this logger is to provide default configuration and handlers for other loggers that do not have specific handlers attached. If the logger does not have its own handlers, its messages will be automatically propagated to the parent logger.

> We will learn more about logger hierarchy and log message propagation in the *Logging configuration* section.

Once we have access to the root logger, we are able to provide the default configuration. The `root_logger.setLevel(logging.INFO)` call makes sure that the logger will emit only messages with a severity level greater than or equal to `logging.INFO`. That's quite a verbose setting. If you don't use custom log levels, the only more verbose setting would be `logging.DEBUG`.

We didn't configure the logging system using the `logging.basicConfig()` function so our root logger doesn't have a default handler. We add a `TimedRotatingFileHandler()` instance to the root logger using the `addHandler()` method. The `when="D"` argument specifies a daily rollover strategy and `backupCount` specifies the number of logfile backup files we want to keep on disk.

You probably don't have a full month of time to run the application just to see how rotated logfiles pile up on the disk. To speed up the process, we will use a neat trick and employ the `freezegun` package to trick Python into thinking that time is passing at an increased pace.

We used the `freezegun` package previously to test time-dependent code in *Chapter 10, Testing and Quality Automation*.

The following code simulates an application that produces one message every hour but speeds up the process by 36,000:

```python
from datetime import timedelta, datetime
import time
import logging
import freezegun

logger = logging.getLogger()

def main():
    with freezegun.freeze_time() as frozen:
        while True:
            frozen.tick(timedelta(hours=1))
            time.sleep(0.1)
            logger.info(f"Something has happened at {datetime.now()}")

if __name__ == "__main__":
    main()
```

If you run the application in your shell, you won't see any output. But if you list all the files in the current directory, you will see that after few seconds, new logfiles start appearing:

```
$ ls -al
total 264
drwxr-xr-x  35 swistakm  staff  1120  8 kwi 00:22 .
drwxr-xr-x   4 swistakm  staff   128  7 kwi 23:00 ..
-rw-r--r--   1 swistakm  staff   583  8 kwi 00:22 application.log
-rw-r--r--   1 swistakm  staff  2491  8 kwi 00:21 application.log.2021-
04-07
-rw-r--r--   1 swistakm  staff  1272  7 kwi 23:59 application.log.2021-
04-08
```

The `application.log` file is the current logfile and files ending with a date are historical backup files. If you let your program spin for a little longer, you will see that the number of historical backup files never exceeds 30. After some time, `TimedRotatingFileHandler` will start replacing old backups with new ones.

In our example, we have configured the log without any formatters. In such cases, the handler will emit messages as is and without any extra information. Here's a sample taken from the most recent logfile:

```
Something has happened at 2021-04-08 17:31:54.085117
Something has happened at 2021-04-07 23:32:04.347385
Something has happened at 2021-04-08 00:32:04.347385
Something has happened at 2021-04-08 01:32:04.347385
```

It is missing some important context. We see the date of the message only because we included it in the log message, but we are missing information about the logger from which messages are originating and information about message severity. We can customize the output from a specific log handler by attaching a custom message formatter.

Formatters are attached to log handlers using the `setFormatter()` method of the `handler` object. The formatter should be a `logging.Formatter()` instance that accepts four initialization arguments:

- `fmt`: This is a string formatting pattern for the output message. A string formatting pattern can reference any of the attributes of the `logging.LogRecord` class. It defaults to `None`, which is a plain text log message without any formatting.

- `datefmt`: This is a date formatting pattern for representing the message timestamp. It accepts the same formatting directives as the `time.strftime()` function. It defaults to `None`, which translates to an ISO8601-like format.

- `style`: This specifies the string formatting style used for the `fmt` argument. It can be either `'%'` (percent formatting), `'{'` (str.format() formatting), or `'$'` (`string.Template` formatting). It defaults to `'%'`.

- `validate`: This specifies whether to validate the `fmt` formatting argument against the `style` argument. It defaults to `True`.

We can, for instance, provide a custom formatter that includes the severity message time, level name, logger name, and exact line of the log call along with the emitted message. Our logging setup would then look as follows:

```python
root_logger = logging.getLogger()
root_logger.setLevel(logging.INFO)
formatter = logging.Formatter(
    fmt=(
        "%(asctime)s | %(levelname)s | "
        "%(name)s | %(filename)s:%(lineno)d | "
        "%(message)s"
```

```
    )
)
handler = logging.handlers.TimedRotatingFileHandler(
    filename="application.log",
    when="D",
    backupCount=30,
)
handler.setFormatter(formatter)
root_logger.addHandler(handler)
```

> You can learn more about the available `LogRecord` attributes
> that can be referenced with custom formatters at `https://`
> `docs.python.org/3/library/logging.html#logrecord-`
> `attributes`.

With such a configuration, a message in our logfile would look as follows:

```
2021-04-08 02:03:50,780 | INFO | __main__ | logging_handlers.py:34 |
Something has happened at 2021-04-08 00:03:50.780432
```

The choice of whether to use a specialized log handler usually depends on multiple factors, like the target operating system, the existing system logging infrastructure in your organization, the expected volume of logs, and the deployment methodology. Standalone applications, system services, or Docker containers usually log differently, and you need to take that into account. For instance, having historical logfiles easily accessible is definitely a practical thing, but managing a collection of thousands of logfiles in a large distributed system consisting of tens or hundreds of hosts can be a real hurdle. On the other hand, you can't expect end users of standalone applications to run a distributed logging system on their own personal computer just to use your application.

The same applies to log message formatting. Some log collection and processing systems can take advantage of structured message formats such as JSON, msgpack, or avro. Others will be able to parse and extract semantic elements of the message using customized parsing rules. Plaintext messages are easier to inspect and understand for humans but are harder to process using specialized log analysis software. Structured log messages are easier to process by machines but are harder to read with the naked eye.

Regardless of your current needs, you can be sure that your logging choices and preferences will change over time. That's why providing a logging setup by manually creating handlers and formatters is rarely a convenient method.

We've already learned about the `logging.basicConfig()` function. It sets up a reasonable logging default but also allows you to provide some shortcuts for defining the message format or the default handler selection. Unfortunately, it works only at the root logger level and doesn't allow you to target other loggers. In the next section we will learn about alternative logging configuration methods that allow you to define complex logging rules for applications of any size.

Logging configuration

Using the root logger is a convenient method of defining the top-level configuration of your application logging system. Unfortunately, you will quickly learn that some libraries often use the Python `logging` module to emit information about important events happening inside them. At some point, you may find it necessary to fine-tune the logging behavior of those libraries as well.

Usually, libraries use their own logger names. The common pattern for naming loggers is to use a special __name__ attribute that contains the fully qualified name of a module:

```python
import logging

logger = logging.getLogger(__name__)
```

For instance, the __name__ attribute inside the `utils` sub-module of the package `acme` will be `acme.utils`. If `acme.utils` defines its logger as `logging.getLogger(__name__)` then the name of that logger will be `'acme.utils'`.

When you know the name of the logger, you can obtain it at any time and provide a custom configuration. Common use cases are:

- **Silencing the logger**: Sometimes you are not interested in messages coming from a library at all. In your main application file, you can find a logger and silence it with the `disabled` attribute as in the following example:

  ```python
  acme_logger = logging.getLogger("acme.utils")
  acme_logger.disabled = True
  ```

- **Overriding handlers**: Libraries should not define their own handlers; that should be the responsibility of the library user. Anyway, not every programmer is aware of good logging practices and it happens sometimes that a useful third-party package comes with a logger that has an attached handler. It will effectively ignore your root logger configuration.

You can use such logger and override handlers as in the following example:

```
acme_logger = logging.getLogger("acme.utils")
acme_logger.handlers.clear()
```

- **Changing logger verbosity**: It may happen that some loggers are too verbose. Sometimes warnings inside a library are not severe enough to be included in the main application log. You can find a logger and override its severity level:

```
acme_logger = logging.getLogger("acme.utils")
acme_logger.setLevel(logging.CRITICAL)
```

Using per-module loggers is also useful in non-library code. It is good practice to use per-module loggers in larger applications consisting of multiple sub-packages or sub-modules. This will allow you to easily fine-tune the verbosity and manage the handlers of multiple loggers. And the key in consistently managing Python logging configuration is understanding the parent-child relationship of loggers.

Whenever you create a logger with a name that contains periods, the `logging` module will actually build a hierarchy of loggers. If you, for instance, try to obtain a logger using `logging.getLogger("acme.lib.utils")`, the `logging` module will perform the following steps:

1. It will first search for a logger registered under `"acme.lib.utils"` in a thread-safe manner. If it does not exist, it will create a fresh logger and register it under `"acme.lib.utils"`.

2. If a new logger has been created, it will iteratively remove the last segments from the logger name and search for a new name until it finds a registered logger. If the name doesn't have a logger registered under it, the `logging` module will register special placeholder object. For `"acme.lib.utils"`, it will search first for `"acme.lib"` and then for `"acme"`. The first non-placeholder logger will become the parent of `"acme.lib.utils"`.

3. If there are no non-placeholder loggers above the newly created logger, the root logger becomes the parent of this logger.

Moreover, the `logging` module ensures that existing placeholders are actively replaced with proper loggers the first time they are explicitly accessed with the `logging.getLogger()` function. In such a case, the parent-child relationships are retroactively updated. Thanks to this, concrete loggers can be configured in any order and regardless of their hierarchy.

This parent-child relationship plays a role in the evaluation of logger handlers. Whenever you log a new message through a specific logger, handlers are invoked according to the following rules:

1. If a logger has its own handlers, a message is passed to every handler:
    -   If the logger has the `propagate` attribute set to `True` (default value), the message is propagated to the parent logger.
    -   If the logger has the `propagate` attribute set to `False`, processing stops.

2. If a logger does not have its own handlers, a message is passed to the parent logger

Best practice is to define log handlers only at the top-level root logger. Otherwise, it will be hard to track all propagation rules and make sure that every message is logged only once. But sometimes specifying handlers on lower-level (per-module) loggers can be useful for applying special treatment to a very specific category of errors. For instance, your application may generally log using the default console handler, but if there is a business-critical module, you may want to attach `SMTPHandler`. That way you will be sure that all log messages coming from that module will be additionally delivered to you as SMTP emails.

The logging hierarchy is also useful for controlling the verbosity of whole logger groups. For example, if the `acme` package contains multiple sub-loggers and none of them has handlers attached, you can disable the whole `"acme"` logger to silence every sub-logger.

Complex hierarchies can be intimidating, especially if you need to perform fine-tuning of loggers living in multiple modules. Creating individual loggers, handlers, and formatters using Python may sometimes require a substantial amount of boilerplate. That's why the `logging.config` module offers two functions that allow you to configure the whole Python logging system in a more declarative way:

-   `fileConfig()`: This takes the path of an INI-like configuration file. The syntax of that file is the same as the syntax of configuration files handled with the built-in `configparser` module.

-   `dictConfig()`: This takes a dictionary of configuration values.

> You can find more information about Python logging configuration options at https://docs.python.org/3/library/logging.config.html.

Both configuration ways assume similar configuration sections and options. The only difference is the format. The following is an example of the configuration file for logging with time-based file rotation:

```
[formatters]
keys=default

[loggers]
keys=root

[handlers]
keys=logfile

[logger_root]
handlers=logfile
level=INFO

[formatter_default]
format=%(asctime)s | %(levelname)s | %(name)s | %(filename)s:%(lineno)d
| %(message)s

[handler_logfile]
class=logging.handlers.TimedRotatingFileHandler
formatter=default
kwargs={"filename": "application.log", "when": "D", "backupCount": 30}
```

And the following is the same configuration defined with use of the `dictConfig()` function:

```
logging.config.dictConfig({
    "version": 1,
    "formatters": {
        "default": {
            "format": (
                "%(asctime)s | %(levelname)s | "
                "%(name)s | %(filename)s:%(lineno)d | "
                "%(message)s"
            )
        },
    },
    "handlers": {
        "logfile": {
            "class": "logging.handlers.TimedRotatingFileHandler",
            "formatter": "default",
            "filename": "application.log",
```

```
            "when": "D",
            "backupCount": 30,
        }
    },
    "root": {
        "handlers": ["logfile"],
        "level": "INFO",
    }
})
```

With so many formatting options and a hierarchical logger structure, logging in Python can be quite complex. But it rarely has to be complex. You can reduce the complexity of logging by following some good practices explained in the next section.

Good logging practices

Python provides a flexible and powerful logging system. With so many configuration options, it is really easy to get tangled in unnecessary complexity. But in fact, logging should be as simple as possible. It is, in the end, the first and foremost tool to use to understand how your application works. You won't be able to understand how your code works if you're spending countless hours trying to understand what's wrong with your logging.

The key to good logging is to follow good practices. The following are common rules for efficient and simple logging in Python:

- **Use per-module loggers**: By using `logging.getLogger(__name__)` to create new loggers, you make it simple to control the logging behavior of whole module trees. This practice is especially useful for libraries because it allows developers to tune library logging from within the main application logging configuration.

- **Use one event per line**: If you use text-based outputs (stdout/stderr streams, files), a single log message should preferably fit a single line of text. This allows you to avoid most text buffering issues and the mixing of messages coming from multiple threads or processes logging to the same output. If you really need to fit multiline text into a message (for instance, for an error stack trace), you can still do that by leveraging structured log formats like JSON.

- **Offload work to system logging facilities or a distributed logging system**: Log compaction, forwarding, or logfile rotation can all be done in Python code using custom log handlers, but that is rarely a wise choice. It is usually better to use a system logging facility like syslog or a dedicated log processing tool like logstash or fluentd. Dedicated tools will usually do these things better and more consistently.

They will also remove a lot of log processing complexity from your own code and will reduce problems with logging in concurrent applications.

- **If possible, log directly to standard output or error streams**: Writing to stdout or stderr is one of the most basic things every application can do. This is true for writing to files, but not every environment will have a writable or persistent filesystem. If you need logs stored in files, you can simply use shell output redirection. If you need your logs delivered over the network, you can use dedicated log forwarders like logstash or fluentd.

- **Keep log handlers at the root logger level**: Defining handlers and formatters for specific loggers other than the root logger makes a logging configuration unnecessarily complex. It is usually better to define a single console handler at the root level and leave the rest of the log processing logic to external logging utilities.

- **Avoid the urge to write a custom distributed logging system**: Reliably delivering logs over a network in a large distributed system isn't a simple thing. It is usually better to leave this task to dedicated tools like syslog, fluentd, or logstash.

- **Use structured log messages**: As the volume of logs grows, it becomes harder and harder to extract meaningful information from them. Dedicated log processing systems allow you to search through large volumes of textual information using various queries and even do sophisticated analytics of historical log events. Most of these systems can perform efficient parsing of plain textual log messages, but they always work better if they have access to structured log messages from the very beginning. Common structured message formats for logs are JSON and msgpack.

> Python's `logging` and `logging.handlers` modules lack a dedicated handler for structured log messages. `python-json-logger` is a popular package from PyPI that provides a formatter capable of emitting messages in JSON format. You can learn more about `python-json-logger` at `https://github.com/madzak/python-json-logger`.
>
> Another popular package is `structlog`. It extends the Python logging system with various utilities for capturing log context, processing messages, and outputting them in various structured formats. You can learn more about the `structlog` package at `https://www.structlog.org/en/stable/index.html`.

- **Configure logging in only one place**: Logging configuration (understood as a collection of handlers and formatters) should be defined only in one part of your application. Preferably, that's the main application entrypoint script, like the __main__.py file or a WSGI/ASGI application module (for web-based applications).

- **Use** basicConfig() **whenever possible**: If you follow most of the previous rules, the basicConfig() function is all that you need to perform a complete logging configuration. It works at the root logger level and by default defines the StreamHandler class attached to the stderr stream. It also allows for the convenient configuration of date and log message format strings and usually, you won't need anything more beyond that. If you want to use structured log messages you can easily use your own handler with a custom message formatter.

- **Prefer** dictConfig() **over** fileConfig(): The syntax of logging configuration files supported by fileConfig() is a bit clunky and way less flexible than dictConfig(). It is, for instance, a common pattern to control the verbosity of application logging through command-line arguments or environment variables. Such functionality is definitely easier to implement through a dictionary-based logging configuration than through a file-based configuration.

It may seem like a lot of rules, but generally, they are all about keeping things simple and not overengineering your logging system. If you keep the configuration simple, log to standard output, and use reasonable log formatting, you are usually set for success.

Good logging hygiene will prepare you for handling and processing arbitrarily large volumes of logs. Not every application will generate a lot of logs, but those that do generally need the support of dedicated log forwarding and processing systems. These systems are usually capable of handling distributed logs coming from hundreds or even thousands of independent hosts.

Distributed logging

With a single service or program running on a single host, you can easily go with a simple logging setup based on rotated logfiles. Almost every process supervision tool (like systemd or supervisord) allows for the redirection of stdout and stderr output to a specified logfile, so you don't have to open any files in your application. You also can easily offload the responsibility of compacting and rotating historical logfiles to a system utility like logrotate.

Simple things will work well only for small applications. If you run multiple services on the same host, you will eventually want to handle logs of various programs in a similar way. The first step to organize logging chaos is to use a dedicated system logging facility like syslog. It will not only digest logs in a consistent format but will also provide command-line utilities to browse and filter past logs.

Things become more complex once you scale your application to run in a distributed manner across many hosts. The main challenges of distributed logging are the following:

- **Large volumes of logs**: Distributed systems can produce enormous amounts of logs. The volume of logs grows with the number of processing nodes (hosts). If you want to store historical logs, you will need a dedicated infrastructure to keep all this data.

- **The need for centralized log access**: It is really impractical to store logs only on the hosts that produce them. When you know that something is going wrong with your application, the last thing you want to do is to jump from host to host to find a logfile that has crucial information about the issue you're experiencing. What you need is a single place where you can access all logs available in your system.

- **Unreliable networks**: Networks are unreliable by nature, and if you are sending data over one, you need to be prepared for temporary connection failures, network congestion, or unexpected communication delays, which may happen in any distributed system. With centralized log access, you will need to deliver logs from each individual host to a dedicated logging infrastructure. To avoid any data loss, you will need specialized software that can buffer log messages and retry delivery in the case of network failures.

- **Information discoverability and correlation**: With large volumes of logs, it will be hard to find useful information in the sea of log messages of various severity. You will eventually need a way to perform custom queries against your log dataset that allows you to filter messages by their sources, logger names, severity levels, and textual content. Also, issues occurring in distributed systems are often very intricate. In order to understand what is really going on in your system, you will need tools that allow you to aggregate information from various log streams and perform statistical data analysis.

The complexity of the logging infrastructure depends mainly on the scale of your application and the observability needs you have. Depending on the capabilities of the logging infrastructure, we can define the following logging maturity model:

- **Level 0 (snowflake)**: Every application is a unique "snowflake." Each handles all extra logging activities, like storage, compaction, archiving, and retention, on its own. Level 0 offers almost no extra observability capabilities beyond the ability to open historical logs in a text editor or shell utility.

- **Level 1 (unified logging)**: Every service on the same host logs messages in a similar way to a system daemon, known destination, or disk. Basic log processing tasks like compaction or retention are delegated to system logging facilities or common utilities like logrotate. To view logs from a particular host, the developer needs to "shell into" the host or communicate with the dedicated service daemon running on that host.

- **Level 2 (centralized logging)**: Logs from every host are delivered to a dedicated host or a cluster of hosts to prevent data loss and for archival storage. Services may need to deliver logs to a specific destination (a logging daemon or a location on disk). It is possible to view a slice of all logs in a given time frame, sometimes with advanced filtering, aggregation, and analytics capabilities.

- **Level 3 (logging mesh)**: Services are completely unaware of the logging infrastructure and can log directly to their stdout or stderr streams. A logging mesh working on every host is capable of automatically discovering new log streams and/or logfiles from every running service. Upon the deployment of a new application, its output streams will be automatically included in the logging infrastructure. A centralized log access infrastructure is capable of performing sophisticated queries over arbitrarily large time spans. It is possible to run log aggregation and analytics on stored messages. Logging information is available in near real time.

Centralized logging and logging mesh infrastructures can offer similar capabilities with regards to log filtering, aggregation, and analytics. The key differentiator here is the ubiquity of logging in logging mesh infrastructures. In Level 3 logging architectures, everything that runs your infrastructure and outputs information on standard output or error streams is automatically a source of logging information and will be automatically available in a centralized log access system.

There are many companies offering Level 2 or Level 3 logging infrastructures as a service. A popular example is AWS CloudWatch, which is well integrated with other AWS services. Other cloud providers also offer alternative solutions. If you have enough time and determination, you can also build a full-fledged Level 2 or Level 3 logging infrastructure from scratch using open-source tools.

> A popular open-source choice for building full-fledged logging infrastructures is the Elastic Stack. This is a software ecosystem composed of multiple components: Logstash/Beats (log and metric ingestors/collectors), Elasticsearch (document store and search system), and Kibana (frontend application and dashboard for the Elastic Stack). You can learn more about the Elastic Stack at `https://www.elastic.co/`.

Capturing errors for later review

Logging is the most popular choice for storing and discovering information about errors and issues happening in the system. Mature logging infrastructures offer advanced correlation capabilities that allow you to have more insight into what is happening in the whole system at any given time. They can often be configured to alert on error message occurrence, which enables fast response times in situations of unexpected failure.

But the main problem of tracking errors using ordinary logging infrastructures is that they provide only a limited overview of the full context of the error occurrence. Simply put, you will only have access to the information that was included in a logging call.

In order to fully understand why a particular problem has occurred, you will need more information than was initially included in the logging call. With a traditional logging infrastructure, you would have to modify the code with additional logging, release the new version of the application, and wait for the error to occur again. If you missed something, you would have to repeat the process over and over again.

Another problem is the filtering of meaningful errors. It is not uncommon for large distributed systems to log tens of thousands of error messages or warnings daily. But not every error message has to be acted upon immediately. Teams doing maintenance often perform a bug triaging process to estimate the impact of an error. Usually, you concentrate on issues that occur very often or those that happen in a critical part of your application. For efficient triaging, you will need a system that is at least capable of deduplicating (merging) error events and assessing their frequency. Not every logging infrastructure will be able to do that efficiently.

That's why for tracking errors in your application, you should usually use a dedicated error tracking system that can work independently from your logging infrastructure.

One of the popular tools that gives a great error tracking experience is Sentry. It is a battle-tested service for tracking exceptions and collecting crash reports. It is available as open source, written in Python, and originated as a tool for backend Python developers. Now, it has outgrown its initial ambitions and has support for many more languages, including PHP, Ruby, and JavaScript, Go, Java, C, C++, and Kotlin. It still remains one of the most popular error tracking tools for many Python web developers.

> You can learn more about Sentry at `https://sentry.io/`.

Sentry is available as a paid software-as-a-service model, but it is also an open-source project, so it can be hosted for free on your own infrastructure. The library that provides integration with Sentry is `sentry-sdk` (available on PyPI). If you haven't worked with it yet and want to test it without having to run your own Sentry server, then you can easily sign up for a free trial at the Sentry service site. Once you have access to a Sentry server and have created a new project, you will obtain a string called a **Data Source Name** (**DSN**) string. This DSN string is the minimal configuration setting needed to integrate your application with Sentry. It contains the protocol, credentials, server location, and your organization/project identifier in the following format:

```
'{PROTOCOL}://{PUBLIC_KEY}:{SECRET_KEY}@{HOST}/{PATH}{PROJECT_ID}'
```

Once you have your DSN, the integration is pretty straightforward, as shown in the following code:

```python
import sentry_sdk

sentry_sdk.init(
    dsn='https://<key>:<secret>@app.getsentry.com/<project>'
)
```

From now on, every unhandled exception in your code will be automatically delivered to the Sentry server using the Sentry API. The Sentry SDK uses the HTTP protocol and delivers errors as compressed JSON messages over a secure HTTP connection (HTTPS). By default, messages are sent asynchronously from a separate thread to limit the impact on application performance. Error messages can then be later reviewed in the Sentry portal.

Error capture happens automatically on every unhandled exception, but you can also explicitly capture exceptions as in the following example:

```python
try:
    1 / 0
except Exception as e:
    sentry_sdk.capture_exception(e)
```

The Sentry SDK has numerous integrations with the most popular Python frameworks, such as Django, Flask, Celery, and Pyramid. These integrations will automatically provide additional context that is specific to the given framework. If your web framework of choice does not have dedicated support, the `sentry-sdk` package provides generic WSGI middleware that makes it compatible with any WSGI-based web server, as shown in the following code:

```python
from sentry_sdk.integrations.wsgi import SentryWsgiMiddleware

sentry_sdk.init(
    dsn='https://<key>:<secret>@app.getsentry.com/<project>'
)
# ...
application = SentryWsgiMiddleware(application)
```

> **Exceptions in Python web applications**
>
> Commonly, web applications do not exit on unhandled exceptions because HTTP servers are obliged to return an error response with a status code from the 5XX group if any server error occurs. Most Python web frameworks do such things by default. In such cases, the exception is, in fact, handled either on the internal web framework level or by the WSGI server middleware. Anyway, this will usually still result in the exception stack trace being printed (usually on standard output). The Sentry SDK is aware of the WSGI conventions and will automatically capture such exceptions as well.

The other notable integration is the ability to track messages logged through Python's built-in `logging` module. Enabling such support requires only the following few additional lines of code:

```python
import logging

import sentry_sdk
from sentry_sdk.integrations.logging import LoggingIntegration
```

```python
sentry_logging = LoggingIntegration(
    level=logging.INFO,
    event_level=logging.ERROR,
)

sentry_sdk.init(
    dsn='https://<key>:<secret>@app.getsentry.com/<project>',
    integrations=[sentry_logging],
)
```

The capturing of logging messages may have caveats, so make sure to read the official documentation on the topic if you are interested in such a feature. This should save you from unpleasant surprises.

If you decide to use Sentry or any other similar service, carefully consider the "build versus buy" decision. As you may already heard, "there ain't no such thing as a free lunch." If you decide to run an error tracking service (or any utility system) on your own infrastructure, you will eventually pay additional infrastructure costs. Such an additional system will also be just another service to maintain and update. Maintenance = additional work = costs!

As your application grows, the number of exceptions grows, so you will be forced to scale Sentry (or any other system) as you scale your product. Fortunately, Sentry is a very robust project, but it will not give you any value if it's overwhelmed with too much load. Also, keeping Sentry prepared for a catastrophic failure scenario, where thousands of crash reports can be sent per second, is a real challenge. So, you must decide which option is really cheaper for you and whether you have enough resources to do all of this by yourself.

> There is, of course, no "build versus buy" dilemma if security policies in your organization deny sending any data to third parties. If so, just host Sentry or similar software on your own infrastructure. There are costs, of course, but they are ones that are definitely worth paying.

Capturing errors allows for later review, eases debugging, and allows you to respond quickly in situations when errors suddenly start to accumulate. But this is a reactive approach in situations when users have likely been exposed to buggy software.

Mindful developers want to observe their applications all the time and be able to react before actual failures occur. This type of application observability can be implemented by instrumenting your code with custom metrics. Let's learn about various types of custom metrics and systems used for metrics collection.

Instrumenting code with custom metrics

If we want to keep our application running smoothly, we need to be proactive. Observability isn't only about being able to do post-mortem analysis of logs and error reports. It is also about collecting various metrics that provide insights about service load, performance, and resource usage. If you monitor how your application behaves during normal operation, you will be able to spot anomalies and anticipate failures before they happen.

The key in monitoring software is defining metrics that will be useful in determining the general service health. Typical metrics can be divided into a few categories:

- **Resource usage metrics**: Typical metrics are memory, disk, network, and CPU time usage. You should always monitor these metrics because every infrastructure has limited resources. That's true even for cloud services, which provide seemingly unlimited resource pools. If one service has abnormal resource usage, it can starve other services and even induce cascading failures in your infrastructure. This is especially important when running code on cloud infrastructure where you often pay for the resources that you use. A resource-hungry service that goes rogue can cost you a lot of money.

- **Load metrics**: Typical metrics are the number of connections and the number of requests in a given time frame. Most services can accept a limited number of parallel connections and their performance degrades after reaching a specific threshold. The usual pattern for overloaded services is a gradual performance drop followed by sudden unavailability after reaching the critical load point. Monitoring load metrics allows deciding whether you need to scale out or not. Low load periods can also be an opportunity to scale your infrastructure in to reduce operating costs.

- **Performance metrics**: Typical metrics are request or task processing times. Performance metrics are often correlated with load metrics but can also highlight performance hotspots in code that needs optimization. Good application performance improves user experience and allows you to save money on infrastructure costs because performant code will likely need fewer resources to run. The continuous monitoring of performance allows discovering performance regressions that can happen when introducing new changes in an application.

- **Business metrics**: These are the key performance metrics of your business. Typical examples might be the number of signups or sold items in a given time frame. Anomalies in these metrics allow you to discover functional regressions that slipped through the testing process (a defective cart checkout process, for instance) or evaluate dubious changes in the application interface that may confuse frequent users.

Some of those metrics can sometimes be derived from application logs, and many mature logging infrastructures can perform rolling aggregations on log event streams. This practice works best for load metrics and performance metrics, which often can be reliably derived from the access logs of web servers, proxies, or load balancers. The same approach can be used also for business metrics, although that requires careful consideration of log formats for key business events and can be very brittle.

Log processing has the least utility in the case of resource usage metrics. That's because resource metrics are based on regular probing of resource usage, and log processing is concerned with streams of discrete events that often happen on a rather irregular basis. Also, centralized log infrastructures are not well suited for storing time series data.

Usually, to monitor application metrics, we use dedicated infrastructure that is independent of logging infrastructure. Advanced metric monitoring systems offer complex metric aggregation and correlation capabilities and are focused on dealing with time series data. Metric infrastructure systems also offer faster access to the most recent data because they are more concerned with observing live systems than logging infrastructures. With logging infrastructures, it is more common to observe noticeable lag in data propagation.

There are generally two architectures of metric monitoring systems:

- **Push architectures**: In this architecture, the application is responsible for pushing data to the system. It is usually a remote metrics server or local metric daemon (metric forwarder) responsible for pushing metrics to a higher-level daemon on a different server (layered architecture). The configuration is usually distributed: each service needs to know the location of the destination daemon or server.

- **Pull architectures**: In this architecture, the application is responsible for exposing a metrics endpoint (usually an HTTP endpoint) and the metric daemon or server pulls information from known services. The configuration can be either centralized (where the main metric server knows the locations of monitored services) or semi-distributed (the main server knows the location of metric forwarders that pull metrics from lower layers).

Both types of architectures can have mesh-like capabilities by utilizing a service discovery mechanism. For instance, in push architectures, metric forwarders can advertise themselves as services capable of forwarding metrics to other services. In pull architectures, monitored services usually advertise themselves in the service discovery catalog as services that provide metrics to be collected.

One of the popular choices for monitoring applications is **Prometheus**.

Using Prometheus

Prometheus is a prime example of pull-based metrics infrastructure. It is a complete system capable of collecting metrics, defining forwarding topologies (through so-called metric exporters), data visualization, and alerting. It comes with an SDK for multiple programming languages, including Python.

The architecture of Prometheus (presented in *Figure 12.2*) consists of the following components:

- **Prometheus server**: This is a standalone metrics server that is capable of storing time series data, responding to metrics queries, and retrieving metrics (pulling) from metrics exporters, monitored services (jobs), and other Prometheus servers. A Prometheus server provides an HTTP API for querying data and includes a simple interface that allows us to create various metrics visualizations.

- **Alert manager**: This is an optional service that is able to store the definition of alerting rules and trigger notifications when specific alerting conditions are met.

- **Metric exporters**: These are the processes that the Prometheus server can pull data from. An exporter can be any service that uses the Prometheus SDK to expose a metrics endpoint. There are also standalone exporters that can probe information directly from hosts (like general host usage information), expose metrics for services without Prometheus SDK integration (like databases), or act as a push gateway for short-lived processes (like cron jobs). Every Prometheus server can act as a metrics exporter for other Prometheus servers.

The above three components are the bare minimum for having a fully functional metrics infrastructure with data visualizations and reliable alerting. Moreover, many Prometheus deployments extend this architecture with the following extra components:

- **Service discovery catalog**: A Prometheus server is capable of reading information from various service discovery solutions to locate available metrics exporters. The usage of service discovery mechanisms simplifies configuration and allows for a mesh-like developer experience. Popular choices for service discovery catalogs are Consul, ZooKeeper, and etcd. Container orchestration systems (like Kubernetes) often have built-in service discovery mechanisms.

- **Dashboarding solution**: Prometheus servers have a simple web interface capable of performing simple data visualizations. These basic capabilities are often not enough for operations teams. Thanks to its open API, Prometheus can easily be extended with a custom dashboarding solution. The most popular choice is Grafana, which can also be easily integrated with other metrics systems and data sources.
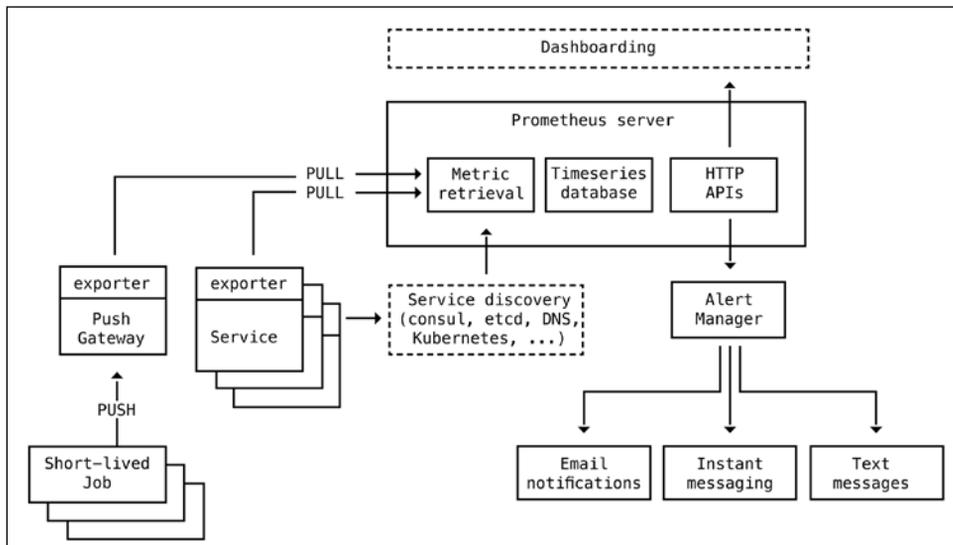
Figure 12.2: Architecture of a typical Prometheus deployment

To see how applications can be easily extended with Prometheus monitoring, we will take one of the applications written in previous chapters and try to furnish it with the Prometheus SDK. We will also define a small Docker Compose setup that will allow us to evaluate the whole solution locally.

For evaluating metrics systems, it is essential to have an application that actually does something useful. One practical example that served us well in various parts of the book was the pixel tracking service from *Chapter 5*, *Interfaces, Patterns, and Modularity*. You should be familiar with how it works, so we will use it as the basis of our experiment.

The heart of our application is the `tracking.py` file. It includes module imports, definitions of views, HTTP route bindings, and an instance of a Flask application object. If we skip the view functions, it looks roughly as follows:

```python
from flask import Flask, request, Response
from flask_injector import FlaskInjector

from interfaces import ViewsStorageBackend
import di

app = Flask(__name__)

@app.route("/track")
def track(storage: ViewsStorageBackend):
    ...

@app.route("/stats")
def stats(storage: ViewsStorageBackend):
    ...

@app.route("/test")
def test():
    ...

if __name__ == "__main__":
    FlaskInjector(app=app, modules=[di.RedisModule()])
    app.run(host="0.0.0.0", port=8000)
```

> For the sake of brevity, we are omitting parts of the application code that won't deal directly with metrics. Full code examples of the pixel tracking service can be found in *Chapter 5, Interfaces, Patterns, and Modularity*. You can also find all the source code for this section (including the configuration, `Dockerfile`, and `docker-compose.yml`) in the `Chapter 12/05 - Using Prometheus` directory of the code repository for this book (see the *Technical requirements* section).

In order to make our application observable by the Prometheus server, we need to embed it with a Prometheus metrics exporter. We will do that using the official `prometheus-client` package available on PyPI. We start by defining some metric objects using classes from the `prometheus_client` module:

```python
from prometheus_client import Summary, Gauge, Info

REQUEST_TIME = Summary(
    "request_processing_seconds",
    "Time spent processing requests"
)
AVERAGE_TOP_HITS = Gauge(
    "average_top_hits",
    "Average number of top-10 page counts "
)
TOP_PAGE = Info(
    "top_page",
    "Most popular referrer"
)
```

REQUEST_TIME is a Summary metric that can be used to track the size and number of events. We will use it to track the time and number of processed requests.

AVERAGE_TOP_HITS is a Gauge metric that can be used to track how a single value changes over time. We will use it to track the average number of hit counts of the 10 most popular pages.

TOP_PAGE is an Info metric that can be used to expose any textual information. We will use it to track the most popular pixel-tracked page at any given time.

Individual metric values can be updated using various code patterns. One of the more popular ways is to use specific metric methods as decorators. This is the most convenient way to measure time spent in a function. We will use this method to track the REQUEST_TIME metric as in the following example:

```python
@app.route("/track")
@REQUEST_TIME.time()
def track(storage: ViewsStorageBackend):
    ...


@app.route("/stats")
@REQUEST_TIME.time()
def stats(storage: ViewsStorageBackend):
    ...


@app.route("/test")
@REQUEST_TIME.time()
def test():
    ...
```

Another way is to use specific metric object methods as normal function calls. This is a common technique for counters, gauges, and info metrics. We will use this pattern to track the values of the `AVERAGE_TOP_HITS` and `TOP_PAGE` metrics. We can get a quite good estimate by inspecting the values of the `stats()` view function:

```python
@app.route("/stats")
@REQUEST_TIME.time()
def stats(storage: ViewsStorageBackend):
    counts: dict[str, int] = storage.most_common(10)

    AVERAGE_TOP_HITS.set(
        sum(counts.values()) / len(counts) if counts else 0
    )
    TOP_PAGE.info({
        "top": max(counts, default="n/a", key=lambda x: counts[x])
    })

    return counts
```

When the metrics are defined, we can finally embed the metric exporter. This can be done either by starting a separate metrics thread using the `prometheus_client.start_http_server()` function or using specific integration handlers. `prometheus-client` comes with nice Flask support through Werkzeug's `DispatcherMiddleware` class.

> Werkzeug is a toolkit for creating web applications based on the WSGI interface. Flask is built using Werkzeug, so it is compatible with Werkzeug middlewares. You can learn more about Werkzeug at `https://palletsprojects.com/p/werkzeug/`.

We will use exactly this solution:

```python
from prometheus_client import make_wsgi_app
from werkzeug.middleware.dispatcher import DispatcherMiddleware
app.wsgi_app = DispatcherMiddleware(app.wsgi_app, {
    '/metrics': make_wsgi_app()
})

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000)
```

> `prometheus-client` uses thread-safe in-memory storage for metric objects. That's why it works best with a threaded concurrency model (see *Chapter 6*, *Concurrency*). Due to CPython threading implementation details (mainly Global Interpreter Lock), multiprocessing is definitely a more popular concurrency model for web applications. It is possible to use `prometheus-client` in multiprocessing applications, although it requires slightly more setup. For details, see the official client documentation at `https://github.com/prometheus/client_python`.

When we start our application and visit `http://localhost:8000/test` and `http://localhost:8000/stats` in a web browser, we will automatically populate metric values. If you the visit the metrics endpoint at `http://localhost:8000/metrics`, you will see output that may look as follows:

```
# HELP python_gc_objects_collected_total Objects collected during gc
# TYPE python_gc_objects_collected_total counter
python_gc_objects_collected_total{generation="0"} 595.0
python_gc_objects_collected_total{generation="1"} 0.0
python_gc_objects_collected_total{generation="2"} 0.0
# HELP python_info Python platform information
# TYPE python_info gauge
python_info{implementation="CPython",major="3",minor="9",patchlevel="0"
,version="3.9.0"} 1.0
# HELP process_virtual_memory_bytes Virtual memory size in bytes.
# TYPE process_virtual_memory_bytes gauge
process_virtual_memory_bytes 1.88428288e+08
# HELP process_cpu_seconds_total Total user and system CPU time spent
in seconds.
# TYPE process_cpu_seconds_total counter
process_cpu_seconds_total 0.13999999999999999
# HELP process_open_fds Number of open file descriptors.
# TYPE process_open_fds gauge
process_open_fds 7.0
# HELP request_processing_seconds Time spent processing requests
# TYPE request_processing_seconds summary
request_processing_seconds_count 1.0
request_processing_seconds_sum 0.0015633490111213177
# HELP request_processing_seconds_created Time spent processing
requests
# TYPE request_processing_seconds_created gauge
request_processing_seconds_created 1.6180166638851087e+09
```

```
# HELP average_top_hits Average number of top-10 page counts
# TYPE average_top_hits gauge
average_top_hits 6.0
# HELP top_page_info Most popular referrer
# TYPE top_page_info gauge
top_page_info{top="http://localhost:8000/test"} 1.0
```

As you can see, the output is a human- and machine-readable representation of the current metric values. Besides our custom metrics, it also includes useful default metrics regarding garbage collection and resource usage. These are exactly the same metrics that the Prometheus server will pull from our service.

Now it's time to set up the Prometheus server. We will use Docker Compose so you won't have to install it manually on your own development host. Our `docker-compose.yml` file will include the definition of three services:

```yaml
version: "3.7"

services:
  app:
    build:
      context: .
    ports:
      - 8000:8000
    volumes:
      - ".:/app/"

  redis:
    image: redis

  prometheus:
    image: prom/prometheus:v2.15.2
    volumes:
      - ./prometheus/:/etc/prometheus/
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.path=/prometheus'
      - '--web.console.libraries=/usr/share/prometheus/console_
libraries'
      - '--web.console.templates=/usr/share/prometheus/consoles'
    ports:
      - 9090:9090
    restart: always
```

The `app` service is our main application container. We will build it from a local `Dockerfile` containing the following code:

```
FROM python:3.9-slim
WORKDIR app

RUN pip install \
 Flask==1.1.2 \
 redis==3.5.3 \
 Flask_Injector==0.12.3 \
 prometheus-client==0.10.1


ADD *.py ./
CMD python3 tracking.py –reload
```

The `redis` service is a container running the Redis data store. It is used by our pixel-tracking application to store information about visited page counts.

Last but not least is the `prometheus` service with our Prometheus server container. We override the default `prometheus` image command to provide custom configuration locations. We need a custom configuration file mounted as a Docker volume to tell Prometheus about our metrics exporter location. Without configuration, the Prometheus server won't know where to pull metrics from. We don't have a service discovery catalog, so we will use a simple static configuration:

```
global:
  scrape_interval:     15s
  evaluation_interval: 15s

  external_labels:
      monitor: 'compose'

scrape_configs:
  - job_name: 'prometheus'
    scrape_interval: 5s
    static_configs:
      - targets: ['localhost:9090']

  - job_name: 'app'
    scrape_interval: 5s
    static_configs:
      - targets: ["app:8000"]
```

We can start the whole solution using the `docker-compose` command as follows:

```
$ docker-compose up
```

> The initial `docker-compose` run may take a little longer because Docker will have to download or build images that do not exist on your filesystem yet.

When all services are running, you will notice in the `docker-compose` output that the Prometheus server is asking the `app` service for metrics every few seconds:

```
app_1 | 172.21.0.3 - - [10/Apr/2021 01:49:09] "GET /metrics HTTP/1.1"
200 -
app_1 | 172.21.0.3 - - [10/Apr/2021 01:49:14] "GET /metrics HTTP/1.1"
200 -
app_1 | 172.21.0.3 - - [10/Apr/2021 01:49:19] "GET /metrics HTTP/1.1"
200 -
app_1 | 172.21.0.3 - - [10/Apr/2021 01:49:24] "GET /metrics HTTP/1.1"
200 -
```

You can access the Prometheus server web interface at `http://localhost:9090`. It allows you to browse registered services and perform simple metrics visualization (as in *Figure 12.3*). The following is an example query that will give you an average response time per request over a 5-minute window:

```
rate(request_processing_seconds_sum[5m])
/ rate(request_processing_seconds_count[5m])
```

The results, displayed in Prometheus, will look similar to the following:
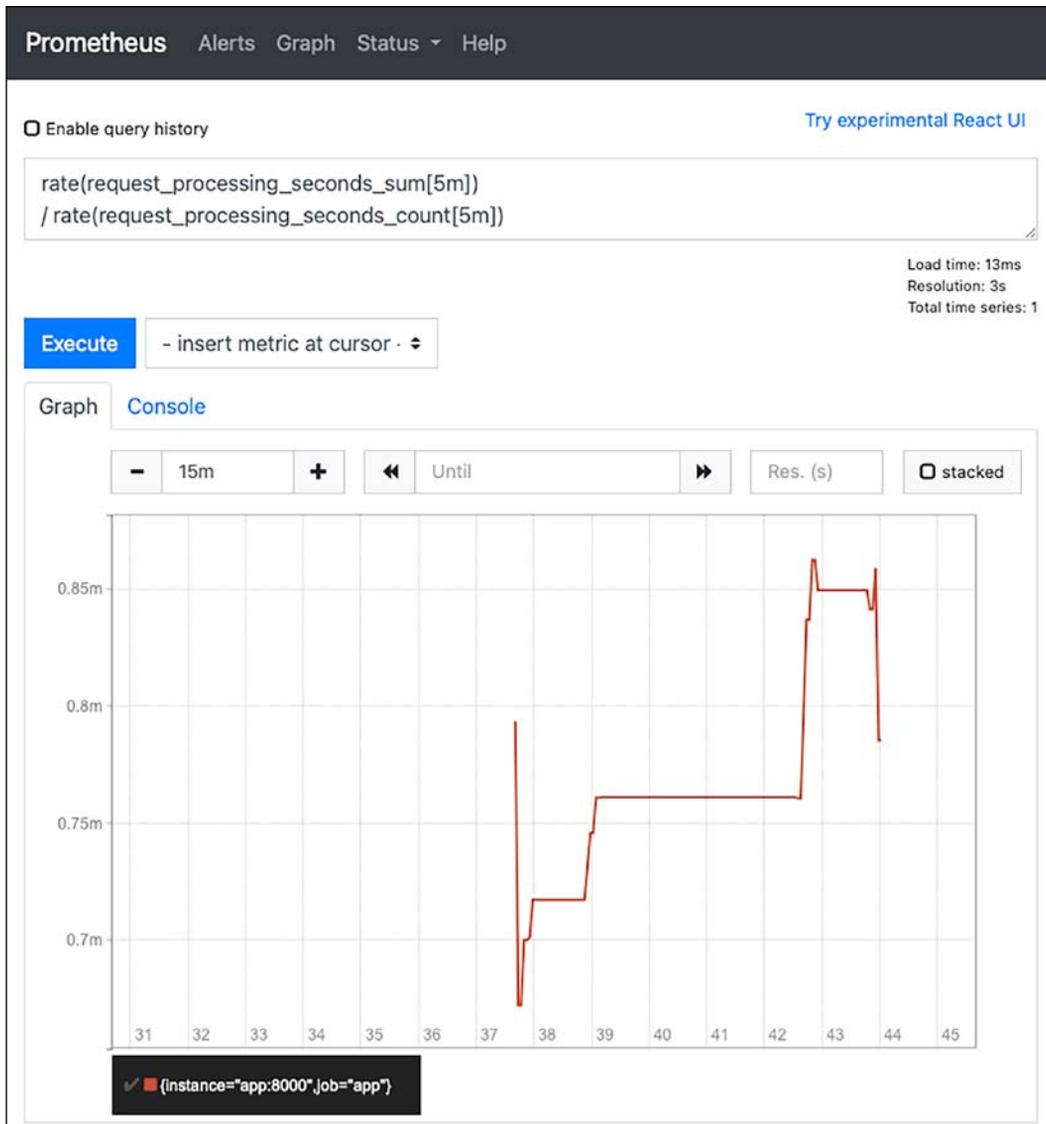
Figure 12.3: Example of visualizations in the Prometheus server web interface

As you can see, the Prometheus UI can present plots of your custom metrics. It can give an overview of how your application behavior changes over time. It can be a useful tool in spotting anomalies or ensuring that your application meets the desired performance levels. Give it a try and test a few queries. Try to use application endpoints and see how the data is reflected in your queries.

Using metrics allows you to have better knowledge of how your application works in production environments. It can also provide you with insight into how well it fulfills its purpose by tracking business metrics. Advanced queries allow you to correlate metrics of various types and from various services. This is especially useful in distributed systems, which can have many moving parts.

But using a metrics infrastructure isn't the only way to have better visibility into a distributed system. A completely different approach is to measure each individual transaction and the relationships between related transactions happening in various system components. This methodology is called **distributed tracing** and it tries to connect elements of logging and traditional metrics infrastructures.

Distributed application tracing

Distributed and networked systems are inevitable in any large-scale infrastructure. That's because you can put only a limited amount of hardware in a single server rack. If you have to serve a large number of users concurrently, you will have to scale out eventually. Also, having all of your software running on a single host will be risky from a reliability and availability standpoint. If one machine fails, the whole system goes down. And it isn't that uncommon to see even whole datacenters being brought down due to natural disasters or other unpredictable events. This means that highly available systems are often forced to be spread over multiple datacenters located in different regions or even managed by different providers, just to ensure enough redundancy.

Another reason for distributed systems is splitting infrastructures into independent domain services in order to split large codebases and enable multiple teams to work efficiently without stepping on each others' toes. That allows you to reduce elaborate change management processes, simplify deployments of large systems, and limit the amount of orchestration between independent development teams.

But distributed systems are hard. Networks are unreliable and introduce delays. Unnecessary communication roundtrips add up with every service-to-service hop and can have a noticeable impact on application performance. Also, every new host increases operational costs and the failure surface. Organizations with large and distributed architectures usually have dedicated teams responsible for keeping them healthy.

This is also true for clients of cloud services like AWS, Azure, or Google Cloud Platform, as the complexity of these cloud environments requires skilled and trained professionals that know how to harness the configuration in an auditable and maintainable way.

But the real nightmare of distributed systems (from a developer's perspective) is observability and debugging. If you have many applications and services running on multiple hosts, it is really hard to get a good overview of what's actually happening inside of your system. A single web transaction can, for instance, go through several layers of a distributed architecture (web balancers, caching proxies, backend APIs, queues, databases) and involve a dozen independent services in the process (see *Figure 12.4*):
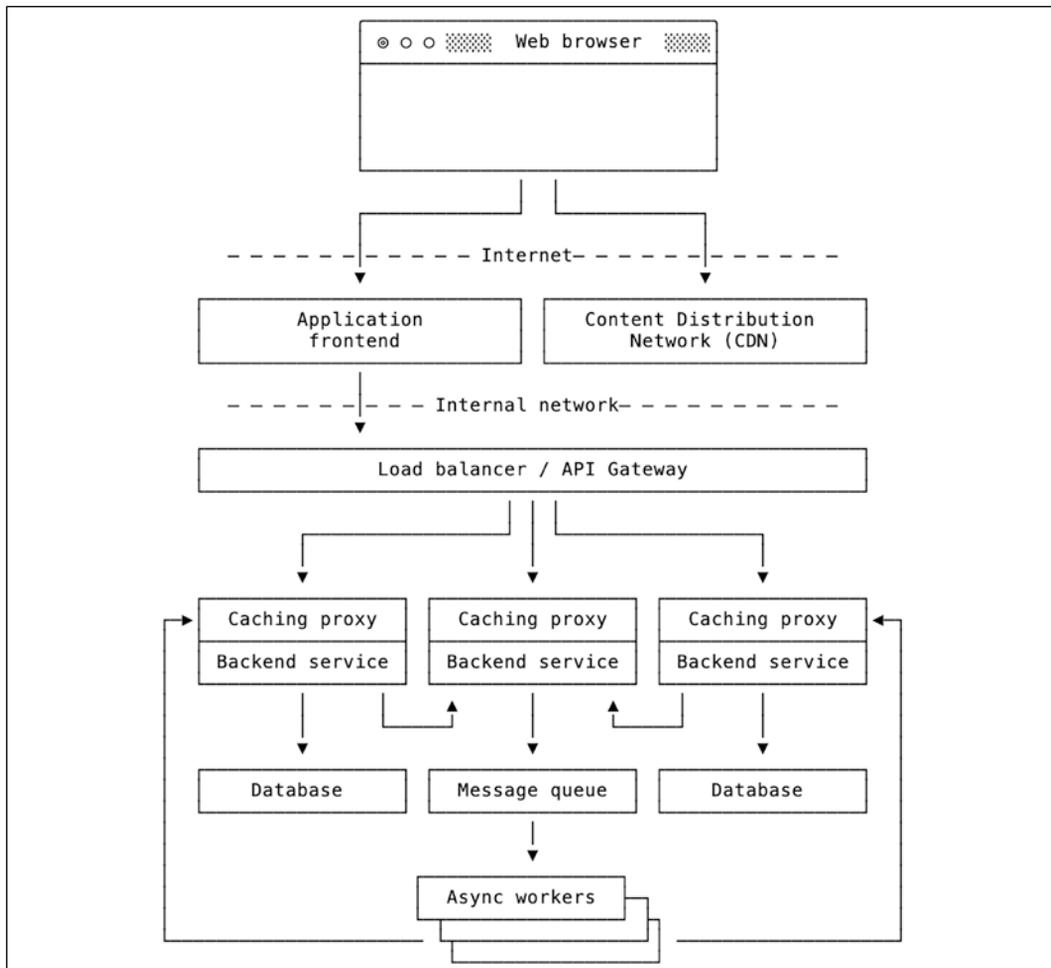
Figure 12.4: Example of an elaborate distributed application architecture with many inter-service connections

With architectures that complex, it is hard to know what the real source of the user's problems is. Logging and error tracking via dedicated systems can help to discover unhandled exceptions but doesn't help that much if erroneous behavior does not produce any observable notifications. Also, errors in a single service can cascade into errors in dependent services. Classic error tracking and logging solutions may produce a lot of noise when such a cascade happens. It will then be harder to track the real root cause of a problem.

When it comes to understanding performance issues, classic metrics systems can highlight obvious performance hotspots but usually do so on a single service level. Common reasons for performance problems in distributed systems are inefficient service usage patterns and not the isolated performance characteristics of a single service. Consider for instance the following example of a distributed transaction from an imaginary eCommerce website:

1. The user requests data about an order summary from Checkout Service.

2. Checkout Service retrieves the user session identifier from the request headers and asks Authentication Service to verify if it is valid:

    - Authentication Service verifies the session identifier against a list of active sessions in the relational database acting as a session store.

    - If the session is valid, Authentication Service returns a successful response.

3. If the session is valid, Checkout Service retrieves the list of items in the user's cart that is stored in another relational database.

4. For every entry in the cart, Checkout Service asks Inventory Service if there are enough items in the warehouse to fulfill the order. To ensure authorized access, it forwards the user's session identifier in the request header:

    - For every request, Inventory Service asks Authentication Service to validate the session identifier.

    - If the session is valid, Inventory Service checks the warehouse inventory state for the requested item type in its own database and returns the result.

5. If the order can be fulfilled, Checkout Service makes a series of requests to Pricing Service to obtain the current item prices and tax charges for every entry in the cart:

    - For every request, Checkout Service asks Authentication Service to validate the session identifier.

- If the session is valid, Pricing Service checks the warehouse inventory state for the requested item type in its own database and returns the result.

6. Checkout Service then returns a complete order summary for the user's cart contents.

Do you see the pattern there? There is a lot of redundancy, network hopping, and potential sources of problems. And there is definitely a lot of room for improvement, like reducing unnecessary requests to Authentication Service or batching queries to dependent services.

It is debatable whether distributed applications should be designed in such a granular way. Distributed architecture has its merits, like scalability on an individual service level, distributed code ownership, and (usually) faster release processes, but introduces a whole new class of problems to deal with. Unfortunately, with the advent of microservice architectures, inter-service communication usually gets out of hand quickly. Moreover, with independent teams working on separate services it is not uncommon to see sudden performance regressions due to additional communication links.

Traditional observability techniques like collecting logs and per-service metrics may not be good enough to provide complete insight into a distributed system's operation. What we usually need instead is the ability to track whole user interactions as atomic distributed transactions that span across many services and many hosts within the whole system. This technique is called **distributed tracing** and often combines the features of traditional logging and metrics solutions.

As with logging and metric collection, there are good paid SaaS solutions for providing whole-stack distributed tracing capabilities for architectures of virtually any size. These may be pricy, but there are also good open-source solutions for building distributed tracing infrastructures on your own. One of the more popular open-source implementations of distributed tracing is Jaeger.

> As with every other observability solution, you need to carefully evaluate the "build versus buy" factors when deciding to use such a system. There are high-quality, open-source distributed tracing solutions available, but they all require some expertise and maintaining your own infrastructure. Hiring skilled professionals costs money and hardware doesn't come for free either. Depending on your scale and needs, it may actually be cheaper to pay some other company to run and maintain the whole distributed tracing infrastructure for you.

Distributed tracing with Jaeger

Many distributed tracing solutions are based on the **OpenTracing** standard. OpenTracing is an open protocol and a set of libraries for various programming languages that can be used to instrument code with the ability to send transaction traces to an OpenTracing compatible server. Jaeger is one of the most popular implementations of such servers.

> **OpenTelemetry** is a new standard that is supposed to supersede the **OpenTracing** (an open protocol for tracing collection) and **OpenCensus** (an open protocol for metrics and traces collection) standards. OpenTelementry is a protocol that is backward-compatible with former ones and future versions of Jaeger are expected to support the OpenTelemetry standard as well.

The key concept of the OpenTracing standard is the **span**. A span is a building block of a distributed transaction trace that represents a logical operation within the transaction. Every trace consists of one or more spans, and spans can reference other spans in a nested fashion. Every span contains the following information:

- The name of the operation that the span represents
- A pair of timestamps that define the start and end of a span
- A set of span tags that allow you to query, filter, and analyze traces
- A set of span logs that are specific to the operation within the span
- A span context that is a container for cross-process information carried over between spans

In distributed systems, spans usually represent complete request-response cycles within a single service. OpenTracing libraries allow you also to easily define smaller spans that can be used to track smaller logical blocks of processing like database queries, file access requests, or individual function calls.

OpenTracing provides the `opentracing-python` package on PyPI, but we can't use it to interact with the Jaeger server. It is a reference implementation that is just an empty shell to be extended by actual implementations of OpenTracing. For instance, for Jaeger users, we will use the official `jaeger-client` package. Instrumenting code with `jaeger-client` is really simple.

> Despite the need to use implementation-specific libraries like `jaeger-client`, the OpenTracing architecture is designed in a way that makes it very easy to migrate between various implementations.

It would be great if we had some distributed service to evaluate distributed tracing with Jaeger, but our simple pixel-tracking application will do the job too. It maintains connections to the Redis server and is complex enough to give us the opportunity to create our own custom spans.

The integration with `jaeger-client` starts with initializing the `tracer` object:

```python
from jaeger_client import Config

tracer = Config(
    config={
        'sampler': {
            'type': 'const',
            'param': 1,
        },
    },
    service_name="pixel-tracking",
).initialize_tracer()
```

The `'sampler'` section in the tracer config defines the event sampling strategy. Our setting uses a constant sampling strategy with a value of `1`. This means that all transactions will be reported to the Jaeger server.

> Jaeger provides multiple event sampling strategies. The correct sampling strategy depends on the expected service load and the scale of the Jaeger deployment. You can read more about Jaeger sampling at `https://www.jaegertracing.io/docs/1.22/sampling/`.

Every configuration should be created with the `service_name` argument. This allows Jaeger to tag and identify spans coming from the same service and allows for better traces. In our case, we've set `service_name` to `"pixel-tracking"`.

Once we have an instance of `tracer`, we can start defining spans. The most convenient way to do this is through the context manager syntax as in the following example:

```python
@app.route("/stats")
def stats(storage: ViewsStorageBackend):
    with tracer.start_span("storage-query"):
        return storage.most_common(10)
```

Applications created with the use of web frameworks usually have multiple request handlers, and you usually want to track all of them. Instrumenting every request handler manually would be unproductive and error prone. That's why it is generally better to use a framework-specific OpenTracing integration that will automate this process. For Flask we can use the `Flask-Opentracing` package from PyPI. You can enable this integration by simply creating a `FlaskTracing` class instance in your main application module:

```python
from flask import Flask
from flask_opentracing import FlaskTracing
from jaeger_client import Config

app = Flask(__name__)

tracer = Config(
    config={'sampler': {'type': 'const', 'param': 1}},
    service_name="pixel-tracking",
).initialize_tracer()

FlaskTracing(tracer, app=app)
```

Another useful technique is to enable automatic integration for libraries that we use to communicate with external services, databases, and storage engines. That way, we will be able to track outgoing transactions and OpenTracing will build a relationship between spans coming from various services.

In our example, we have only one service, so there is no opportunity to correlate distributed spans. But we use Redis as a datastore, so we could at least instrument queries made to Redis. There's a dedicated package on PyPI for that purpose named `redis_opentracing`. To enable this integration, you need to perform only a single function call:

```python
import redis_opentracing

redis_opentracing.init_tracing(tracer)
```

> You can find all the source code for this section (including the configuration, `Dockerfile`, and `docker-compose.yml`) in the `Chapter 12/ Distributed tracing with Jaeger` directory of code repository for this book (see the *Technical requirements* section).

`redis_opentracing` is a new package, so we will have to modify our Dockerfile as follows:

```
FROM python:3.9-slim
WORKDIR app

RUN pip install \
 Flask==1.1.2 \
 redis==3.5.3 \
 Flask_Injector==0.12.3 \
 prometheus-client==0.10.1 \
 jaeger-client==4.4.0 \
 opentracing==2.4.0 \
 Flask-OpenTracing==1.1.0

RUN pip install --no-deps redis_opentracing==1.0.0


ADD *.py ./
CMD python3 tracking.py --reload
```

> Note that we've installed `redis_opentracing` with `pip install --no-deps`. This tells `pip` to ignore package dependencies. Unfortunately, at the time of writing, `install_requires` of `redis_opentracing` does not list `opentracing=2.4.0` as a supported package version, although it works with it pretty well. Our approach is a trick to ignore this dependency conflict. Hopefully, the new release of `redis_opentracing` will resolve this issue.

Last but not least, we need to start the Jaeger server. We can do that locally using Docker Compose. We will use the following `docker-compose.yml` file to start our pixel-tracking application, Redis server, and Jaeger server:

```
version: "3.7"

services:
  app:
    build:
      context: .
    ports:
      - 8000:8000
```

```
    environment:
      - JAEGER_AGENT_HOST=jaeger
    volumes:
      - ".:/app/"

  redis:
    image: redis

  jaeger:
    image: jaegertracing/all-in-one:latest
    ports:
      - "6831:6831/udp"
      - "16686:16686"
```

The Jaeger server (similarly to the Prometheus server) consists of a few components. The `jaegertracing/all-in-one:latest` Docker image is a convenient packaging of all those components for simple deployments or for local experimentation. Note that we used the `JAEGER_AGENT_HOST` environment variable to tell the Jaeger client about the location of our Jaeger server. This is a pretty common pattern among observability SDKs that allows for the easy swapping of solutions without affecting the configuration of the monitored application.

Once we have everything set up, we can start the whole solution using Docker Compose:

```
$ docker-compose up
```

> The initial `docker-compose` run may take a little longer because Docker will have to download or build non-existent images.

When all services are running, you can visit `http://localhost:8000/test` and `http://localhost:8000/stats` in your browser to generate some traces. You can visit `http://localhost:16686/` to access the web interface of the Jaeger server and browse the collected traces.

The example trace, displayed in Jaeger, may look similar to the following:
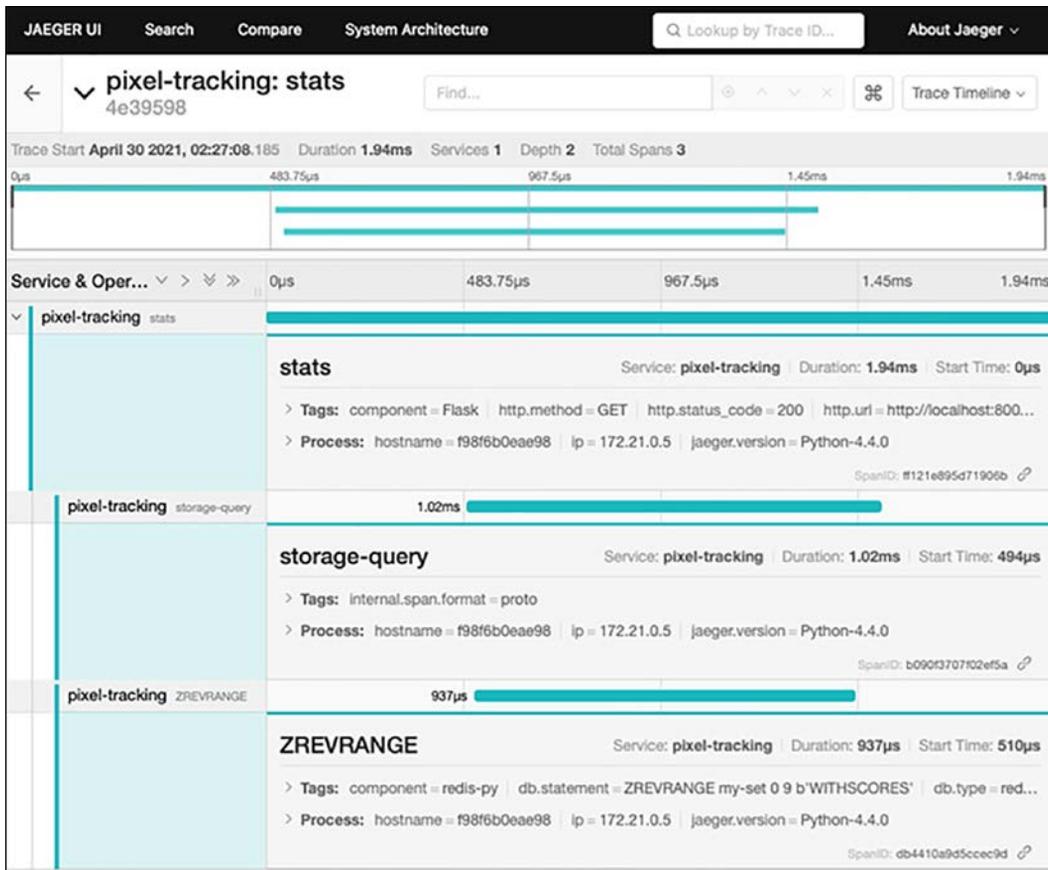
Figure 12.5: Example visualization of a distributed trace in the Jaeger web interface
with multiple nested spans visible

As you can see, the `pixel-tracing: stats` transaction consists of three spans:

- `stats`: This is the topmost span of the `stats()` endpoint, added automatically by the FlaskTracing middleware.
- `storage-query`: This is the middle span, added manually using the `tracer.start_span("storage-query")` context manager.
- `ZVREVRANGE`: This is the span added automatically by the `redis_opentracing` integration.

Every trace comes with a precise time measurement and additional tags like IP address, library version, or span-specific information (database statement, HTTP response code, and so on). These elements are indispensable in pinpointing performance issues and understanding communication patterns between components of distributed systems.

Summary

We've discussed the three main observability techniques of modern applications: logging, metrics collection, and distributed tracing. All of them have their advantages, but logging is definitely the most important way of collecting information from your application. That's because it is simple, does not require any special infrastructure (although it is good to have one), and is least likely to fail you.

But logging has some limitations. Unstructured logging messages make it harder to extract insights from logs. Logging is also not suitable for the periodic probing of information about resource usage and performance. It is good for auditing purposes and post-mortem analysis after major failures but is rarely helpful in tracking current information and reacting to sudden events.

That's why metrics collection systems are a natural and valuable extension of logging infrastructures. They allow you to collect information in real time, create custom metrics, and create powerful operational dashboards that can be actively monitored by operations teams. They can also be used to some extent to provide an overview of business metrics, although larger organizations usually prefer dedicated analytic software. Metrics systems are indispensable in monitoring application performance.

Last but not least, there are distributed tracing infrastructures, which are great in complex systems with many moving pieces. These are indispensable in situations where issues arise at the meeting points between multiple communicating services. They give a great overview of the information flow within the system and make it easy to spot communication anomalies.

Each type of observability solution provides a useful view of what's going on within your system but no single one solves all of your problems. It would be best if we could have all of them, but unfortunately, not every organization can afford to build or buy all of these solutions. If you have constrained resources and need to pick only one, you should definitely consider logging first, because it will usually have the greatest return on your investment.

When you have an efficient observability solution in place, you will start discovering intricate problems and performance issues. Some of them will really surprise you. In the end, the best way to evaluate software is to run it in the desired environment and under real-life load. You will usually notice that most of your performance issues are concentrated in isolated hotspots. Once you have identified them, it's time for detailed performance analysis and optimization—the topic of our next chapter.

13

Code Optimization

Code optimization is the process of making an application work more efficiently, usually without modifying its functionality and accuracy. Code optimization is usually concerned with the speed of processing (CPU time optimization), but can also be used to minimize the usage of different resources, such as memory, disk space, or network bandwidth.

In the previous chapter, we learned about various observability techniques that can be used to identify performance hotspots in applications. Logging, metric monitoring, and tracing can be used to create a general performance overview and prioritize optimization efforts. Moreover, operations teams often use custom alerts of performance and resource usage metrics or log messages related to operation timeouts to determine components that require immediate action.

But even the best logging, metrics, and tracing systems will give you only a rough overview of the performance problem. If you decide to fix it, you will have to perform a careful profiling process that will uncover detailed resource usage patterns.

There are plenty of optimization techniques. Some concentrate on algorithmic complexity and the use of optimized data structures. Others can trade the accuracy or consistency of results for quick and easy performance gains. Only when you know how your application works and how it uses resources will you be able to apply the proper optimization technique.

In this chapter, we will discuss the following topics:

- Common culprits for bad performance
- Code profiling
- Reducing complexity by choosing appropriate data structures
- Leveraging architectural trade-offs

Python comes with built-in profiling capabilities and useful optimization utilities, so theoretically, we could use nothing other than Python and its standard library. But extra tools are like meta-optimization—they increase your own efficiency in finding and solving performance issues. That's why we will rely heavily on some extra utilities. Let's consider them.

Technical requirements

The following are the Python packages that are mentioned in this chapter that you can download from PyPI:

- `gprof2dot`
- `objgraph`
- `pymemcache`

Information on how to install packages is included in *Chapter 2*, *Modern Python Development Environments*.

The following are some extra applications that you may need to install:

- **Graphviz**: An open source diagram visualization solution available at `https://graphviz.org`
- **Memcached**: An open source caching service available at `https://memcached.org`

The code files for this chapter can be found at `https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition/tree/main/Chapter%2013`.

Common culprits for bad performance

Before we go into the details of profiling, let's first discuss the typical reasons for bad performance in applications.

In practice, many performance problems are pretty common and reoccurring. Knowledge of potential culprits is essential in choosing the right profiling strategy. Also, some specific culprits may manifest in recognizable patterns. If you know what to look for, you will be able to fix obvious performance issues without a detailed analysis and save a lot of time on further optimization attempts.

The most common reasons for experiencing bad application performance are the following:

- Excessive complexity
- Excessive resource allocation and resource leaks
- Excessive I/O and blocking operations

The most dreaded culprit of application inefficiency is excessive complexity, so we will discuss this first.

Code complexity

The first and most obvious thing to look for when trying to improve application performance is complexity. There are many definitions of what makes a program complex, and there are many ways to express it. Some measurable complexity metrics can provide objective information about how code behaves, and such information can often be extrapolated into performance expectations. An experienced programmer can even reliably guess how two different implementations will perform in practice, as long as they're aware of their complexities and the execution context.

The two most popular ways to define application complexity are as follows:

- **Cyclomatic complexity**, which is very often correlated with application performance.
- The **Landau notation**, also known as **big O notation**, is an algorithm classification method that is useful in objectively judging code performance.

The optimization process may therefore be sometimes understood as a process of reducing complexity. In the following sections, we will take a closer look at the definitions of these two types of code complexity.

Cyclomatic complexity

Cyclomatic complexity is a metric that was developed by Thomas J. McCabe in 1976; because of its author, it's also known as **McCabe's complexity**. Cyclomatic complexity measures the number of linear paths through a piece of code. In short, all branching points (`if` statements) and loops (`for` and `while` statements) increase code complexity.

Depending on the value of measured cyclomatic complexity, code can be classified into various complexity classes. The following is a table of commonly used McCabe complexity classes:

Cyclomatic complexity value	Complexity class
1 to 10	Not complex
11 to 20	Moderately complex
21 to 50	Really complex
Above 50	Too complex

Cyclomatic complexity usually correlates inversely with application performance (higher complexity = lower performance). Still, it is more of a code quality score than a performance metric. It does not replace the need for code profiling when looking at performance bottlenecks. Code that has high cyclomatic complexity often tends to utilize rather complex algorithms that may not perform well with larger inputs.

Although cyclomatic complexity is not a reliable way to judge application performance, it has an important advantage: it is a source code metric that can be measured with proper tools. This cannot be said of other canonical ways of expressing complexity, including big O notation. Thanks to its measurability, cyclomatic complexity may be a useful addition to profiling, as it gives you more information about problematic parts of your software. Complex parts of code are the first things you should review when considering radical code architecture redesigns.

Measuring McCabe's complexity is relatively simple in Python because it can be deduced from its abstract syntax tree. Of course, you don't need to do that by yourself. `mccabe` is a popular Python package from PyPI that can perform automated complexity analysis over Python sources. It is also available as a Pytest plugin named `pytest-mccabe`, so it is easy to include in your automated quality and testing processes (see *Chapter 10, Testing and Quality Automation*).

> You can learn more about the `mccabe` package at `https://pypi.org/project/mccabe/`.
>
> The `pytest-mccabe` package is available at `https://pypi.org/project/pytest-mccabe/`.

The big O notation

The canonical method of defining function complexity is the **big O notation**. This metric defines how an algorithm is affected by the size of the input. For instance, does an algorithm's complexity scale linearly with the size of the input, or quadratically?

Manually assessing the big O notation for an algorithm is the best approach when trying to achieve an overview of how its performance is related to the size of the input. Knowing the complexity of your application's components gives you the ability to detect and focus on aspects that will significantly slow down the code.

To measure the big O notation, all constants and low-order terms are removed in order to focus on the portion that really matters when the size of the input data grows very large. The idea is to categorize the algorithm in one of the known complexity categories (even if it is an approximation). The following are the most common complexity classes, where *n* is equal to the number of input elements of a problem:

Notation	Type
O(1)	Constant; does not depend on the input size
O(n)	Linear; will grow at a constant rate as n grows
O(n log n)	Quasi-Linear
O(n^2)	Quadratic
O(n^3)	Cubic
O(n!)	Factorial

We can easily understand complexity classes by comparing basic lookup operations in some Python standard data types:

- **Python list lookup by index has a complexity of O(1)**: Python lists are pre-allocated resizable arrays akin to a `Vector` container from the C++ standard library. The in-memory position of the `list` item of a given index is known in advance, so accessing it is a constant-time operation.

- **Finding the element index in a list by value has a complexity of O(n)**: If you use the `list.index(value)` method, Python will have to iterate over items until it finds the element that matches the `value` expression. In the worst case, it will have to iterate over all the elements of the `list`, so it will perform n operations, where n is the `list` length. On average, however, finding random elements will take $n/2$ operations, so the complexity is O(n).

- **Key lookup in a dictionary has a complexity of O(1)**: In Python, you can use any immutable value as keys in a dictionary. Key values do not translate directly into exact positions in memory, but Python uses an advanced hashing technique that ensures that the average complexity of the key lookup operation is constant.

To understand how function complexity can be determined, let's take a look at the following example:

```python
def function(n):
    for i in range(n):
        print(i)
```

In the preceding function, the `print()` function will be executed n times. Loop length depends linearly on *n*, so the complexity of the whole function is O(*n*).

If the function has conditions, the correct notation is that of the worst-case scenario. Consider the following example:

```python
def function(n, print_count=False):
    if print_count:
        print(f'count: {n}')
    else:
        for i in range(n):
            print(i)
```

In this example, the function could be O(1) or O(*n*), depending on the value of the `print_count` argument. The worst case is O(*n*), so the whole function complexity is *O(n)*.

We don't always have to consider the worst-case scenario when determining complexity. Many algorithms change runtime performance, depending on the statistical characteristic of input data, or amortize the cost of worst-case operations by performing clever tricks. This is why, in many cases, it may be better to review your implementation in terms of average complexity or amortized complexity.

For example, take a look at the operation of appending a single element to Python's `list` type instance. We know that a `list` in CPython uses an array with overallocation for the internal storage instead of linked lists. If an array is already full, appending a new element requires the allocation of a new array and copying all existing elements (references) to a new area in the memory. If we look at this from the point of view of worst-case complexity, it is clear that the `list.append()` method has O(n) complexity, which is a bit expensive compared to a typical implementation of the linked list structure. We also know, however, that the CPython `list` type implementation uses the mechanism of overallocation (it allocates more space than is required at a given time) to mitigate the complexity of occasional reallocation. If we evaluate the complexity over a sequence of operations, we will notice that the average complexity of `list.append()` is O(1), and this is actually a great result.

Always be mindful of big O notation, but don't be too dogmatic about it. Big O notation is asymptotic notation, meaning it is intended to analyze the limiting behavior of a function when the size of the input trends toward infinity. It therefore may not offer a reliable performance approximation for real-life data. Asymptotic notation is a great tool for defining the growth rate of a function, but it won't give a direct answer to the simple question of "Which implementation will take the least time?" Worst-case complexity ignores all the details about timings of individual operations to show you how your program will behave asymptotically. It works for arbitrarily large inputs that you may not always have considered.

For instance, let's assume that you have a problem that requires the processing of n independent elements. Let's say we have *Program A* and *Program B*. You know that *Program A* requires $100n^2$ operations to complete the task, while *Program B* requires $5n^3$ operations to provide a solution. Which one would you choose?

When speaking about very large inputs, *Program A* is the better choice because it behaves better asymptotically. It has O(n^2) complexity compared to *Program B*'s O(n^3) complexity. However, by solving a simple $100n^2 > 5n^3$ inequality, we can find that *Program B* will take fewer operations when n is less than 20. Therefore, if we know a bit more about our input bounds, we can make slightly better decisions. Moreover, we can't assume that individual operations in both programs take the same amount of time. If we want to know which one runs faster for a given input size, we will have to profile both applications.

Excessive resource allocation and leaks

With increased complexity often comes the problem of excessive resource allocation. Code that is complex (either algorithmically or asymptotically) is more likely to include inefficient data structures or allocate too many resources without freeing them afterward.

Although resource usage inefficiency often goes hand in hand with complexity, excessive resource usage needs to be treated as a separate performance issue. And this is for two reasons:

- **Allocating (and freeing) resources takes time**: This includes RAM, which is supposed to be fast memory (when compared to disks). Sometimes, it is better to maintain a pool of resources instead of allocating and freeing them constantly.

- **Your application is unlikely to run alone on the same host**: If it allocates too many resources, it may starve other programs. In extreme cases, excessive usage of resources by a single application may render the whole system unavailable or lead to abrupt program termination.

Whether it is network bandwidth, disk space, memory, or any other resource, if an application appropriates all available resources to itself, this will have a detrimental impact on other programs running in the same environment.

Moreover, many operating systems or environments allow more resources to be requested than are technically available. A common case is memory overcommitting, which allows an operating system to assign processes more memory than hosts physically have available. It works by assigning processes virtual memory. In the event of a physical memory shortage, an operating system will usually temporarily swap unused memory pages to disk to create more free space.

This swapping may go unnoticed at healthy usage levels. However, if virtual memory is overused, normal swapping can transform into 'thrashing,' where the operating system is constantly swapping pages in and out of the disk. This can lead to a dramatic performance collapse.

Excessive resource usage may stem from ineffective data structures, overly large resource pool allocations, or be caused by unintentional **resource leaks**. Resource leaks happen when an application is allocated some specific resources but never frees them, even after it is no longer necessary. Resource leaks are most common for memory allocations, but can happen for other resources (such as file descriptors or open connections) too.

Excessive I/O and blocking operations

When writing applications, we often forget that every operation requires writing to and reading from a disk or via a network connection, which takes time. These Input/Output (I/O) operations always have a noticeable performance impact on applications, and this is what many software architects ignore when designing elaborate networked systems.

With the advent of fast **SSDs** (**Solid State Drives**), disk I/O is faster than ever but is still not as fast as RAM (and RAM is not as fast as a CPU cache). Also, some drives have fast throughput for big files, but perform surprisingly poorly in random access mode.

The same goes for network connections. It cannot go faster than the speed of light. If processes are running on two hosts separated by a great distance, every communication exchange will add a noticeable round-trip overhead. This is a great concern in distributed systems that are often characterized by the fine granulation of many networked services.

Excessive I/O operations will also have an impact on the ability to serve multiple users concurrently. We've already learned about various concurrency models in *Chapter 6*, *Concurrency*. Even with the asynchronous concurrency model, which should theoretically deal best with I/O, it is possible to accidentally include synchronous I/O operations that can potentially ruin the benefits of concurrency. In such a situation, a performance drop may be observed as a drop in application maximum processing throughput.

As you can see, there are many possible reasons for performance problems. In order to fix them, you will need to identify their culprit. We usually do that through a process known as **profiling**.

Code profiling

Knowing what can potentially go wrong allows you to make hypotheses and bets on what are the performance issue culprits and how can you fix them. But profiling is the only way to verify these hypotheses. You should usually avoid optimization attempts without profiling your application first.

Experience helps, so there is of course nothing wrong with doing a small code overview and experiments before profiling. Also, some profiling techniques require the incorporation of additional code instrumentation or the writing of performance tests. It means that often you will have to read it thoroughly anyway. If you perform some small experiments along the way (for instance, in the form of debugging sessions), you may spot something obvious.

Low-hanging fruit happens, but don't rely on it. A good ratio between freeform experiments and classic profiling is around 1:9. My favorite way of organizing the profiling and optimization process is as follows:

1. **Decide how much time you can spend**: Not every optimization is possible, and you will not always be able to fix it. Accept the possibility that you won't succeed on your first attempt. Decide when you will stop in advance. It is better to withdraw and successfully retry later than lose a long-standing battle.

2. **Split that time into sessions**: Optimization is like debugging. It requires focus, organization, and a clear mind. Pick a length of the session that will allow you to run a whole cycle, consisting of putting forward a hypothesis, profiling, and experimentation. This should be no more than a few hours.

3. **Make a schedule**: Implement your sessions when you are least likely to be interrupted and will be most effective. It is often better to plan a few days ahead if the problem is big so you can start every day with fresh ideas.

4. **Don't make other development plans**: If you don't succeed in your first session, you will probably be constantly thinking about your problem. It will be hard to focus on other bigger tasks. If you work in a team, make it transparent because you won't contribute fully until you close the topic. Have a handful of small bite-sized tasks to keep you distracted between sessions.

> Optimization will likely require code changes in your application. To reduce the risk of breaking things, it is a good practice to ensure that code is well covered in tests. We discussed common testing techniques in *Chapter 10, Testing and Quality Automation*.

The key to every profiling session is a hypothesis in terms of what could be wrong. It allows you to decide what profiling technique to use and what tools will work best. A great source of hypotheses are metrics collection and distributed tracing systems, which we discussed in detail in *Chapter 12, Observing Application Behavior and Performance*.

If you have no clue as to what could be wrong, the best way is to start traditional CPU profiling because this has the best chance of leading you to the next viable hypothesis. That's because many common resource and network usage anti-patterns can manifest themselves in CPU profiles as well.

Profiling CPU usage

Typically, performance problems arise in a narrow part of code that has a significant impact on application performance as a whole. We call such places **bottlenecks**. Optimization is a process of identifying and fixing those bottlenecks.

The first source of bottlenecks is your code. The standard library provides all the tools that are required to perform code profiling. They are based on a deterministic approach. A **deterministic profiler** measures the time spent in each function by adding a timer at the lowest level. This introduces a noticeable overhead but provides a good idea of where the time is consumed. A **statistical profiler**, on the other hand, samples instruction pointer usage and does not instrument the code. The latter is less accurate, but allows you to run the target program at full speed.

There are two ways to profile the code:

- **Macro-profiling**: This profiles the whole program while it is being used and generates statistics.
- **Micro-profiling**: This measures a precise part of the program by instrumenting it manually.

Where we don't know the exact fragment of the code that runs slowly, we usually start with macro-profiling of the whole application to get a general performance profile and determine components that act as performance bottlenecks.

Macro-profiling

Macro-profiling is done by running the application in a special mode, where the interpreter is instrumented to collect statistics regarding code usage. The Python standard library provides several tools for this, including the following:

- `profile`: This is a pure Python implementation.
- `cProfile`: This is a C implementation that provides the same interface as that of the profile tool, but has less overhead.

The recommended choice for most Python programmers is `cProfile` due to its reduced overhead. In any case, if you need to extend the profiler in some way, then `profile` will be a better choice because it doesn't use C extensions and so is easier to extend.

Both tools have the same interface and usage, so we will use only one of them here. The following is a `myapp.py` module with a `main` function that we are going to profile with the `cProfile` module:

```python
import time

def medium():
    time.sleep(0.01)

def light():
    time.sleep(0.001)

def heavy():
    for i in range(100):
        light()
        medium()
        medium()
    time.sleep(2)

def main():
    for i in range(2):
        heavy()

if __name__ == '__main__':
    main()
```

This module can be called directly from the prompt, and the results are summarized here:

```
$ python3 -m cProfile myapp.py
```

Example profiling output for our myapp.py script can be as follows:

```
         1208 function calls in 8.243 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        2    0.001    0.000    8.243    4.121 myapp.py:13(heavy)
        1    0.000    0.000    8.243    8.243 myapp.py:2(<module>)
        1    0.000    0.000    8.243    8.243 myapp.py:21(main)
      400    0.001    0.000    4.026    0.010 myapp.py:5(medium)
      200    0.000    0.000    0.212    0.001 myapp.py:9(light)
        1    0.000    0.000    8.243    8.243 {built-in method exec}
      602    8.241    0.014    8.241    0.014 {built-in method sleep}
```

The meaning of each column is as follows:

- ncalls: The total number of calls
- tottime: The total time spent in the function, excluding time spent in the calls of sub-functions
- cumtime: The total time spent in the function, including time spent in the calls of sub-functions

The percall column to the left of tottime equals tottime / ncalls, and the percall column to the left of cumtime equals the cumtime / ncalls.

These statistics are a print view of the statistic object that was created by the profiler. You can also create and review this object within the interactive Python session, as follows:

```
>>> import cProfile
>>> from myapp import main
>>> profiler = cProfile.Profile()
>>> profiler.runcall(main)
>>> profiler.print_stats()
         1206 function calls in 8.243 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall file:lineno(function)
        2    0.001    0.000    8.243    4.121 myapp.py:13(heavy)
        1    0.000    0.000    8.243    8.243 myapp.py:21(main)
      400    0.001    0.000    4.026    0.010 myapp.py:5(medium)
      200    0.000    0.000    0.212    0.001 myapp.py:9(light)
      602    8.241    0.014    8.241    0.014 {built-in method sleep}
```

The statistics can also be saved in a file and then read by the `pstats` module. This module provides a class that knows how to handle profile files, and gives a few helpers to more easily review the profiling results. The following transcript shows how to access the total number of calls and how to display the first three calls, sorted by the `time` metric:

```
>>> import pstats
>>> import cProfile
>>> from myapp import main
>>> cProfile.run('main()', 'myapp.stats')
>>> stats = pstats.Stats('myapp.stats')
>>> stats.total_calls
1208
>>> stats.sort_stats('time').print_stats(3)
Mon Apr  4 21:44:36 2016    myapp.stats

         1208 function calls in 8.243 seconds

   Ordered by: internal time
   List reduced from 8 to 3 due to restriction <3>

   ncalls  tottime  percall  cumtime  percall file:lineno(function)
      602    8.241    0.014    8.241    0.014 {built-in method sleep}
```

```
    400    0.001    0.000    4.025    0.010 myapp.py:5(medium)
      2    0.001    0.000    8.243    4.121 myapp.py:13(heavy)
```

From there, you can browse the code by printing out the callers and callees for each function, as follows:

```
>>> stats.print_callees('medium')
   Ordered by: internal time
   List reduced from 8 to 1 due to restriction <'medium'>

Function            called...
                    ncalls  tottime  cumtime
myapp.py:5(medium) -> 400     4.025    4.025  {built-in method sleep}

>>> stats.print_callees('light')
   Ordered by: internal time
   List reduced from 8 to 1 due to restriction <'light'>

Function            called...
                     ncalls  tottime  cumtime
myapp.py:9(light)  -> 200     0.212    0.212  {built-in method sleep}
```

Being able to sort the output allows you to work on different views to find the bottlenecks. For instance, consider the following scenarios:

- When the number of small calls (a low value of percall for the tottime column) is really high (high value of ncalls) and takes up most of the global time, the function or method is probably running in a very long loop. Often, optimization can be done by moving this call to a different scope to reduce the number of operations.

- When a single function call is taking a very long time, a cache might be a good option, if possible.

Another great way to visualize bottlenecks from profiling data is to transform them into diagrams (see *Figure 13.1*). The gprof2dot.py script can be used to turn profiler data into a dot graph:

```
$ gprof2dot.py -f pstats myapp.stats | dot -Tpng -o output.png
```

> The gprof2dot.py script is part of the gprof2dot package available on PyPI. You can download it using pip. Working with it requires the installation of Graphviz software. You can download this for free from http://www.graphviz.org/.

The following is an example output of the preceding `gprof2dot.py` invocation in a Linux shell that turns the `myapp.stats` profile file into a PNG diagram:
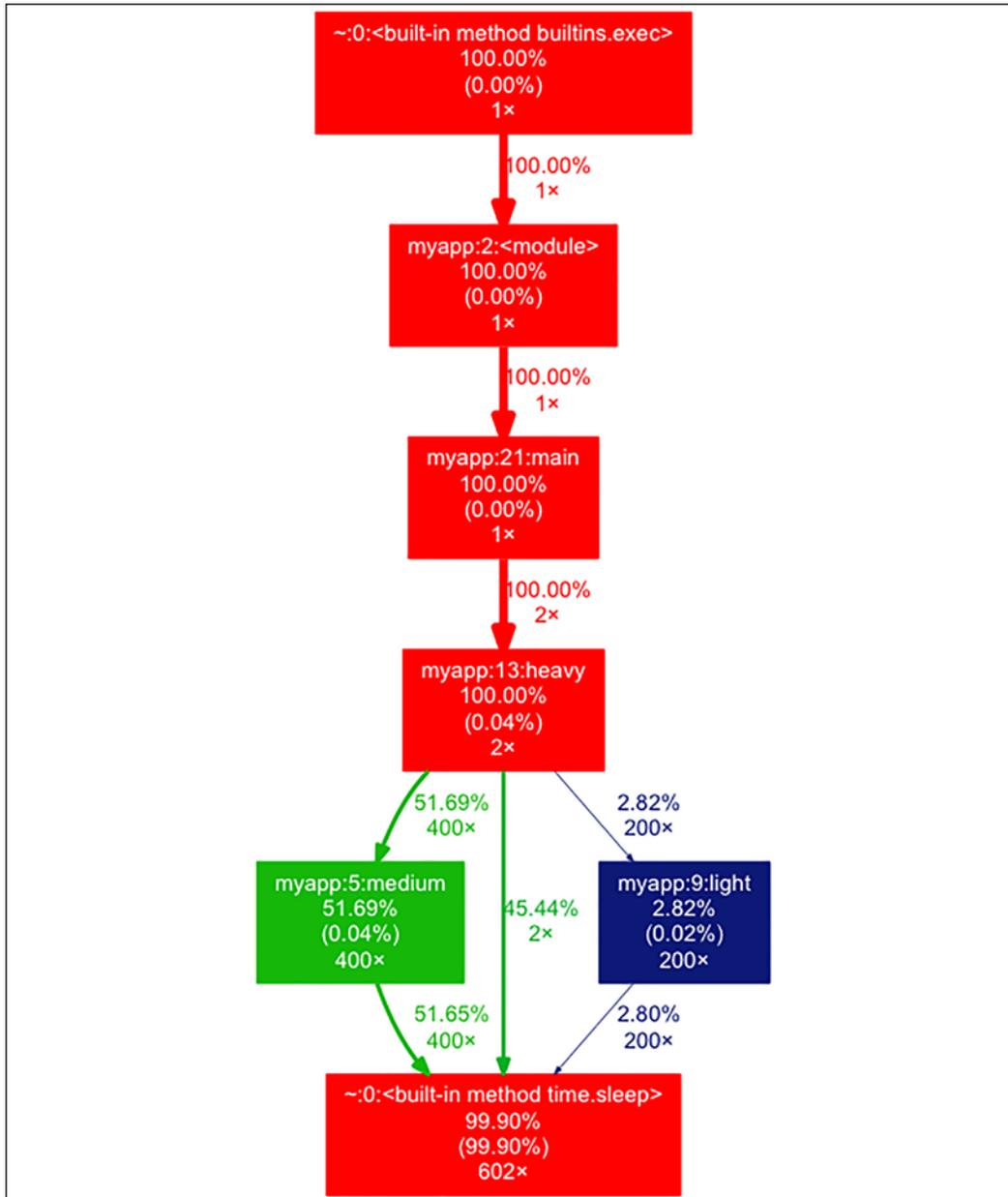
Figure 13.1: An example of a profiling overview diagram that was generated with gprof2dot

The example diagram shows different code paths that were executed by the program and the relative time spent in each path. Each box represents a single function. Linking the functions, you have the number of times a given code path was executed and the percentage of total execution time spent on those paths. Using diagrams is a great way to explore the performance patterns of large applications.

> The advantage of `gprof2dot` is that it tries to be language-agnostic. It is not limited to a Python profile or cProfile output and can read from multiple other profiles, such as Linux `perf`, `xperf`, `gprof`, Java HPROF, and many others.

Macro-profiling is a good way to detect the function that has a problem, or at least its neighborhood. When you have found it, you can proceed to micro-profiling.

Micro-profiling

When the slow function is found, we usually proceed with micro-profiling to generate a profile focused on the smallest possible amount of code. This is done by manually instrumenting a part of the code in a specifically created speed test.

For instance, the `cProfile` module can be used in the form of a decorator, as in the following example:

```python
import time
import tempfile
import cProfile
import pstats


def profile(column='time', list=3):
    def parametrized_decorator(function):
        def decorated(*args, **kw):
            s = tempfile.mktemp()

            profiler = cProfile.Profile()
            profiler.runcall(function, *args, **kw)
            profiler.dump_stats(s)

            p = pstats.Stats(s)
            print("=" * 5, f"{function.__name__}() profile", "=" * 5)
            p.sort_stats(column).print_stats(list)
        return decorated
```

```
        return parametrized_decorator

def medium():
    time.sleep(0.01)

@profile('time')
def heavy():
    for i in range(100):
        medium()
        medium()
    time.sleep(2)

def main():
    for i in range(2):
        heavy()

if __name__ == '__main__':
    main()
```

This approach allows for testing only selected parts of the application (here, the heavy() function) and sharpens the statistics' output. This way, you can collect many isolated and precisely targeted profiles on a single application run, as follows. The following is example output of running the preceding code in a Python interpreter:

```
===== heavy() profile =====
Wed Apr 10 03:11:53 2019 /var/folders/jy/wy13kx0s7sb1dx2rfsqdvzdw0000gq
/T/tmpyi2wejm5

        403 function calls in 4.330 seconds

 Ordered by: internal time
 List reduced from 4 to 3 due to restriction <3>

 ncalls tottime percall cumtime percall filename:lineno(function)
    201 4.327   0.022   4.327   0.022   {built-in method time.sleep}
    200 0.002   0.000   2.326   0.012   cprofile_decorator.
py:24(medium)
      1 0.001   0.001   4.330   4.330   cprofile_decorator.py:28(heavy)

===== heavy() profile =====
Wed Apr 10 03:11:57 2019 /var/folders/jy/wy13kx0s7sb1dx2rfsqdvzdw0000gq
/T/tmp8mubgwjw
```

```
        403 function calls in 4.328 seconds

 Ordered by: internal time
 List reduced from 4 to 3 due to restriction <3>

 ncalls tottime percall cumtime percall filename:lineno(function)
    201 4.324   0.022   4.324   0.022   {built-in method time.sleep}
    200 0.002   0.000   2.325   0.012   cprofile_decorator.
py:24(medium)
      1 0.001   0.001   4.328   4.328.  cprofile_decorator.py:28(heavy)
```

In the preceding output, we see that the heavy() function was called exactly twice and both times the profile was very similar. In the list of calls, we made 201 calls to the time.sleep() function, which cumulatively takes around 4.3 seconds to execute.

Sometimes at this stage, having just a profile with a list of callees is not enough to understand the problem. A common practice is to create alternative implementations of specific parts of the code and measure how long it takes to execute them. If the heavy() function was to do anything useful, we could try, for instance, to solve the same problem with code of lower complexity.

timeit is a useful module that provides a simple way to measure the execution time of a small code snippet, with the best underlying timer the host system provides (the time.perf_counter() object), as shown in the following example:

```
>>> from myapp import medium
>>> import timeit
>>> timeit.timeit(light, number=1000)
1.2880568829999675
```

The timeit.timeit() function will run the specified function 1,000 times (specified by the number parameter) and return the total amount of time spent during all executions. If you expect a larger variance in terms of results, you can use timeit. repeat() to repeat the whole test a specified number of times:

```
>>> timeit.repeat(light, repeat=5, number=1000)
[1.283251813999982, 1.276404459999913, 1.2787090969999326,
1.279601415000002, 1.2796783549999873]
```

The timeit module allows you to repeat the call multiple times, and can be easily used to try out isolated code snippets. This is very useful outside the application context, in a prompt, for instance, but is not really ideal for use within an existing application.

The `timeit` module can be used to create speed tests in your automated testing framework, but that approach should be used with caution. Timing results may vary each time. Repeating the same test multiple times and taking the average provides more accurate results. Furthermore, some computers have special CPU features, which might change the processor clock time depending on the load or temperature. So you can see different time results when the computer is idling and when it is really busy. Other programs running concurrently will also very likely affect overall timings. So, continually repeating the test is good practice for small code snippets. That's why, when measuring time in automated testing flows, we usually try to observe patterns rather than put concrete timing thresholds as assertions.

When doing CPU profiling, we often spot various patterns related to acquiring and releasing resources as these often happen through function calls. Traditional Python profiling with `profile` and `cProfile` modules can provide a general overview of resource usage patterns, but when it comes to memory usage, we prefer dedicated memory profiling solutions.

Profiling memory usage

Before we start to hunt down memory issues in Python, you should know that the nature of memory leaks in Python is quite special. In some compiled languages such as C and C++, the memory leaks are almost exclusively caused by allocated memory blocks that are no longer referenced by any pointer. If you don't have a reference to memory, you cannot release it. This very situation is called a **memory leak**.

In Python, there is no low-level memory management available for the user, so we instead deal with leaking references — references to objects that are no longer needed but were not removed. This stops the interpreter from releasing resources, but is not the same situation as a classic memory leak in C.

> There is always the exceptional case of memory leaking through pointers in Python C extensions, but they are a different kind of beast that needs completely different tools to diagnose. These cannot be easily inspected from Python code. A common tool for inspecting memory leaks in C is Valgrind. You can learn more about Valgrind at `https://valgrind.org/`.

So, memory issues in Python are mostly caused by unexpected or unplanned resource acquisition patterns. It happens very rarely that this is the effect of real bugs caused by the mishandling of memory allocation and deallocation routines. Such routines are available to the developer only in CPython when writing C extensions with Python/C APIs. You will deal with these very rarely, if ever. Thus, most memory leaks in Python are mainly caused by the overblown complexity of the software and subtle interactions between its components that are really hard to track. To spot and locate such deficiencies in your software, you need to know how actual memory usage looks in the program.

Getting information about how many objects are controlled by the Python interpreter and inspecting their real size is a bit tricky. For instance, knowing how much memory a given object takes in bytes would involve crawling down all its attributes, dealing with cross-references, and then summing up everything. This is a pretty complex problem if you consider the way objects tend to refer to each other. The built-in `gc` module, which is the interface of Python's garbage collector, does not provide high-level functions for this, and it would require Python to be compiled in debug mode to have a full set of information.

Often, programmers just ask the system about the memory usage of their application before and after a given operation has been performed. But this measure is an approximation and depends a lot on how the memory is managed at the system level. Using the `top` command under Linux, or the task manager under Windows, for instance, makes it possible to detect memory problems when they are obvious. However, this approach is laborious and makes it really hard to track down the faulty code block.

Fortunately, there are a few tools available to make memory snapshots and calculate the number and size of loaded objects. But let's keep in mind that Python does not release memory easily and prefers to hold on to it in case it is needed again.

For some time, one of the most popular tools to use when debugging memory issues and usage in Python was Guppy-PE and its Heapy component. Unfortunately, it no longer appears to be maintained and it lacks Python 3 support. Luckily, the following are some of the other alternatives that are Python 3-compatible to some extent:

- **Memprof** (`http://jmdana.github.io/memprof/`): This is declared to work on some POSIX-compliant systems (macOS and Linux). Last updated in December 2019.

- **memory_profiler** (`https://pypi.org/project/memory-profiler`): Declared to be OS-independent. Actively maintained.

- **Pympler** (`https://pypi.org/project/Pympler/`): Declared to be OS-independent. Actively maintained.

- **objgraph** (`https://mg.pov.lt/objgraph/`): Declared to be OS-independent. Actively maintained.

> Note that the preceding information regarding compatibility is based purely on trove classifiers that are used by the latest distributions of featured packages, declarations from the documentation, and the inspection of projects' build pipeline definitions at the time of writing the book. These values may be different at the time you read this.

As you can see, there are a lot of memory profiling tools available to Python developers. Each one has some constraints and limitations. In this chapter, we will focus on one profiler that is known to work well with the latest release of Python (that is, Python 3.9) on different operating systems. This tool is `objgraph`.

The API of `objgraph` may seem a bit clumsy, and it has a very limited set of functionalities. But it works, does what it needs to do very well, and is really simple to use. Memory instrumentation is not a thing that is added to the production code permanently, so this tool does not need to be pretty.

Using the objgraph module

`objgraph` is a simple module for creating diagrams of object references within applications. These are extremely useful when hunting memory leaks in Python.

> `objgraph` is available on PyPI, but it is not a completely standalone tool and requires Graphviz, which was installed in the *Macro-profiling* section, in order to create memory usage diagrams.

`objgraph` provides multiple utilities that allow you to list and print various statistics regarding memory usage and object counts. An example of such utilities in use is shown in the following transcript of interpreter sessions:

```
>>> import objgraph
>>> objgraph.show_most_common_types()
function                    1910
dict                        1003
wrapper_descriptor          989
tuple                       837
weakref                     742
method_descriptor           683
builtin_function_or_method 666
getset_descriptor           338
set                         323
member_descriptor           305
>>> objgraph.count('list')
266
>>> objgraph.typestats(objgraph.get_leaking_objects())
{'Gt': 1, 'AugLoad': 1, 'GtE': 1, 'Pow': 1, 'tuple': 2, 'AugStore':
1, 'Store': 1, 'Or': 1, 'IsNot': 1, 'RecursionError': 1, 'Div': 1,
'LShift': 1, 'Mod': 1, 'Add': 1, 'Invert': 1, 'weakref': 1, 'Not': 1,
'Sub': 1, 'In': 1, 'NotIn': 1, 'Load': 1, 'NotEq': 1, 'BitAnd': 1,
'FloorDiv': 1, 'Is': 1, 'RShift': 1, 'MatMult': 1, 'Eq': 1, 'Lt': 1,
'dict': 341, 'list': 7, 'Param': 1, 'USub': 1, 'BitOr': 1, 'BitXor': 1,
'And': 1, 'Del': 1, 'UAdd': 1, 'Mult': 1, 'LtE': 1}
```

Note that the preceding numbers of allocated objects displayed by `objgraph` are already high due to the fact that a lot of Python built-in functions and types are ordinary Python objects that live in the same process memory. Also, `objgraph` itself creates some objects that are included in this summary.

As mentioned earlier, `objgraph` allows you to create diagrams of memory usage patterns and cross-references that link all the objects in the given namespace. The most useful functions of the `objgraph` module are `objgraph.show_refs()` and `objgraph.show_backrefs()`. They both accept a reference to the object being inspected and save a diagram image to file using the Graphviz package. Examples of such graphs are presented in *Figure 13.2* and *Figure 13.3*. Here is the code that was used to create these diagrams:

```python
from collections import Counter
import objgraph


def graph_references(*objects):
    objgraph.show_refs(
        objects,
        filename='show_refs.png',
        refcounts=True,
        # additional filtering for the sake of brevity
        too_many=5,
        filter=lambda x: not isinstance(x, dict),
    )
    objgraph.show_backrefs(
        objects,
        filename='show_backrefs.png',
        refcounts=True
    )


if __name__ == "__main__":
    quote = """
    People who think they know everything are a
    great annoyance to those of us who do.
    """
    words = quote.lower().strip().split()
    counts = Counter(words)
    graph_references(words, quote, counts)
```

> Without Graphviz installed, `objgraph` will output diagrams in DOT format, which is a special graph description language.

The following diagram shows the diagram of all references held by `words`, `quote`, and `counts` objects:
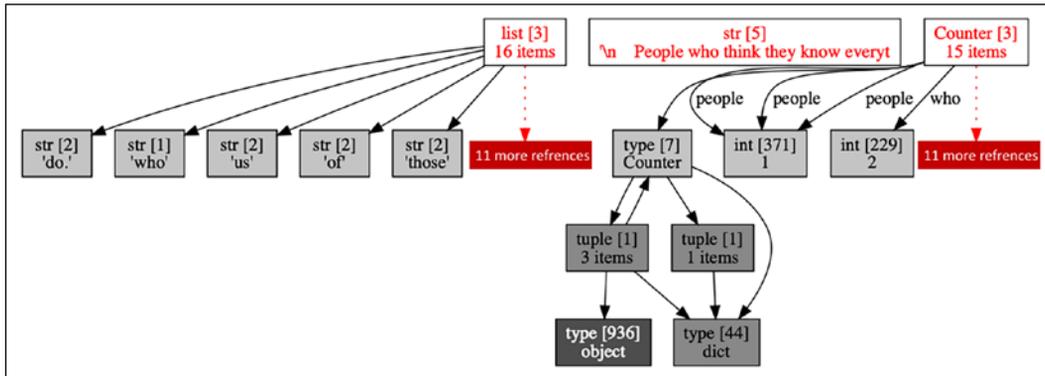
Figure 13.2: An example result of the show_refs() diagram from the graph_references() function

As you can see, the `words` object (denoted as `list [3]`) holds references to 16 objects. The `counts` object (denoted as `Counter [3]`) holds references to 15 objects. It is one object less than the `words` object because the word "who" appears twice. The `quote` object (denoted as `str [5]`) is a plain string, so it doesn't hold any extra references.

The following diagram shows the back references leading to `words`, `quote`, and `counts` objects:
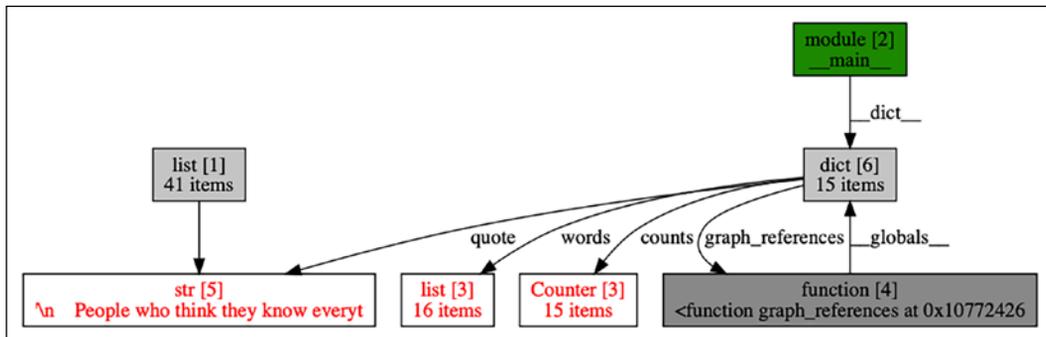
Figure 13.3: An example result of the show_backrefs() diagram from the graph_references() function

In the preceding diagram, we see that the `quote`, `words`, and `counts` objects are references in a dictionary (denoted as `dict[6]`) of global variables of the `__main__` module named `__globals__`. Moreover, the `quote` object is referenced in a special list object (denoted as `list [1]`) due to CPython's string interning mechanism.

> String interning is the memory optimization mechanism of CPython. Most of the string literals are pre-allocated by CPython when a module is loaded. Strings in Python are immutable, so duplicate occurrences of the same string literal will refer to the same address in memory.

To show how `objgraph` may be used in practice, let's review a code example that may create memory issues under certain versions of Python. As we already noted in *Chapter 9*, *Bridging Python with C and C++*, CPython has its own garbage collector that exists independently from its reference counting mechanism. It is not used for general-purpose memory management, and its sole purpose is to solve the problem of cyclic references. In many situations, objects may reference each other in a way that would make it impossible to remove them using simple techniques based on reference counting. Here is the simplest example:

```
x = []
y = [x]
x.append(y)
```

Such a situation is presented visually in *Figure 13.4*:
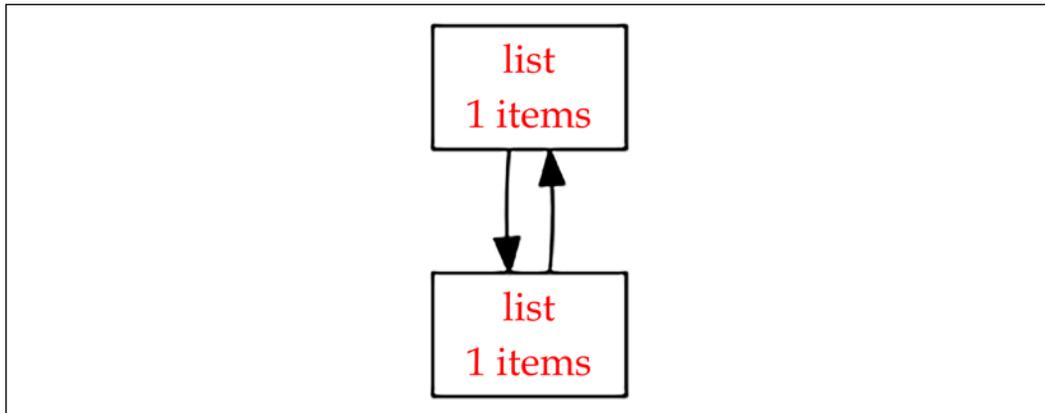
Figure 13.4: An example diagram generated by objgraph of cyclic references between two objects

In the preceding case, even if all external references to *x* and *y* objects are removed (for instance, by returning from the local scope of a function), these two objects cannot be removed through reference counting because there will always be two cross-references owned by these two objects. This is a situation where the Python garbage collector steps in. It can detect cyclic references to objects and trigger their deallocation if there are no other valid references to these objects outside of the cycle.

The real problem starts when at least one of the objects in such a cycle has the custom __del__() method defined. This is a custom deallocation handler that will be called when the object's reference count finally goes to zero. It can execute any arbitrary Python code and thus can also create new references to featured objects. This is the reason why the garbage collector prior to Python 3.4 could not break reference cycles if at least one of the objects provided the custom __del__() method implementation. PEP 442 introduced safe object finalization to Python and became part of the language standard, starting from Python 3.4. Anyway, this may still be a problem for packages that worry about backward compatibility and target a wide spectrum of Python interpreter versions. The following snippet of code allows you to show the difference in behavior between the cyclic garbage collector in different Python versions:

```python
import gc
import platform
import objgraph


class WithDel(list):
    """ list subclass with custom __del__ implementation """
    def __del__(self):
        pass


def main():
    x = WithDel()
    y = []
    z = []

    x.append(y)
    y.append(z)
    z.append(x)

    del x, y, z

    print("unreachable prior collection: %s" % gc.collect())
    print("unreachable after collection: %s" % len(gc.garbage))
    print("WithDel objects count:        %s" %
          objgraph.count('WithDel'))
```

```python
if __name__ == "__main__":
    print("Python version: %s" % platform.python_version())
    print()
    main()
```

The following output of the preceding code, when executed under Python 3.3, shows that the cyclic garbage collector in the older versions of Python cannot collect objects that have the __del__() method defined:

```
$ python3.3 with_del.py
Python version: 3.3.5

unreachable prior collection: 3
unreachable after collection: 1
WithDel objects count:       1
```

With a newer version of Python, the garbage collector can safely deal with the finalization of objects, even if they have the __del__() method defined, as follows:

```
$ python3.5 with_del.py
Python version: 3.5.1

unreachable prior collection: 3
unreachable after collection: 0
WithDel objects count:       0
```

Although custom finalization is no longer a memory threat in the latest Python releases, it still poses a problem for applications that need to work in different environments. As we mentioned earlier, the objgraph.show_refs() and objgraph. show_backrefs() functions allow you to easily spot problematic objects that take part in unbreakable reference cycles. For instance, we can easily modify the main() function to show all back references to the WithDel instances to see whether we have leaking resources, as follows:

```python
def main():
    x = WithDel()
    y = []
    z = []

    x.append(y)
    y.append(z)
    z.append(x)
```

```
del x, y, z

print("unreachable prior collection: %s" % gc.collect())
print("unreachable after collection: %s" % len(gc.garbage))
print("WithDel objects count:        %s" %
        objgraph.count('WithDel'))

objgraph.show_backrefs(
    objgraph.by_type('WithDel'),
    filename='after-gc.png'
)
```

Running the preceding example under Python 3.3 will result in the diagram presented in *Figure 13.5*. It shows that `gc.collect()` could not succeed in removing x, y, and z object instances. Additionally, `objgraph` highlights all the objects that have the custom `__del__()` method in red to make spotting such issues easier:
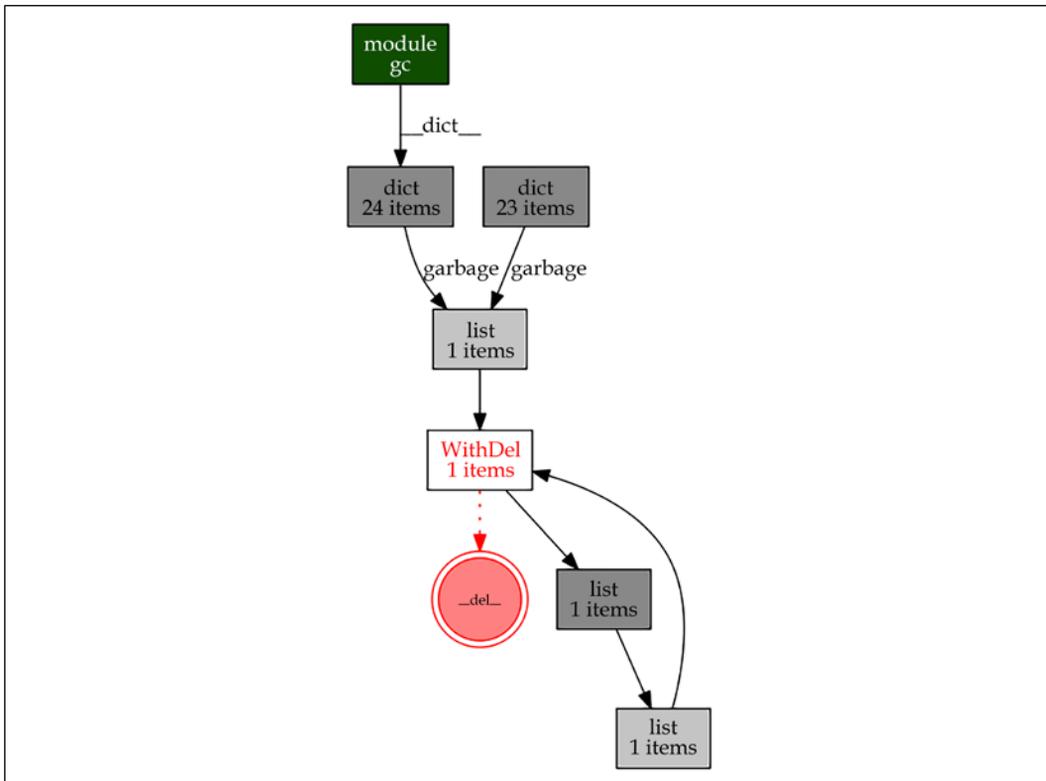
Figure 13.5: A diagram showing an example of cyclic references that can't be picked
by the Python garbage collector prior to Python 3.4

When memory leaks happen in the C extensions (for instance, in Python/C extensions), it is usually much harder to diagnose and profile them. However, harder does not mean impossible, as we will discover in the next section.

C code memory leaks

If the Python code seems perfectly fine and memory still increases when you loop through the isolated function, the leak might be located on the C side. This happens, for instance, when a `Py_DECREF` macro is missing in the critical part of an imported C extension.

The C code of CPython interpreter is pretty robust and tested for the existence of memory leaks, so it is the last place to look for memory problems. But if you use packages that have custom C extensions, they might be a good place to look first. Because you will be dealing with code operating on a much lower level of abstraction than Python, you need to use completely different tools to resolve such memory issues.

Memory debugging is not easy in C, so before diving into extension internals, make sure that you properly diagnose the source of your problem. It is a very popular approach to isolate a suspicious package with code similar in nature to unit tests. To diagnose the source of your problem, you should consider the following actions:

1. Write a separate test for each API unit or functionality of an extension you suspect of leaking memory.

2. Perform the test in a loop for an arbitrarily long time in isolation (one test function per run).

3. Observe from outside which of the tested functionalities increases memory usage over time.

By using such an approach, you will eventually isolate the faulty part of the extension and this will reduce the time required later to inspect and fix its code. This process may seem burdensome because it requires a lot of additional time and coding, but it does pay off in the long run. You can always make your work easier by reusing some of the testing tools that were introduced in *Chapter 10*, *Testing and Quality Automation*. Utilities such as Pytest were perhaps not designed specifically for this case, but can at least reduce the time required to run multiple tests in isolated environments.

If you have successfully isolated the part of the extension that is leaking the memory, you can finally start actual debugging. If you're lucky, a simple manual inspection of the isolated source code section may provide the desired results. In many cases, the problem is as simple as adding the missing `Py_DECREF` call. Nevertheless, in most cases, your work won't be that simple. In such situations, you need to bring out some bigger guns. One of the notable generic tools for combating memory leaks in compiled code that should be in every programmer's toolbelt is Valgrind. This is a whole instrumentation framework for building dynamic analysis tools. Because of this, it may not be easy to learn and master, but you should definitely acquaint yourself with the basics of its usage.

> You can learn more about Valgrind at `https://valgrind.org`.

After profiling, when you know what's wrong with your code performance, it's time to apply actual code optimizations. The most common reason for bad performance is code complexity. Very often, code complexity can be reduced just by applying appropriate data structures. Let's now look at some examples of optimizations with built-in Python data types.

Reducing complexity by choosing appropriate data structures

To reduce code complexity, it's important to consider how data is stored. You should pick your data structure carefully. The following section will provide you with a few examples of how the performance of simple code snippets can be improved by the correct data types.

Searching in a list

Due to the implementation details of the `list` type in Python, searching for a specific value in a `list` isn't a cheap operation. The complexity of the `list.index()` method is $O(n)$, where n is the number of list elements. Such linear complexity won't be an issue if you only need to perform a few element index lookups, but it can have a negative performance impact in some critical code sections, especially if done over very large lists.

If you need to search through a `list` quickly and often, you can try the `bisect` module from Python's standard library. The functions in this module are mainly designed for inserting or finding insertion indexes for given values in a way that will preserve the order of the already sorted sequence. This module is used to efficiently find an element index using a bisection algorithm. The following recipe, from the official documentation of the function, finds an element index using a binary search:

```python
def index(a, x):
    'Locate the leftmost value exactly equal to x'
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    raise ValueError
```

Note that every function from the `bisect` module requires a sorted sequence in order to work. If your `list` is not in the correct order, then sorting it is a task with a worst-case complexity of O(*n log n*). This is a worse class than O(*n*), so sorting the whole `list` to then perform only a single search will not pay off. However, if you need to perform a number of index searches across a large `list` that rarely changes, using a single `sort()` operation for `bisect` may prove to be the best trade-off.

If you already have a sorted `list`, you can also insert new items into that `list` using `bisect` without needing to re-sort it. The `bisect_left()` and `bisect_right()` functions already return insertion points in the left-to-right or right-to-left sorted lists accordingly. The following is an example of inserting a new value in a left-to-right sorted `list` using the `bisect_left()` function:

```python
>>> from bisect import bisect_left
>>> items = [1, 5, 6, 19, 20]
>>> items.insert(bisect_left(items, 15), 15)
>>> items
[1, 5, 6, 15, 19, 20]
```

There are also `insort_left()` and `insort_right()` functions that are shorthand for inserting elements in sorted lists:

```python
>>> from bisect import insort_left
>>> items = [1, 5, 6, 19, 20]
>>> insort_left(items, 15)
>>> items
[1, 5, 6, 15, 19, 20]
```

In the next section, we will see how to use a set instead of a `list` when element uniqueness is required.

Using sets

When you need to build a sequence of distinct values from a given sequence, the first algorithm that might come to mind is as follows:

```
>>> sequence = ['a', 'a', 'b', 'c', 'c', 'd']
>>> result = []
>>> for element in sequence:
...     if element not in result:
...         result.append(element)
...
>>> result
['a', 'b', 'c', 'd']
```

In the preceding example, the complexity is introduced by the lookup in the `result` list. The `in` operator has a time complexity of $O(n)$. It is then used in the loop, which costs $O(n)$. So, the overall complexity is quadratic, that is, $O(n^2)$.

Using a set type for the same work will be faster because the stored values are looked up using hashes (the same as in the `dict` type). The `set` type also ensures the uniqueness of elements, so we don't need to do anything more than create a new set from the sequence object. In other words, for each value in sequence, the time taken to see whether it is already in the set will be constant, as follows:

```
>>> sequence = ['a', 'a', 'b', 'c', 'c', 'd']
>>> unique = set(sequence)
>>> unique
set(['a', 'c', 'b', 'd'])
```

This lowers the complexity to $O(n)$, which is the complexity of the set object creation. The additional advantage of using the `set` type for element uniqueness is shorter and more explicit code.

Sometimes the built-in data types are not enough to handle your data structures efficiently. Python comes with a great set of additional performant data types in the `collections` module.

Using the collections module

The `collections` module provides specialized alternatives to general-purpose, built-in container types. The main types from this module that we will focus on in this chapter are as follows:

- `deque`: A `list`-like type with extra features
- `defaultdict`: A `dict`-like type with a built-in default factory feature
- `namedtuple`: A `tuple`-like type that assigns keys for members

> We've already discussed other types from the `collections` module in other chapters: `ChainMap` in *Chapter 3*, *New Things in Python*; `UserList` and `UserDict` in *Chapter 4*, *Python in Comparison with Other Languages*; and `Counter` in *Chapter 5*, *Interfaces, Patterns, and Modularity*.

We'll discuss these types in the following sections.

deque

A `deque` is an alternative implementation for lists. While the built-in `list` type is based on ordinary arrays, a `deque` is based on a **doubly-linked list**. Hence, a `deque` is much faster when you need to insert something into its tail or head, but much slower when you need to access an arbitrary index.

Of course, thanks to the overallocation of an internal array in the Python `list` type, not every `list.append()` call requires memory reallocation, and the average complexity of this method is O(1). The situation changes dramatically when the element needs to be added at the first index of the `list`. Because all elements to the right of the new one need to be shifted in an array, the complexity of `list.insert()` is O(n). If you need to perform a lot of pops, appends, and inserts, a `deque` in place of a `list` may provide a substantial performance improvement.

> Remember to always profile your code before switching from a `list` to a `deque` because a few things that are fast in arrays (such as accessing an arbitrary index) are extremely inefficient in linked lists.

For example, if we measure the time it takes to append one element and remove it from the sequence with `timeit`, the difference between a `list` and a `deque` may not even be noticeable.

The following is a sample `timeit` run for the `list` type:

```
$ python3 -m timeit \
  -s 'sequence=list(range(10))' \
  'sequence.append(0); sequence.pop();'
1000000 loops, best of 3: 0.168 usec per loop
```

And the following is a sample `timeit` run for the `deque` type:

```
$ python3 -m timeit \
  -s 'from collections import deque; sequence=deque(range(10))' \
  'sequence.append(0); sequence.pop();'
1000000 loops, best of 3: 0.168 usec per loop
```

However, if we perform a similar comparison for situations where we want to add and remove the first element of the sequence, the performance difference is impressive.

The following is a sample `timeit` run for the `list` type:

```
$ python3 -m timeit \
  -s 'sequence=list(range(10))' \
  'sequence.insert(0, 0); sequence.pop(0)'
1000000 loops, best of 3: 0.392 usec per loop
```

And the following is a similar `timeit` run for the `deque` type:

```
$ python3 -m timeit \
  -s 'from collections import deque; sequence=deque(range(10))' \
  'sequence.appendleft(0); sequence.popleft()'
10000000 loops, best of 3: 0.172 usec per loop
```

As you may expect, the difference gets bigger as the size of the sequence grows. The following is an example of running the same test with `timeit` for a `list` that contains 10,000 elements:

```
$ python3 -m timeit \
  -s 'sequence=list(range(10000))' \
  'sequence.insert(0, 0); sequence.pop(0)'
100000 loops, best of 3: 14 usec per loop
```

If we do the same for `deque`, we will see that the timing of the operation does not change:

```
$ python3 -m timeit \
  -s 'from collections import deque; sequence=deque(range(10000))' \
  'sequence.appendleft(0); sequence.popleft()'
10000000 loops, best of 3: 0.168 usec per loop
```

Thanks to the efficient `append()` and `pop()` methods, which work at the same speed from both ends of the sequence, `deque` makes a perfect example of implementing queues. For example, a **First-In First-Out** (**FIFO**) queue will be much more efficient if implemented with `deque` instead of `list`.

> `deque` works well when implementing queues, but there is also a separate `queue` module in Python's standard library that provides a basic implementation for FIFO, **Last-In First-Out** (**LIFO**), and priority queues. If you want to utilize queues as a mechanism of inter-thread communication, you should use classes from the `queue` module instead of `collections.deque`. This is because these classes provide all the necessary locking semantics. If you don't use threading and choose not to utilize queues as a communication mechanism, `deque` should be enough to provide queue implementation basics.

defaultdict

The `defaultdict` type is similar to the `dict` type, except it adds a default factory for new keys. This avoids the need to write an extra test to initialize the mapping entry and is also a bit more efficient than the `dict.setdefault` method.

`defaultdict` may seem like simple syntactic sugar over `dict` that allows us to write shorter code. However, the fallback to a pre-defined value on a failed key lookup is slightly faster than the `dict.setdefault()` method.

The following is a timeit run for the `dict.setdefault()` method:

```
$ python3 -m timeit \
  -s 'd = {}' \
  'd.setdefault("x", None)'
10000000 loops, best of 3: 0.153 usec per loop
```

And the following is a timeit run for the equivalent `defaultdict` usage:

```
$ python3 -m timeit \
-s 'from collections import defaultdict; d = defaultdict(lambda: None)' \
'd["x"]'
10000000 loops, best of 3: 0.0447 usec per loop
```

> The difference in the preceding example may seem more substantial, but the computational complexity hasn't changed. The `dict.setdefault()` method consists of two steps (key lookup and key set), both of which have a complexity of O(1). There is no way to have a complexity class lower than O(1), but sometimes it is worth looking for faster alternatives in the same O(1) class. Every small speed improvement counts when optimizing critical code sections.

The `defaultdict` type takes a factory as a parameter and can therefore be used with built-in types or classes whose constructors do not take arguments. The following code snippet is an example from the official documentation that demonstrates how to use `defaultdict` for counting:

```
>>> from collections import defaultdict
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
...
>>> list(d.items())
[('i', 4), ('p', 2), ('s', 4), ('m', 1)]
```

For this particular example (counting unique elements), the `collections` module also offers a special `Counter` class. It can be used to query efficiently for a number of top elements.

namedtuple

namedtuple is a class factory that takes a name of a type with a list of attributes and creates a class out of it. The new class can then be used to instantiate tuple-like objects and also provides accessors for their elements, as follows:

```
>>> from collections import namedtuple
>>> Customer = namedtuple(
...     'Customer',
...     'firstname lastname'
... )
>>> c = Customer('Tarek', 'Ziadé')
>>> c.firstname
'Tarek'
```

As shown in the preceding example, it can be used to create records that are easier to write compared to a custom class that may require boilerplate code to initialize values. On the other hand, it is based on a tuple, so gaining access to its elements by means of an index is a quick process. The generated class can also be sub-classed to add more operations.

The advantage of using namedtuple over other data types may not be obvious at first glance. The main advantage is that it is easier to use, understand, and interpret than ordinary tuples. Tuple indexes don't carry any semantics, so it is great to be able to access tuple elements by attributes. Note that you could also get the same benefits from dictionaries that have an O(1) average complexity of get and set operations.

The main advantage in terms of performance is that namedtuple is still a flavor of tuple. This means that it is immutable, so the underlying array storage is allocated for the necessary size. Dictionaries, on the other hand, need to use overallocation in the internal hash table to ensure the low-average complexity of get/set operations. So, namedtuple wins over dict in terms of memory efficiency.

> Similar micro memory optimization can be done on user-defined classes using slots. Slots were discussed in *Chapter 4*, *Python in Comparison with Other Languages*.

The fact that `namedtuple` is based on a tuple may also be beneficial for performance. Its elements may be accessed by an integer index, as in other simple sequence objects—lists and tuples. This operation is both simple and fast. In the case of `dict` or custom class instances (that use dictionaries for storing attributes), the element access requires a hash table lookup. Dictionaries are highly optimized to ensure good performance independently from collection size, but as mentioned, O(1) complexity is actually only regarded as average complexity. The actual, amortized worst-case complexity for set/get operations in `dict` is O(n). The real amount of work required to perform such an operation is dependent on both collection size and history. In sections of code that are critical for performance, it may be wise to use lists or tuples over dictionaries, as they are more predictable. In such a situation, `namedtuple` is a great type that combines the following advantages of dictionaries and tuples:

- In sections where readability is more important, the attribute access may be preferred.
- In performance-critical sections, elements may be accessed by their indexes.

> Named tuples can be a useful optimization tool, but when readability matters, usually, data classes are a better choice for storing struct-like data. We've discussed data classes and their advantages in *Chapter 4*, *Python in Comparison with Other Languages*.

Reduced complexity can be achieved by storing data in an efficient data structure that works well with the way the algorithm will use it. That said, when the solution is not obvious, you should consider dropping and rewriting the incriminating part of the code instead of sacrificing the readability for the sake of performance. Often, Python code can be both readable and fast, so try to find a good way to perform the work instead of trying to work around a flawed design.

But sometimes, the problem we are trying to solve doesn't have an efficient solution or we don't have a good and performant structure at hand. In such situations, it is worth considering some architectural trade-offs. We will discuss examples of such trade-offs in the next section.

Leveraging architectural trade-offs

When your code can no longer be improved by reducing the complexity or choosing a proper data structure, a good approach may be to consider a trade-off. If we review users' problems and define what is really important to them, we can often relax some of the application's requirements. Performance can often be improved by doing the following:

- Replacing exact solution algorithms with heuristics and approximation algorithms
- Deferring some work to delayed task queues
- Using probabilistic data structures

Let's move on and take a look at these improvement methods.

Using heuristics and approximation algorithms

Some algorithmic problems simply don't have good state-of-the-art solutions that could run within a time span that would be acceptable to the user.

For example, consider a program that deals with complex optimization problems, such as the **Traveling Salesman Problem** (**TSP**) or the **Vehicle Routing Problem** (**VRP**). Both problems are **NP-hard problems** in combinatorial optimization. Exact low-complexity algorithms for these problems are not known; this means that the size of a problem that can be solved practically is greatly limited. For larger inputs, it is unlikely that you'll be able to provide the correct solution in time.

Fortunately, a user will likely not be interested in the best possible solution, but one that is good enough and can be obtained in a timely manner. In these cases, it makes sense to use heuristics or approximation algorithms whenever they provide acceptable results:

- **Heuristics** solve given problems by trading optimality, completeness, accuracy, or precision for speed. Thus, it may be hard to prove the quality of their solutions compared to the result of exact algorithms.
- **Approximation algorithms** are similar in idea to heuristics, but, unlike heuristics, have provable solution quality and runtime bounds.

There are many known good heuristics and approximation algorithms that can solve extremely large TSP or VRP problems within a reasonable amount of time. They also have a high probability of producing good results, just 2-5% from the optimal solution.

Another good thing about heuristics is that they don't always need to be constructed from scratch for every new problem that arises. Their higher-level versions, called **metaheuristics**, provide strategies for solving mathematical optimization problems that are not problem-specific and can thus be applied in many situations. Popular metaheuristic algorithms include the following:

- **Simulated annealing**, which imitates the physical processes happening during annealing in metallurgy (controlled heating and cooling of materials)
- **Evolutionary computation**, which is inspired by biological evolutions and uses evolutionary mechanisms, such as mutation, reproduction, recombination, and selection, to efficiently search a large area of solutions in complex problems
- **Genetic algorithms**, which are a specialized form of evolutionary computation that represent possible problem solutions as sets of genotypes and perform genetic transformations, such as crossover and mutations, to produce better results
- **Tabu search**, which is a general problem search technique that introduces prohibited search paths (taboos) to reduce the probability of algorithm settling in local optimums
- **Ant colony optimization**, which imitates the behavior of ants in a colony when searching through the possible space of problem solutions

Heuristic and approximation algorithms are viable optimization techniques when the majority of the performance happens in a single algorithmic task of the application. But often, performance issues are caused by general system architecture and communication links between different system components.

A common architectural trade-off that improves the perceived performance of complex applications involves the use of task queues and delayed processing.

Using task queues and delayed processing

Sometimes, it's not about doing too much, but about doing things at the right time. A common example that's often mentioned in literature is sending emails within a web application. In this case, increased response times for HTTP requests may not necessarily translate to your code implementation. The response time may instead be dominated by a third-party service, such as a remote email server. So, can you ever successfully optimize your application if it spends most of its time waiting on other services to reply?

The answer is both yes and no. If you don't have any control over a service that is the main contributor to your processing time, and there is no faster solution you can use, you cannot speed up the service any further. A simple example of processing an HTTP request that results in sending an email is presented in *Figure 13.6*:
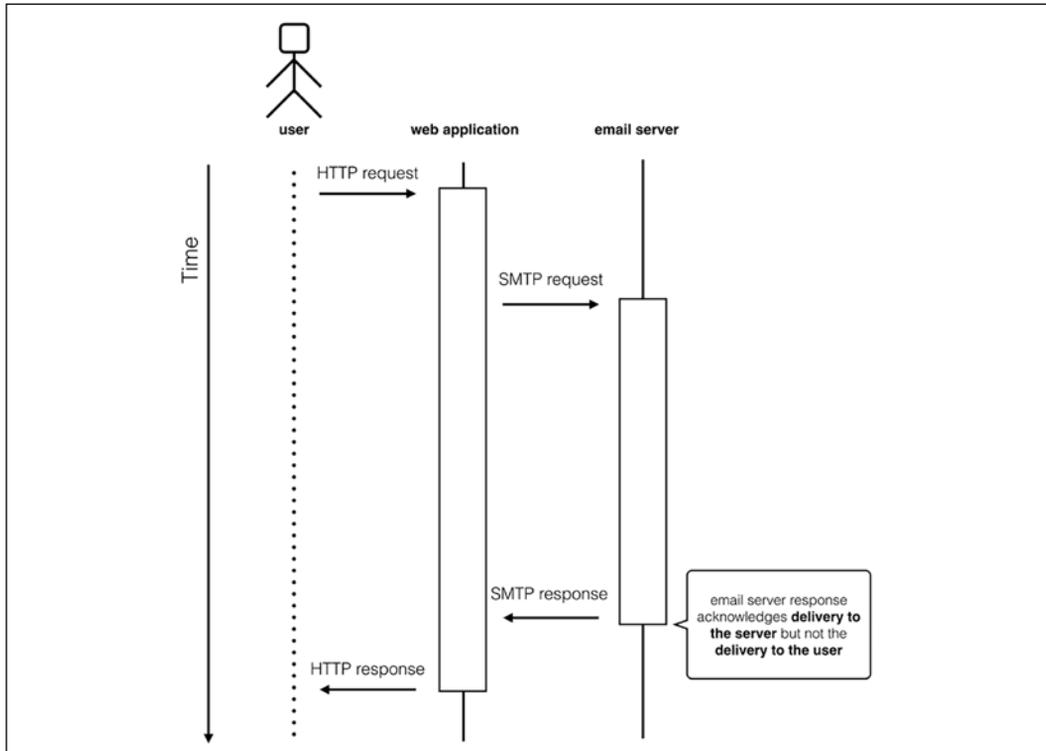
Figure 13.6: An example of synchronous email delivery in a web application

If your code relies on third-party services, you often cannot reduce the waiting time. But you can change the way users will perceive it.

The usual solution to this is kind of problem is to use message or task queues (see *Figure 13.7*). When you need to do something that could take an indefinite amount of time, add it to the queue of work that needs to be done and immediately tell the user that their request was accepted. This is why sending emails is such a great example: emails are already task queues! If you submit a new message to an email server using the SMTP protocol, a successful response does not mean your email was delivered to the addressee — it means that the email was delivered to the email server. If a response from the server does not guarantee that an email was delivered, you don't need to wait for it in order to generate an HTTP response for the user:
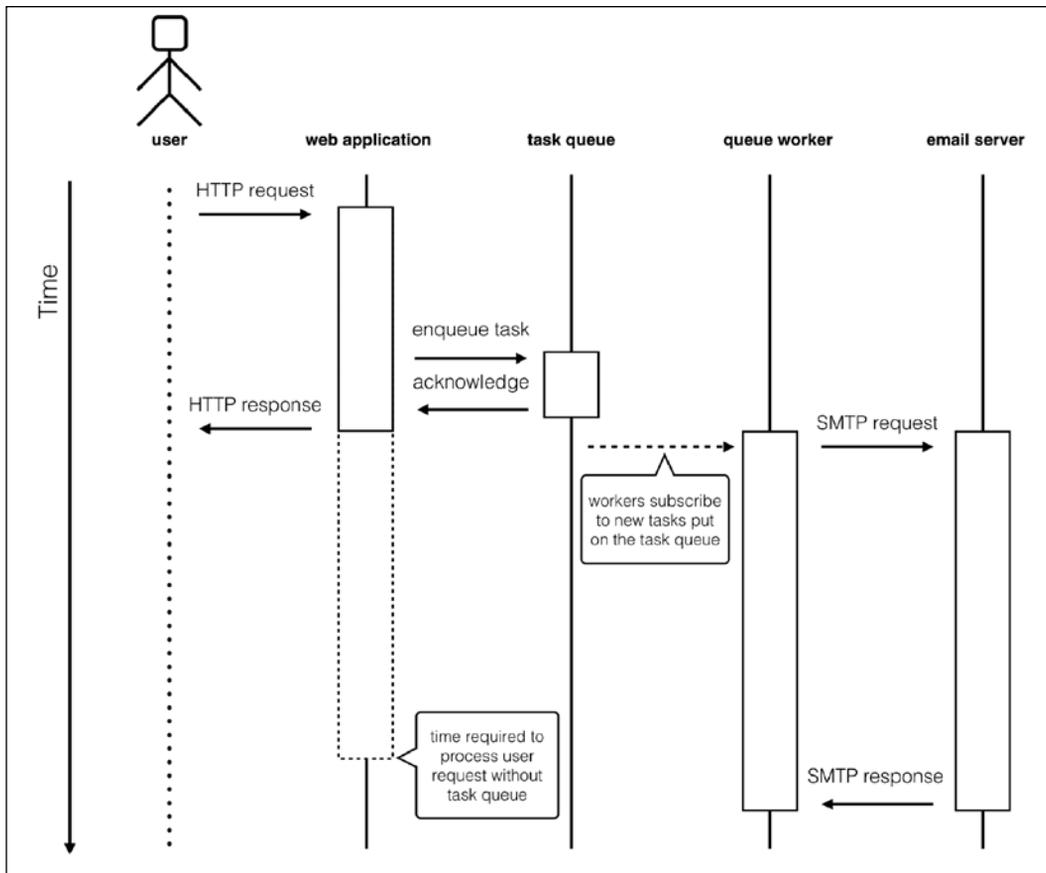
Figure 13.7: An example of asynchronous email delivery in a web application

Even if your email server is responding at blazing speed, you may need some more time to generate the message that needs to be sent. Are you generating yearly reports in an XLS format? Or are you delivering invoices via PDF files? If you use an email transport system that is already asynchronous, then you can put the whole message generation task to the message processing system. If you cannot guarantee the exact time of delivery, then you should not bother generating your deliverables synchronously.

The correct usage of task and message queues in critical sections of an application can also give you other benefits, including the following:

- Web workers that serve HTTP requests will be relieved from additional work and will be able to process requests faster. This means that you will be able to process more requests with the same resources and thereby handle greater loads.

- Message queues are generally more immune to transient failures of external services. For instance, if your database or email server times out from time to time, you can always re-queue the currently processed task and retry it later.

- With a good message queue implementation, you can easily distribute work on multiple machines. This approach may improve the scalability of some of your application components.

As you can see in *Figure 13.7*, adding an asynchronous task process to your application inevitably increases the complexity of the whole system's architecture. You will need to set up some new backing services (a message queue such as RabbitMQ) and create workers that will be able to process asynchronous jobs. Fortunately, there are some popular tools available for building distributed task queues.

> A popular Python tool for asynchronous job handling is **Celery**. Celery is a fully fledged task queue framework with the support of multiple message brokers that also allows for the scheduled execution of tasks. It can even replace your cron jobs. You can read more about Celery at `http://www.celeryproject.org/`.
>
> If you need something simpler, then **RQ** might be a good alternative. RQ is a lot simpler than Celery and uses Redis key-value storage as its message broker **Redis Queue** (**RQ**). You can read more about RQ at `http://python-rq.org/`.

Although there are some good and battle-hardened tools available, you should always carefully consider your approach to task queues. Not every kind of task should be processed in queues. While queues are good at solving a number of issues, they can also introduce the following problems:

- The increased complexity of system architecture
- The possibility of processing a single message more than once

- More services to maintain and monitor
- Larger processing delays
- More difficult logging

A completely different approach to architectural tradeoffs involves the use of non-deterministic/probabilistic data structures.

Using probabilistic data structures

Probabilistic data structures are structures that are designed to store collections of values in a way that allows you to answer specific questions within time or resource constraints that would otherwise be impossible. A common example would be efficiently storing unique view counts in a large video streaming platform such as YouTube that has billions of videos and billions of users. Having naïve implementation that stores exact information on who watched what would take a tremendous amount of memory and would probably be hard to operate efficiently. When the problem is that big, it may be necessary to consider the use of probabilistic data structures.

The most important feature of probabilistic data structures is that the answers they give are only probable to be true; in other words, they are just approximations of real values. The probability of a correct answer can be easily estimated, however. Despite not always giving the correct answer, probabilistic data structures can still be useful if there is some room for potential error.

There are a lot of data structures with such probabilistic properties. Each one of them solves specific problems, but due to their stochastic nature, they cannot be used in every situation. As a practical example, we'll talk about one of the more popular structures, **HLL** (**HyperLogLog**).

HLL is an algorithm that approximates the number of distinct elements in a multiset. With ordinary sets, if you want to know the number of unique elements, you need to store all of them. This is obviously impractical for very large datasets. HLL is distinct from the classical way of implementing sets as programming data structures.

Without digging into implementation details, let's say that it only concentrates on providing an approximation of set cardinality; real values are never stored. They cannot be retrieved, iterated, or tested for membership. HLL trades accuracy and correctness for time complexity and size in memory. For instance, the Redis implementation of HLL takes only 12k bytes with a standard error of 0.81%, with no practical limit on collection size.

Using probabilistic data structures is an interesting way of solving performance problems. In most cases, it is about trading off some accuracy for faster processing or more efficient resource usage. It does not always need to do so, however. Probabilistic data structures are often used in key/value storage systems to speed up key lookups. One of the most popular techniques that's used in such systems is called an **Approximate Member Query** (**AMQ**). An interesting probabilistic data structure that is often used for this purpose is the Bloom filter.

In the next section, we'll take a look at caching.

Caching

When some of your application functions take too long to compute, a useful technique to consider is caching. Caching saves the return values of function calls, database queries, HTTP requests, and so on for future reference. The result of a function or method that is expensive to run can be cached as long as one of the following conditions is met:

- The function is deterministic and the results have the same value every time, given the same input.
- The return value of the function is non-deterministic, but continues to be useful and valid for certain periods of time.

In other words, a deterministic function always returns the same result for the same set of arguments, whereas a non-deterministic function returns results that may vary in time. The caching of both types of results usually greatly reduces the time of computation and allows you to save a lot of computing resources.

The most important requirement for any caching solution is a storage system that allows you to retrieve saved values significantly faster than it takes to calculate them. Good candidates for caching are usually the following:

- Results from callables that query databases
- Results from callables that render static values, such as file content, web requests, or PDF rendering
- Results from deterministic callables that perform complex calculations
- Global mappings that keep track of values with expiration times, such as web session objects
- Results that need to be accessed often and quickly

Another important use case for caching is when saving results from third-party APIs served over the web. This may greatly improve application performance by cutting off network latencies, but it also allows you to save money if you are billed for every request to an API.

Depending on your application architecture, the cache can be implemented in many ways and with various levels of complexity. There are many ways of providing a cache, and complex applications can use different approaches on different levels of the application architecture stack. Sometimes, a cache may be as simple as a single global data structure (usually a dictionary) that is kept in the process memory. In other situations, you may want to set up a dedicated caching service that will run on carefully tailored hardware. The following sections will provide you with basic information on the most popular caching approaches, guiding you through some common use cases as well as the common pitfalls.

So, let's move on and see what deterministic caching is.

Deterministic caching

Deterministic functions are the easiest and safest use case for caching. Deterministic functions always return the same value if given the same input, so caching can generally store their results indefinitely. The only limitation to this approach is storage capacity. The simplest way to cache results is to put them into process memory, as this is usually the fastest place to retrieve data from. Such a technique is often called **memoization**.

Memoization is very useful when optimizing recursive functions that may need to evaluate the same input multiple times. We already discussed recursive implementations for the Fibonacci sequence in *Chapter 9*, *Bridging Python with C and C++*. Earlier on in this book, we tried to improve the performance of our program with C and Cython. Now, we will try to achieve the same goal by simpler means—through caching. Before we do that, let's first recall the code for the `fibonacci()` function, as follows:

```
def fibonacci(n):
    if n < 2:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

As we can see, `fibonacci()` is a recursive function that calls itself twice if the input value is more than two. This makes it highly inefficient. The runtime complexity is $O(2^n)$ and its execution creates a very deep and vast call tree. For a large input value, this function will take a long time to execute. There is also is a high chance that it will exceed the maximum recursion limit of the Python interpreter.

If you take a closer look at the following *Figure 13.8*, which presents an example call tree of the `fibonacci()` function, you will see that it evaluates many of the intermediate results multiple times. A lot of time and resources can be saved if we reuse some of these values:
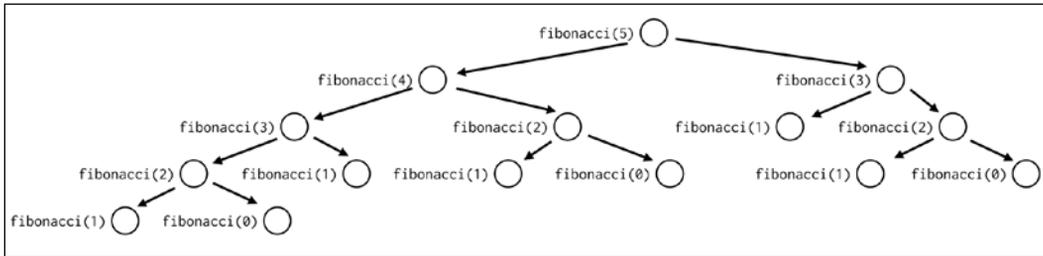
Figure 13.8: A call tree for the fibonacci(5) execution

An example of a simple memoization attempt would be to store the results of previous runs in a dictionary and to retrieve them if they are available.

Both the recursive calls in the `fibonacci()` function are contained in a single line of code, as follows:

```
return fibonacci(n - 1) + fibonacci(n - 2)
```

We know that Python evaluates instructions from left to right. This means that, in this situation, the call to the function with a higher argument value will be executed before the call to the function with a lower argument value. Thanks to this, we can provide memoization by constructing a very simple decorator, as follows:

```
def memoize(function):
    call_cache = {}

    def memoized(argument):
        try:
            return call_cache[argument]
        except KeyError:
            return call_cache.setdefault(
                argument, function(argument)
            )

    return memoized
```

We can then apply it to the `fibonacci()` function as follows:

```
@memoize
def fibonacci(n):
```

```
    if n < 2:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

We've used the dictionary on the closure of the `memoize()` decorator as a simple storage solution from cached values. Saving and retrieving values in the preceding data structure has an average complexity of O(1). This should greatly reduce the overall complexity of the memoized function. Every unique function call will be evaluated only once. The call tree of such an updated function is presented in *Figure 13.9*. Even without mathematical proof, we can visually deduce that we have reduced the complexity of the `fibonacci()` function from the very expensive $O(2^n)$ to the linear $O(n)$:
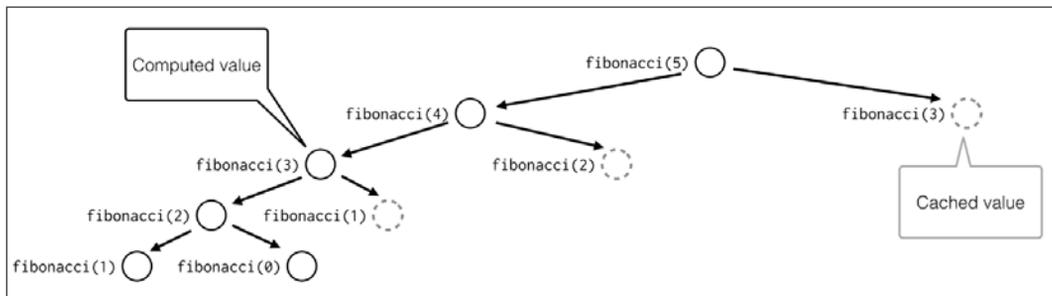
Figure 13.9: A call tree for the fibonacci(5) execution with memorization

The implementation of our `memoize()` decorator is, of course, not perfect. It worked well for the preceding example, but it isn't a reusable piece of software. If you need to memoize functions with multiple arguments, or want to control the size of your cache, you will require something more generic.

Luckily, the Python standard library provides a very simple and reusable utility that can be used in most cases of deterministic caching. This utility is the `lru_cache()` decorator from the `functools` module. The name comes from the **LRU (Last Recently Used) algorithm**. The following additional parameters allow for finer control of memoization behavior:

- `maxsize`: This sets the maximum size of the cache. The `None` value means no limit at all.

- `typed`: This defines whether values of different types that compare as equal should be mapped to the same result.

The usage of `lru_cache()` in our Fibonacci sequence example would be as follows:

```python
from functools import lru_cache


@lru_cache(None)
def fibonacci(n):
    if n < 2:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

In the next section, we will take a look at non-deterministic caching.

Non-deterministic caching

Caching non-deterministic functions is trickier than memoization. Since every execution of such a function may provide different results, it is usually impossible to use previous values for an arbitrarily long amount of time. What you need to do instead is to decide for how long a cached value can be considered valid. After a defined period of time passes, the stored results are considered stale, and the cache will need to be refreshed with a new value.

So, in other words, non-deterministic caching is performed in any situation where pre-computed results are used temporarily. Cached non-deterministic functions often depend on some external state that is hard to track inside your application code. Typical examples of components include the following:

- Relational databases, or generally any type of structured data storage engine
- Third-party services accessible through a network connection (web APIs)
- Filesystems

Note that such an implementation is a trade-off. If you resign from running part of your code whenever necessary and instead use historical results, you are risking using data that is stale or represents an inconsistent state of your system. In this case, you are trading accuracy and/or completeness for speed and performance.

Of course, such caching is only efficient as long as the time taken to interact with the cache is less than the time the cached function takes to execute. If it's faster to simply recalculate the value, by all means do so! That's why setting up a cache has to be done only if it's worth it; setting it up properly has a cost.

Things that can actually be cached are usually the results of interactions with other components of your system. For instance, if you want to save time and resources when communicating with the database, it is worth caching frequent and expensive queries. If you want to reduce the number of I/O operations, you may want to cache the content of files that are most frequently accessed or responses from external APIs.

Techniques for caching non-deterministic functions are actually very similar to those used in caching deterministic ones. The most notable difference is that they usually require the option of invalidating cached values by their age. This means that the `lru_cache()` decorator from the `functools` module has limited use; however, it should not be too difficult to extend this function to provide the expiration feature. As this is a very common problem that has been solved numerous times by many developers, you should be able to find multiple libraries on PyPI that have been designed for caching non-deterministic values.

Using local process memory is a fast way to cache values, but every process maintains its own cache. If you have a lot of processes, independent caches can take a substantial amount of your memory. In distributed systems, it is common to use dedicated cache services.

Cache services

Although non-deterministic caching can be implemented using local process memory, it is actually rarely done that way in a distributed system. That's because a cache needs to be duplicated for every service and this is often a waste of resources. Moreover, multiple process instances can have different cache values, and this may lead to data inconsistencies.

If you run into a situation where non-deterministic caching is your preferred solution to performance problems, you may well need something more. Usually, non-deterministic caching is your must-have solution when you need to serve data or a service to multiple users at the same time.

Sooner or later, you may also need to ensure that users can be served concurrently. While local memory provides a way of sharing data between multiple threads, threading may not be the best concurrency model for every application. It does not scale well, so you will eventually need to run your application as multiple processes.

If you are lucky enough, you may be able to run your application on hundreds or thousands of machines. If you would like to store cached values in local memory in this scenario, your cache will need to be duplicated on every process that requires it. This is not just a total waste of resources—if every process has its own cache, which is already a trade-off between speed and consistency, how can you guarantee that all caches are consistent with each other?

Consistency across subsequent requests is a serious concern, especially for web applications with distributed backends. In complex distributed systems, it is extremely hard to ensure that the user will always be served by the same process hosted on the same machine. It is, of course, doable to some extent, but once you solve that problem, 10 others will pop up.

If you are making an application that needs to serve multiple concurrent users, the best way to handle a non-deterministic cache is to use a dedicated service. With tools such as Redis or Memcached, you allow all of your application processes to share the same cached results. This both reduces the use of precious computing resources and saves you from any problems caused by having too many independent and inconsistent caches.

Caching services such as Memcached are useful for implementing memoization-like caches with states that can be easily shared across multiple processes, and even multiple servers. There is also another way of caching that can be implemented on a system architecture-level, and such an approach is extremely common in applications working over the HTTP protocol. Many elements of a typical HTTP application stack provide elastic caching capabilities that often use mechanisms that are well standardized by the HTTP protocol. This kind of caching can, for instance, take the form of the following:

- Caching reverse-proxy (for example, nginx or Apache): Where a proxy caches full responses from multiple web workers working on the same host

- Caching load balancer (for example, HAProxy): Where a load balancer not only distributes the load over multiple hosts, but also caches their responses

- Content distribution network: Where resources from your servers are cached by a system that also tries to keep them in close geographical proximity to users, thus reducing network roundtrip times

In the next section, we will take a look at Memcached.

Using Memcached

If you want to be serious about caching, Memcached is a very popular and battle-hardened solution. This cache server is used by big applications, including Facebook and Wikipedia, to scale their websites. Among simple caching features, it has clustering capabilities that make it possible for you to set up an efficiently distributed cache system in no time.

Memcached is a multi-platform service, and there are a handful of libraries for communicating with it available in multiple programming languages. Many Python clients differ slightly from one another, but the basic usage is usually the same. The simplest interaction with Memcached almost always consists of the following three methods:

- `set(key, value)`: This saves the value for the given key.
- `get(key)`: This gets the value for the given key if it exists.
- `delete(key)`: This deletes the value under the given key if it exists.

The following code snippet is an example of integration with Memcached using one popular Python package available on PyPI, `pymemcache`:

```python
from pymemcache.client.base import Client

# setup Memcached client running under 11211 port on localhost
client = Client(('localhost', 11211))

# cache some value under some key and expire it after 10 seconds
client.set('some_key', 'some_value', expire=10)

# retrieve value for the same key
result = client.get('some_key')
```

One of the downsides of Memcached is that it is designed to store values as binary blobs. This means that more complex types need to be serialized in order to be successfully stored in Memcached. A common serialization choice for simple data structures is JSON. An example of how to use JSON serialization with `pymemcached` is as follows:

```python
import json
from pymemcache.client.base import Client

def json_serializer(key, value):
    if type(value) == str:
        return value, 1
    return json.dumps(value), 2

def json_deserializer(key, value, flags):
    if flags == 1:
        return value
    if flags == 2:
```

```
        return json.loads(value)
    raise Exception("Unknown serialization format")

client = Client(('localhost', 11211), serializer=json_serializer,
                deserializer=json_deserializer)
client.set('key', {'a':'b', 'c':'d'})
result = client.get('key')
```

Another problem that is very common when working with a caching service that works on the key/value storage principle is how to choose key names.

For cases when you are caching simple function invocations that have basic parameters, the solution is usually simple. Here, you can convert the function name and its arguments into strings and then concatenate them together. The only thing you need to worry about is making sure that there are no collisions between keys that have been created for different functions if you are caching in different places within an application.

A more problematic case is when cached functions have complex arguments that consist of dictionaries or custom classes. In that case, you will need to find a way to convert invocation signatures into cache keys in a consistent manner.

Many caching services (including Memcached) store their cache in RAM to provide the best cache lookup performance. Often, old cache keys can be removed when the working dataset gets too large. The whole cache can be also cleared when the service gets restarted. It is important to take this into account and use caching services to store data that should remain persistent. Sometimes, it may also be necessary to provide a proper cache warmup procedure that will populate the cache with the most common cache entries (for instance, in the case of a service upgrade or new application release).

The last problem is that Memcached, like many other caching services, does not respond well to very long key strings, which may either reduce performance or will simply not fit the hardcoded service limits. For instance, if you cache whole SQL queries, the query strings themselves are generally suitable unique identifiers that can be used as keys. On the other hand, complex queries are generally too long to be stored in a caching service such as Memcached. A common practice is to instead calculate the MD5, SHA, or any other hash function and use that as a cache key instead. The Python standard library has a `hashlib` module that provides implementation for a few popular hash algorithms. One important thing to note when using hashing functions is hash collisions. No hash function guarantees that collisions will never occur, so always be sure to know and mitigate any potential risks.

Summary

In this chapter, we've learned about the optimization process, from identifying possible bottlenecks, through common profiling techniques, to useful optimization techniques that can be applied to a wide variety of performance problems.

This chapter concludes the book, the same way that optimization usually concludes application development cycles. We perform an optimization process on applications that are known to work well. That's why it is important to have proper methodologies and processes set in place that will ensure that our application continues to work properly.

While optimization often concentrates on reducing algorithmic and computational complexity, it can increase different kinds of complexity. Optimized applications are often harder to read and understand, and so are more complex in terms of readability and maintainability. Architectural trade-offs often rely on introducing dedicated services or using solutions that sacrifice part of application correctness or accuracy. Applications that leverage such architectural trade-offs almost always have more complex architectures.

Code optimization, like every other development practice, requires skill and expertise. Part of that expertise is knowing what the balance between various development processes and activities is. Sometimes, minor optimizations are not worth doing at all. Sometimes, it is worth breaking a few rules to satisfy business needs. That is why, in this book, we tried to capture a holistic view of the entire development process of the application. You may not always be doing everything on your own, but knowing how applications are built, maintained, tested, released, observed, and optimized will allow you to know the right balance between each of those activities.

Share your experience

Thank you for taking the time to read this book. If you enjoyed this book, help others to find it. Leave a review at `https://www.amazon.com/dp/1801071101`

---

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

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:
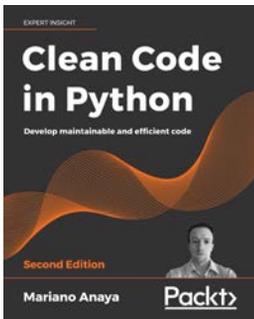
Clean Code in Python - 2nd edition

Mariano Anaya

ISBN: 978-1-80056-021-5

- Set up a productive development environment by leveraging automatic tools
- Leverage the magic methods in Python to write better code, abstracting complexity away and encapsulating details
- Create advanced object-oriented designs using unique features of Python, such as descriptors
- Eliminate duplicated code by creating powerful abstractions using software engineering principles of object-oriented design
- Create Python-specific solutions using decorators and descriptors
- Refactor code effectively with the help of unit tests
- Build the foundations for solid architecture with a clean code base as its cornerstone
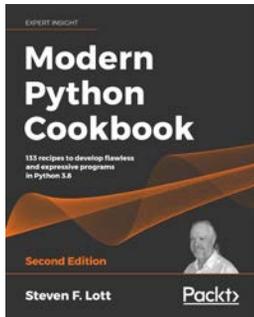
Modern Python Cookbook – Second Edition

Steven F. Lott

ISBN: 978-1-80020-745-5

- See the intricate details of the Python syntax and how to use it to your advantage
- Improve your coding with Python readability through functions
- Manipulate data effectively using built-in data structures
- Get acquainted with advanced programming techniques in Python
- Equip yourself with functional and statistical programming features
- Write proper tests to be sure a program works as advertised
- Integrate application software using Python

Index

Symbols

A

B

S

sdist distributions 445, 447
secrets module 100, 101
semantic versioning 455
SemVer standard 455
  for semantic versioning 455, 456
SemVer standard, numerical segments
  MAJOR segment 455
  MINOR segment 455
  PATCH segment 455
Sentry
  reference link 515
serialized 221
services
  addressing, of Docker Compose
        environment 48-50
session scope 394
setter 125
setup.py script 438, 439
shared fixtures 394
shebang 480
shells
  incorporating, in programs 65, 66
  incorporating, in scripts 65, 66
signal 286
signal-driven programming 286
single-dispatch functions 149, 151
singleton design pattern 208
site-packages directories 19
soft keyword 103
software library
  distributing 434
  packaging 434
Solid State Drives (SSDs) 549
source-compatible 331
source distributions 437
span 534
Sphinx 436
  reference link 436
Stackless Python 255
standalone executables
  creating 480
  need for 481
  popular tools 481, 482
standard commands 439
standard logging library

built-in logging handlers 499-505
statements 79
static type analysis 419, 420
statistical profiler 551
stolen references 354
strings
  formatting, with formatted string
        literals 98, 99
structlog package
  reference link 510
structural error 185
structural pattern matching 103-107
style fixers 415-419
subject-based style 280-286
super-classes
  accessing 112, 113
system components, Python logging
  filters 497
  formatters 497
  handlers 497
  loggers 497
system-level environment isolation 32-34
system-level isolation
  versus application-level isolation 23

T

test coverage 411-415
Test-Driven Development (TDD)
  principles 379-381
testing utilities 427
  realistic data values, faking 427-429
  time values, faking 429, 430
test parameterization 389-392
tests
  writing, with pytest 381-389
threaded() function 217
thread errors
  dealing with 238-241
thread pool
  using 231-235
throttling 241-244
token bucket 242
topic-based style 280, 286-288
topics 286
tracing garbage collection 352
tracking pixels 197