Expert Python Programming

- Third Edition

Become a master in Python by learning coding best practices
and advanced programming concepts in Python 3.7

Michał Jaworski and Tarek Ziadé

Expert Python Programming
Third Edition

Become a master in Python by learning coding best practices and advanced programming concepts in Python 3.7

Michał Jaworski
Tarek Ziadé

Packt>

BIRMINGHAM - MUMBAI

Expert Python Programming
Third Edition

To my beloved wife, Oliwia, for her love, inspiration, and her endless patience.

To my loyal friends, Piotr, Daniel, and Paweł, for their support.

To my mother, for introducing me to the amazing world of programming.

– Michał Jaworski

`mapt.io`

Mapt is an online digital library that gives you full access to over 5,000 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

- Improve your learning with Skill Plans built especially for you

- Get a free eBook or video every month

- Mapt is fully searchable

- Copy and paste, print, and bookmark content

Packt.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.packt.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `customercare@packtpub.com` for more details.

At `www.packt.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the authors

Michał Jaworski has 10 years' of professional experience in Python. He has been in various roles at different companies, from an ordinary full-stack developer, through software architect, to VP of engineering in a fast-paced start-up company. He is currently a senior backend engineer at Showpad. He is highly experienced in designing high-performance distributed services. He is also an active contributor to many open source Python projects.

Tarek Ziadé is a Python developer located in the countryside near Dijon, France. He works at Mozilla in the services team. He founded a French Python user group called Afpy, and has written several books about Python in French and English. When he is not hacking on his computer or hanging out with his family, he's spending time between his two passions, running and playing the trumpet.

You can visit his personal blog (*Fetchez le Python*) and follow him on Twitter (`tarek_ziade`).

About the reviewer

Cody Jackson is a disabled military veteran, the founder of Socius Consulting, an IT and business management consulting company in San Antonio, and a co-founder of Top Men Technologies. He is currently employed at CACI International as the lead ICS/SCADA modeling and simulations engineer. He has been involved in the tech industry since 1994, when he joined the Navy as a nuclear chemist and radcon technician. Prior to CACI, he worked at ECPI University as a computer information systems adjunct professor. A self-taught Python programmer, he is the author of *Learning to Program Using Python* and *Secret Recipes of the Python Ninja*. He holds an Associate in Science degree, a Bachelor of Science degree, and a Master of Science degree.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Section 2: Python Craftsmanship

Section 4: Need for Speed

Preface

Python is a dynamic programming language, used in a wide range of domains thanks to its simple yet powerful nature. Although writing Python code is easy, making it readable, reusable, and easy to maintain is challenging. Complete with best practices, useful tools, and standards implemented by professional Python developers, the third version of *Expert Python Programming* will help you overcome this challenge.

The book will start by taking you through the new features in Python 3.7. You'll learn the Python syntax and understand how to apply advanced object-oriented concepts and mechanisms. You'll also explore different approaches to implement metaprogramming. This book will guide you in following best naming practices when writing packages, and creating standalone executables easily, alongside using powerful tools such as `buildout` and `virtualenv` to deploy code on remote servers. You'll discover how to create useful Python extensions with C, C++, Cython, and Pyrex. Furthermore, learning about code management tools, writing clear documentation, and test-driven development will help you write clean code.

By the end of the book, you will have become an expert in writing efficient and maintainable Python code.

Who this book is for

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

It does not mean that hobbyists won't find anything interesting. This book should be 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 Python 3.7 for those who are still a bit behind and continue to use Python version 2.7 or older.

Finally, the groups that should benefit most from reading this book are web developers and backend engineers. This is because of two topics featured in here that are especially important in their areas of work: reliable code deployments and concurrency.

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, why it is changing, and also why these facts are important for anyone who wants to call themselves a Python professional. We will also take a look at the most popular and canonical ways for working on written in Python—popular productivity tools and conventions that are de facto standards now.

`Chapter 2`, *Modern Python Development Environments*, describes modern ways of setting up repeatable and consistent development environments for Python programmers. We will concentrate on two popular tools for environment isolation: `virtualenv`-type environments and Docker containers.

`Chapter 3`, *Modern Syntax Elements – Below the Class Level*, focuses on best practices for writing code in Python (language idioms) and also provides a summary of selected elements of Python syntax that may be new for intermediate Python users or those experienced with older versions of Python. We will also take a look at useful notes about internal CPython-type implementations and their computational complexities as a rationale for provided idioms.

`Chapter 4`, *Modern Syntax Elements – Above the Class Level*, covers more advanced object-oriented concepts and mechanisms available in Python.

`Chapter 5`, *Elements of Metaprogramming*, presents an overview of common approaches to metaprogramming available to Python programmers.

`Chapter 6`, *Choosing Good Names*, explains what is the most widely-adopted style guide for Python code (PEP-8) and when and why developers should follow it. We will also take a look at some of the author's general advice for naming things.

`Chapter 7`, *Writing a Package*, 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 an often overlooked topic of Python – standalone executables.

`Chapter 8`, *Deploying Code*, presents some common lightweight tools for deploying Python code on remote servers. Deployment is one of the fields where Python shines are backends for web-based services and applications.

`Chapter 9`, *Python Extensions in Other Languages*, explains why writing extensions in C and C++ for Python can sometimes be a good solution and shows that it is not as hard as it seems, as long as the proper tools are used.

`Chapter 10`, *Managing Code*, describes how to properly manage a code base and why version control systems should be used. We will also leverage the power of version control systems (especially Git) in implementing continuous processes, such as continuous integration and continuous delivery.

`Chapter 11`, *Documenting Your Project*, describes the general rules for writing technical documentation that may be applied to software written in any language, and various tools that are especially useful for creating documentation of your Python code.

`Chapter 12`, *Test-Driven Development*, advocates the usage of test-driven development and provides more information on how to use popular Python tools designed for testing.

`Chapter 13`, *Optimization – Principles and Profiling Techniques*, discusses the most basic rules of optimization that every developer should be aware of. We will also learn how to identify application performance bottlenecks and use common profiling tools.

`Chapter 14`, *Optimization – Some Powerful Techniques*, shows how to use that knowledge to actually make your application run faster or be more efficient in terms of used resources.

`Chapter 15`, *Concurrency*, explains how to implement concurrency in Python using different approaches and libraries.

`Chapter 16`, *Event-Driven and Signal Programming*, describes what event-driven/signal programming is and how it relates to asynchronous programming and different concurrency models. We will present the various approaches to event-driven programming available to Python programmers, along with useful libraries that enable these patterns.

`Chapter 17`, *Useful Design Patterns*, implements a set of useful design patterns and example implementations in Python.

`Appendix A`, *reStructuredText Primer*, provides a brief tutorial on how to use reStructuredText markup language.

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 the first chapter provides a recap on common utilities (such as virtual environments and pip) that are now considered standard tools of professional Python developers.

Download the example code files

You can download the example code files for this book from your account at `www.packt.com`. If you purchased this book elsewhere, you can visit `www.packt.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

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

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

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Expert-Python-Programming-Third-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:
`http://www.packtpub.com/sites/default/files/downloads/9781789808896_ColorImages.pdf`.

Conventions used

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

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. 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:

```
print("hello world")
print "goodbye python2"
```

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

```
$ python3 script.py
```

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`.

Section 1: Before You Start

1

This part prepares the user for the modern Python development routine. It explains how Python has changed over the last few years and what the common development tools used by modern Python programmers are.

The following chapters are included in this section:

- Chapter 1, *Current Status of Python*
- Chapter 2, *Modern Python Development Environments*

Current Status of Python 1

Python is amazing.

For a very long time, one of the most important virtues of Python was interoperability. No matter what operating system you or your customers were using, if a Python interpreter was available for that system, your software that was written in Python would work there. And, most importantly, your software would work the same way. However, that's not uncommon anymore. Modern languages such as Ruby and Java provide similar interoperability capabilities. But, interoperability isn't the most important quality of programming language nowadays. 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 the same no matter the operating system. What is still important is the tools that allow programmers to efficiently write reliable and maintainable software. Fortunately, Python is still one of the languages that allows programmers the most efficiency, and is definitely a smart choice for a company's primary development language.

Python stays relevant for so long because it is constantly evolving. This book is focused on the latest Python 3.7 version, and all code examples are written in this version of the language unless another version is explicitly mentioned. Because Python has a very long history, and there are still programmers using Python 2 on a daily basis, this book starts with a chapter that describes the current *status quo* of Python 3. In this chapter, you'll find how and why Python changes, and will learn how to write software that is compatible with both the historic and latest versions of Python.

In this chapter, we will cover the following topics:

- Where are we now and where we are going to?
- Why and how Python changes
- Being up-to-date with changes to PEP documentation
- Python 3 adoption at the time of writing this book
- The main difference between Python 3 and Python 2
- Not only CPython
- Useful resources

Technical requirements

You can download the latest version of Python from `https://www.python.org/downloads/` for this chapter.

Alternative Python interpreter implementations can be found at the following sites:

- Stackless Python: `https://github.com/stackless-dev/stackless`
- PyPy: `https://pypy.org`
- Jython: `https://www.jython.org`
- IronPython: `https://ironpython.net`
- MicroPython: `https://micropython.org`

The code files for this chapter can be found at `https://github.com/PacktPublishing/Expert-Python-Programming-Third-Edition/tree/master/chapter1`.

Where are we now and where we are going to?

Python history starts somewhere in the late 1980s, but its 1.0 release date was in the year 1994. So, it isn't a young language. There could be a whole timeline of major Python releases mentioned here, but what really matters is a single date: Python 3.0—December 3, 2008.

At the time of writing, almost ten years have passed since the first Python 3 release. It is also seven years since the creation of PEP 404—the official document that *unreleased* Python 2.8 and officially closed the 2.*x* branch. Although a lot of time has passed, there is a specific dichotomy in the Python community—while the language is developing very fast, there is a large group of its users that do not want to move forward with it.

Why and how Python changes

The answer is simple—Python changes because there is such a need. The competition does not sleep. Every few months, a new language pops out, out of nowhere, claiming to solve every problem of all its predecessors. Most projects like these lose the developers' attention very shortly, and their popularity is often driven by sudden hype.

This is a sign of some bigger problem. People design new languages because they find that existing ones do not solve their problems in the best way possible. It would be silly to not recognize such a need. Also, more and more widespread usage of Python shows that it could, and should, be improved on in many places.

Many improvements in Python are driven by the needs of particular fields where it is being used. The most significant one is web development. Thanks to the ever-increasing demand for speed and performance in this area, we've seen that ways to deal with concurrency in Python have been drastically improved over the time.

Other changes are simply caused by the age and maturity of the Python project. Throughout the years, it collected some of the clutter in the form of disorganized and redundant standard library modules, or some bad design decisions. First, the Python 3 release aimed to bring with it a major cleanup and refreshment to the language. Unfortunately, time showed that this plan backfired a bit. For a long time, Python 3 was treated by many developers only like a curiosity. Hopefully, this is changing.

Being up-to-date with changes by following PEP documents

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

It 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. The whole purpose, format, and workflow around these documents is also standardized in the form of a PEP—precisely the PEP 1 document (`http://www.python.org/dev/peps/pep-0001`).

PEP documentation is very important for Python, and, depending on the topic, they serve 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 are available in a *living* PEP 0 document (`https://www.python.org/dev/peps/`). Since they are easily accessible in one place, and the actual URL is also very easy to guess, they are usually referred to by the number in the book.

The PEP 0 document is a great source of information for those who are wondering what direction Python language is heading in, but do not have time to track every discussion on Python mailing lists. It shows which documents were already accepted but not yet implemented, and also which are still under consideration.

PEPs also serve additional purposes. 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 extensive answer is already available in specific PEP documents where such a feature was already mentioned. There is a lot of PEP documentation describing Python language features that were proposed but not accepted. This documentation is left as a historical reference.

Python 3 adoption at the time of writing this book

So, thanks to new, exciting features, is Python 3 well adopted among its community? It's hard to say. The once-popular page, Python 3 Wall of Superpowers (`https://python3wos.appspot.com`), that tracked the compatibility of the most popular packages with the Python 3 branch was, at the beginning, named Python 3 Wall of Shame.

The site is no longer maintained, but in the list from the last time it was updated, on April 22, 2018, it shows that exactly 191 from 200 of the most popular Python packages at that time were compatible within Python 3. So, we can see that Python 3 seems to be finally well-adopted in the community of open source Python programmers. Still, this does not mean that all teams building their applications are finally using Python 3. At least, since most of the popular Python packages are available in Python 3, the popular excuse *packages that we use have not been ported yet* is no longer valid.

The main reason for such a situation is that porting the existing application from Python 2 to Python 3 is always a challenge. There are tools such as `2to3` that can perform automated code translation, but they do not assure that the result will be 100% correct. Also, such translated code may not perform as well as in its original form without manual adjustments. Moving existing complex code bases to Python 3 might involve tremendous effort, and a cost that some organizations may not be able to afford. Fortunately, such costs can be split over time. Some good software architecture design methodologies, such as service-oriented architecture or microservices, can help to achieve this goal gradually. New project components (services or microservices) can be written using the new technology, and existing ones can be ported one at a time.

In the long run, moving to Python 3 can have only beneficial effects on a project. According to PEP 404, there won't be another 2.8 release in the 2.*x* branch of Python, and the official end-of-life for Python 2 is scheduled for 2020. Until that time, we can expect only patch version updates for major security issues, but nothing more. Also, there may be a time in the future when all major projects, such as Django, Flask, and NumPy will drop any 2.*x* compatibility and will be available only in Python 3. Django has already made that step, and since version 2.0.0 was released, it no longer supports Python 2.7.

My personal opinion on this topic can be considered controversial. I think that the best incentive for the community would be to completely drop Python 2 support when creating new packages. This, of course, limits a range of such software, but may be the only right way to change the way of thinking in those who insist on sticking to Python 2.*x*.

We'll take a look at the main differences between Python 3 and Python 2 in the next section.

The main differences between Python 3 and Python 2

It has already been stated that Python 3 breaks backward compatibility with Python 2 on a syntax level. Still, it is not a complete redesign. Also, it does not cause every Python module written for some 2.*x* release to stop working under Python 3. It is possible to write completely cross-compatible code that will run on both major releases without additional tools or techniques, but usually it is possible only for simple applications.

Why should I care?

Despite my personal opinion on Python 2 compatibility that I exposed earlier in this chapter, it is impossible to simply forget about it at this time. There are still some useful packages that are really worth using, but are not likely to be ported in the very near future.

Also, sometimes, we may be constrained by the organization we work in. The existing legacy code may be so complex that porting it is not economically feasible. So, even if we decide to move on and live only in the Python 3 world from now on, it will be impossible to live completely without Python 2 for some time.

Nowadays, it is very hard to call yourself a professional developer without giving something back to the community. So, helping the open source developers add Python 3 compatibility to the existing packages is a good way to pay off the moral debt incurred by using them. This, of course, cannot be done without knowing the differences between Python 2 and Python 3. By the way, this is also a great exercise for those new to Python 3.

The main syntax differences and common pitfalls

The Python documentation is the best reference for differences between every Python release. However, for your convenience, this section summarizes the most important ones. This does not change the fact that the documentation is mandatory reading for those not familiar with Python 3 yet (see `https://docs.python.org/3.0/whatsnew/3.0.html`).

The breaking changes that were introduced by Python 3 can be generally divided into three groups:

- Syntax changes, where some syntax elements were removed/changed and other elements were added
- Changes in the standard library
- Changes in datatypes and collections

Syntax changes

Syntax changes that make it difficult for the existing code to run are the easiest to spot—they will cause the code to not run at all. The Python 3 code that uses new syntax elements will fail to run on Python 2 and vice versa. The elements that were removed from official syntax will make Python 2 code visibly incompatible with Python 3. Any attempt to run the code that has such issues will immediately cause the interpreter to fail, raising a `SyntaxError` exception. Here is an example of the broken script that has exactly two statements, of which none will be executed due to the syntax error:

```
print("hello world")
print "goodbye python2"
```

Its actual result when run on Python 3 is as follows:

```
$ python3 script.py
  File "script.py", line 2
    print "goodbye python2"
                          ^
SyntaxError: Missing parentheses in call to 'print'
```

When it comes to new elements of Python 3 syntax, the total list of differences is a bit long, and any new Python 3.*x* release may add new elements of syntax that will raise such errors on earlier releases of Python (even on the same 3.*x* branch). The most important of them are covered in `Chapter 2`, *Modern Python Development Environments*, and `Chapter 3`, *Modern Syntax Elements – Below the Class Level*, so there is no need to list all of them here.

The list of things that used to work in Python 2 that will cause syntax or functional errors in Python 3 is shorter. Here are the most important backwards incompatible changes:

- `print` is no longer a statement, but a function, so the parenthesis is now obligatory.
- Catching exceptions changed from `except exc, var` to `except exc as var`.

- The `<>` comparison operator has been removed in favor of `!=`.
- `from module import *`
  (`https://docs.python.org/3.0/reference/simple_stmts.html#import`) is now allowed only on module level, and no longer inside the functions.
- `from .[module] import name` is now the only accepted syntax for relative imports. All imports not starting with a dot character are interpreted as absolute imports.
- The `sorted()` function and the list's `sort()` method no longer accept the `cmp` argument. The `key` argument should be used instead.
- Division expressions on integers such as one half return floats. The truncating behavior is achieved through the `//` operator like `1//2`. The good thing is that this can be used with floats too, so `5.0//2.0 == 2.0`.

Changes in the standard library

Breaking changes in the standard library are the second easiest to catch after syntax changes. Each subsequent version of Python adds, deprecates, improves, or completely removes standard library modules. Such a process was also common in the older branches of Python (1.*x* and 2.*x*), so it does not come as a shock in Python 3. In most cases, depending on the module that was removed or reorganized (such as `urlparse` being moved to `urllib.parse`), it will raise exceptions on the import time just after it is interpreted. This makes such issues so easy to catch. In order to be sure that all such issues are covered, full test code coverage is essential. In some cases (for example, when using lazily loaded modules), the issues that are usually noticed at import time will not appear before some modules are used in code as function calls. This is why it is so important to make sure that every line of code is actually executed during tests suite.

> **Lazily loaded modules**
>
> A lazy loaded module is a module that is not loaded on import time. In Python, the `import` statements can be included inside functions, so an import will happen on function call and not on import time. In some cases, such loading of modules may be a reasonable choice, but in most cases, it is a workaround for poorly designed module structure (for example, to avoid circular imports). It is considered bad *code smell* and should be generally avoided. There is no justifiable reason to lazily load standard library modules. In well-structured code, all imports should be grouped at the top of module.

Changes in data types and collections and string literals

Changes in how Python represents datatypes and collections require the most effort when the developer tries to maintain compatibility or simply ports existing code to Python 3. While incompatible syntax or standard library changes are easily noticeable and often easy to fix, changes in collections and types are either non-obvious or require a lot of repetitive work. The list of such changes is long and the official documentation is the best reference.

Still, this section must cover the change in how string literals are treated in Python 3, because it seems to be the most controversial and discussed change in Python 3, despite being a very good move that makes things more explicit.

All string literals are now Unicode, and `bytestring` literals require `b` or `B` prefix. For Python 3.0 and 3.1, the old Unicode `u` prefix (like `u"foo"`) is illegal and will raise a syntax error. Dropping off that prefix was the main reason for most of the controversies. It made it really hard to create code compatible with different branches of Python—Python in version 2.*x* relied on these prefixes in order to create Unicode literals. This prefix was brought back in Python 3.3 to ease the integration process, although it now lacks any syntactic meaning.

The popular tools and techniques used for maintaining cross-version compatibility

Maintaining compatibility between versions of Python is a challenge. It may add a lot of additional work depending on the size of the project, but is definitely doable and worth doing. For packages that are meant to be reused in many environments it is absolutely a must-have. Open source packages without well-defined and tested compatibility bounds are very unlikely to become popular, but closed third-party code that never leaves the company network can also greatly benefit from being tested in different environments.

It should be noted here that, while this part focuses mainly on compatibility between various versions of Python, these approaches apply for maintaining compatibility with external dependencies such as different package versions, binary libraries, systems, or external services.

The whole process can be divided into three main areas, ordered by their importance:

- Defining and documenting target compatibility bounds and how they will be managed
- Testing in every environment and with every dependency version declared as compatible
- Implementing actual compatibility code

Declaration of what is considered compatible is the most important part of the whole process because it gives your code users (developers) the ability to have expectations and make assumptions on how it works and how it can change in the future. Our code can be used as a dependency in different projects that may also strive to manage compatibility, so the ability to reason how it behaves is crucial.

While this book tries to always give a few choices and not to give absolute recommendations on specific options, here is one of the few exceptions. The best way to define how compatibility may change in the future is by using proper approach to versioning numbers using *Semantic Versioning (semver)* (`http://semver.org/`). It describes a broadly accepted standard for marking scope of changes in code by the version specifier, consisting only of three numbers. It also gives some advice on how to handle deprecation policies. Here is an excerpt from its summary (licensed under Creative Commons - CC BY 3.0):

> *Given a version number MAJOR.MINOR.PATCH, increment:*
>
> 1. MAJOR version when you make incompatible API changes,
> 2. MINOR version when you add functionality in a backwards-compatible manner, and
> 3. PATCH *version when you make backward-compatible bug fixes.*
>
> *Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.*

When it comes to testing the sad truth, that is, to be sure that code is compatible with every declared dependency version and in every environment (here Python version), it must be tested in every combination of these. This, of course, may not be possible when the project has a lot of dependencies, because the number of combinations grows rapidly with every new dependency version. So, typically some trade-off needs to be made so that running full compatibility tests does not need to take ages. The selection of tools that help testing in so-called matrixes is presented in `Chapter 12`, *Test-Driven Development*, which discusses testing in general.

> The benefit of using projects that follow semver is that usually what needs to be tested are only major releases, because minor and patch releases are guaranteed to not include backwards incompatible changes. This is, of course, only true if such projects can be trusted to not break such a contract. Unfortunately, mistakes happen to everyone, and backwards incompatible changes happen in a lot of projects, even on patch versions. Still, since semver declares strict compatibility on minor and patch versions, breaking it is considered a bug, so it may be fixed in a patch release.

The implementation of the compatibility layer is the last, and also the least important, step of the process if the bounds of that compatibility are well-defined and rigorously tested. Still, there are some tools and techniques that every programmer interested in such a topic should know.

The most basic is Python's `__future__` module. It backports some features from newer Python releases back into the older ones and takes the form of an import statement:

```
from __future__ import <feature>
```

Features provided by the `future` statements are syntax-related elements that cannot be easily handled by different means. This statement affects only the module where it was used. Here is an example of a Python 2.7 interactive session that brings Unicode literals from Python 3.0:

```
Python 2.7.10 (default, May 23 2015, 09:40:32) [MSC v.1500 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>> type("foo")  # old literals
<type 'str'>
>>> from __future__ import unicode_literals
>>> type("foo")  # now is Unicode
<type 'unicode'>
```

Here is a list of all the available `__future__` statement options that developers concerned with two-thirds compatibility should know:

- `division`: This adds a Python 3 division operator (PEP 238)
- `absolute_import`: This makes every form of an `import` statement not starting from dot character be interpreted as absolute imports (PEP 328)
- `print_function`: This changes a `print` statement into a function call so that parentheses around `print` become mandatory (PEP 3112)
- `unicode_literals`: This makes every string literal be interpreted as Unicode literals (PEP 3112)

A list of the `__future__` statement options is very short, and it covers only a few syntax features. The other things that have changed, such as the `metaclass` syntax (which is an advanced feature that's covered in `Chapter 5`, *Elements of Metaprogramming*), are a lot harder to maintain. Reliable handling of multiple standard library reorganizations also cannot be solved by the `future` statements. Fortunately, there are some tools that aim to provide a consistent layer of ready-to-use compatibility code. The most well-known of these is Six (`https://pypi.python.org/pypi/six/`), which provides a whole common two-thirds compatibility boilerplate as a single module. The other promising, but slightly less popular, tool is the future module (`http://python-future.org/`).

In some situations, developers may not want to include additional dependencies in some small packages. A common practice is the additional module that gathers all the compatibility code, usually named `compat.py`. Here is an example of such compat modules taken from the `python-gmaps` project (`https://github.com/swistakm/python-gmaps`):

```
# -*- coding: utf-8 -*-
"""This module provides compatibility layer for
selected things that have changed across Python versions.
"""
import sys

if sys.version_info < (3, 0, 0):
    import urlparse  # noqa

    def is_string(s):
        """Return True if given value is considered string"""
        return isinstance(s, basestring)

else:
    # note: urlparse was moved to urllib.parse in Python 3
    from urllib import parse as urlparse  # noqa
```

```
def is_string(s):
    """Return True if given value is considered string"""
    return isinstance(s, str)
```

Such `compat.py` modules are popular, even in projects that depend on Six for two-thirds compatibility, because it is a very convenient way to store code that handles compatibility with different versions of packages being used as dependencies.

In the next section, we'll take a look at what CPython is.

Not only CPython

The reference Python interpreter implementation is called **CPython** and, as its name suggests, it is written entirely in the C language. It was always C and probably will be still for a very long time. That's the implementation that most Python programmers choose because it is always up to date with the language specification and is the interpreter that most libraries are tested on. But, besides C, Python interpreter was written in a few other languages. Also, there are some modified versions of CPython interpreter available under different names and tailored exactly for some niche applications. Most of them are a few milestones behind CPython, but provide a great opportunity to use and promote the language in a specific environment.

In this section, we will discuss some of the most prominent and interesting alternative Python implementations.

Why should I care?

There are plenty of alternative Python implementations available. The Python wiki page on that topic (`https://wiki.python.org/moin/PythonImplementations`) features dozens of different language variants, dialects, or implementations of Python interpreter built with something other than C. Some of them implement only a subset of the core language syntax, features, and built-in extensions, but there are at least a few that are almost fully compatible with CPython. The most important thing to know is that, while some of them are just toy projects or experiments, most of them were created to solve some real problems – problems that were either impossible to solve with CPython or required too much of the developer's effort.

Examples of such problems are as follows:

- Running Python code on embedded systems
- Integration with code written for runtime frameworks, such as Java or .NET, or in different languages
- Running Python code in web browsers

The following sections provide a short description of, subjectively, the most popular and up-to-date choices that are currently available for Python programmers.

Stackless Python

Stackless Python advertises itself as an enhanced version of Python. Stackless is named so because it avoids depending on the C call stack for its own stack. It is, in fact, a modified CPython code that also adds some new features that were missing from the core Python implementation at the time Stackless was created. The most important of these are microthreads, which are managed by the interpreter as cheap and lightweight alternatives to ordinary threads, that must depend on system kernel context switching and task scheduling.

The latest available versions are 2.7.15 and 3.6.6 and implement 2.7 and 3.6 versions of Python, respectively. All the additional features provided by Stackless are exposed as a framework within this distribution through the built-in `stackless` module.

Stackless isn't the most popular alternative implementation of Python, but it is worth knowing, because some of the ideas that were introduced in it had a strong impact on the language community. The core switching functionality was extracted from Stackless and published as an independent package named `greenlet`, which is now the basis for many useful libraries and frameworks. Also, most of its features were re-implemented in PyPy—another Python implementation that will be featured later. The official online documentation of Stackless Python can be found at `https://stackless.readthedocs.io` and the project wiki can be found at `https://github.com/stackless-dev/stackless`.

Jython

Jython is a Java implementation of the language. It compiles the code into Java byte code, and allows the developers to seamlessly use Java classes within their Python modules. Jython allows people to use Python as the top-level scripting language on complex application systems, for example, J2EE. It also brings Java applications into the Python world. Making Apache Jackrabbit (which is a document repository API based on JCR; see `http://jackrabbit.apache.org`) available in a Python program is a good example of what Jython allows.

The main differences of Jython compared to the CPython implementation are as follows:

- True Java's garbage collection instead of reference counting
- Lack of **global interpreter lock** (**GIL**) allows better utilization of multiple cores in multi-threaded applications

The main weakness of this implementation of the language is the lack of support for C Python Extension APIs, so no Python extensions written in C will work with Jython.

The latest available version of Jython is Jython 2.7, and this corresponds to the 2.7 version of the language. It is advertised as implementing nearly all of the core Python standard library and using the same regression test suite. Unfortunately, Jython 3.x was never released, and the project can be now safely considered dead. However, Jython is still worth mentioning, even if it is not developed anymore, because it was very unique implementation at the time and had meaningful impact on other alternative Python implementations.

The official project page can be found at `http://www.jython.org`.

IronPython

IronPython brings Python into the .NET Framework. The project is supported by Microsoft, where IronPython's lead developers work. It is quite an important implementation for the promotion of a language. Besides Java, the .NET community is one of the biggest developer communities out there. It is also worth noting that Microsoft provides a set of free development tools that turn Visual Studio into a full-fledged Python IDE. This is distributed as Visual Studio plugins named **Python Tools for Visual Studio** (**PVTS**), and is available as open source code on GitHub (`http://microsoft.github.io/PTVS`).

The latest stable release is 2.7.8, and it is compatible with Python 2.7. Unlike Jython, we can observe active development on both 2.*x* and 3.*x* branches of the interpreter, although Python 3 support still hasn't been officially released yet. Despite the fact that .NET runs primarily on Microsoft Windows, it is also possible to run IronPython on macOS and Linux. This is thanks to Mono, a cross platform, open source .NET implementation.

The main differences and advantages of IronPython compared to CPython are as follows:

- Similar to Jython, the lack of **global interpreter lock** (**GIL**) allows for better utilization of multiple cores in multi-threaded applications
- Code written in C# and other .NET languages can be easily integrated in IronPython and vice versa
- It can be run in all major web browsers through Silverlight (although Microsoft will stop supporting Silverlight in 2021)

When speaking about weaknesses, IronPython seems very similar to Jython, because it does not support the Python/C Extension APIs. This is important for developers who would like to use packages such as NumPy, which are largely based on C extensions. There were a few community attempts to bring the Python/C Extensions API support to IronPython, or at least to provide compatibility for the NumPy package, but unfortunately no project had notable success in that area.

You can learn more about IronPython from its official project page at `http://ironpython.net/`.

PyPy

PyPy is probably the most exciting alternative implementation of Python, as its goal is to rewrite Python in Python. PyPy in the Python interpreter is written in Python. We have a C code layer carrying out the nuts-and-bolts work for CPython. But in PyPy, this C code layer is rewritten in pure Python.

This means that you can change the interpreter's behavior during execution time, and implement code patterns that couldn't be easily done in CPython.

PyPy is currently fully compatible with Python version 2.7.13, while the latest PyPy3 is compatible with Python version 3.5.3.

In the past, PyPy was mostly interesting for theoretical reasons, and it interested those who enjoyed going deep into the details of the language. It was not generally used in production, but this has changed through the years. Nowadays, many benchmarks show that, surprisingly, PyPy is often way faster than the CPython implementation. This project has its own benchmarking site that tracks performance of each version measured using dozens of different benchmarks (refer to `http://speed.pypy.org/`). It clearly shows that PyPy with JIT enabled is usually at least few times faster than CPython. This and other features of PyPy makes more and more developers decide to use PyPy in their production environments.

The main differences of PyPy compared to CPython implementation are as follows:

- Garbage collection used instead of reference counting
- Integrated tracing JIT compiler that allows impressive improvements in performance
- Application-level Stackless features borrowed from Stackless Python

Like almost every other alternative Python implementation, PyPy lacks the full official support of C's Python Extension API. Still, it at least provides some sort of support for C extensions through its CPyExt subsystem, although it is poorly documented and still not feature complete. Also, there is an ongoing effort within the community in porting NumPy to PyPy because it is the most requested feature.

The official PyPy project page can be found at `http://pypy.org`.

MicroPython

MicroPython is one of the youngest alternative implementations on that list, as its first official version was released on May 3, 2014. It is also one of the most interesting implementations. MicroPython is a Python interpreter that was optimized for use on microcontrollers and in very constrained environments. Its small size and multiple optimizations allow it to run in just 256 kilobytes of code space and with just 16 kilobytes of RAM.

The main reference devices that you can test this interpreter on are BBC's micro:bit devices and pyboards, which are simple-to-use microcontroller development boards, that are targeted at teaching programming and electronics.

MicroPython is written in C99 (it's C language standard) and can be built for many hardware architectures, including x86, x86-64, ARM, ARM Thumb, and Xtensa. It is based on Python 3, but due to many syntax differences, it's impossible to say that it is fully compatible with any Python 3.*x* release. It is certainly a dialect of Python 3, with `print()` functions, `async/await` keywords, and many other Python 3 features, but you can't expect that your favorite Python 3 libraries will work properly under that interpreter out of the box.

You can learn more about MicroPython from its official project page at `https://micropython.org`.

Useful resources

The best way to know the status of Python is to stay informed about what's new and to constantly read Python-related resources. The web is full of such resources. The most important and obvious ones were already mentioned earlier, but here they are repeated to keep this list consistent:

- **Python documentation**
- **Python Package Index (PyPI)**
- **PEP 0 – Index of Python Enhancement Proposals (PEPs)**

The other resources, such as books and tutorials, are useful, but often get outdated very fast. What does not get outdated are the resources that are actively curated by the community or released periodically. The few that are worth recommending are as follows:

- Awesome Python (`https://github.com/vinta/awesome-python`) includes a curated list of popular packages and frameworks.
- r/Python (`https://www.reddit.com/r/Python/`) is a Python subreddit where you can find news and interesting questions about Python posted by many members of Python community every day.
- Python Weekly (`http://www.pythonweekly.com/`) is a popular newsletter that delivers to its subscriber's 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. Due to its nature, the content of that newsletter often overlaps with Python Weekly, but sometimes you can find something unique that hasn't been posted elsewhere.

These resources will provide you with tons of additional reading for countless hours.

Summary

This chapter concentrated on the current status of Python and the process of change that was visible throughout the history of that language. We started with a discussion of how and why Python changes and described what the main results of that process are, including the differences between Python 2 and 3. We've learned how to reliably deal with those changes and learned some useful techniques that allow us to provide code that is compatible with various versions of Python and different versions of its libraries.

Then, we took a different look at the idea of changes in programming language. We've reviewed some of the popular alternative Python interpreters and discussed their main differences compared to default CPython implementation.

In the next chapter, we will describe modern ways of setting up repeatable and consistent development environments for Python programmers and discuss the two popular tools for environment isolation: virtualenv-type environments and Docker containers.

2
Modern Python Development Environments

A deep understanding of the programming language of choice is the most important thing in being an expert. This will always be true for any technology. Still, it is really hard to develop good software without knowing the common 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, in direct comparison of 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 spent years polishing standard practices and libraries that help to create more reliable software in a shorter time.

The most obvious and important part of the ecosystem is a huge collection of free and open source packages that solve a multitude of problems. Writing new software is always an expensive and time-consuming process. Being able to reuse the 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.

Because of this, Python developers put a lot of effort into creating tools and standards to work with open source packages that have been created by others—starting from virtual isolated environments, improved interactive shells, and debuggers, to programs that help to discover, search, and analyze the huge collection of packages that are available on **Python Package Index** (**PyPI**).

In this chapter, we will cover the following topics:

- Installing additional Python packages using pip
- Isolating the runtime environment
- Python's venv
- System-level environment isolation
- Popular productivity tools

Technical requirements

You can download 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`

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

- `virtualenv`
- `ipython`
- `ipdb`
- `ptpython`
- `ptbdb`
- `bpython`
- `bpdb`

You can install these packages using the following command:

```
python3 -m pip install <package-name>
```

The code files for this chapter can be found at `https://github.com/PacktPublishing/Expert-Python-Programming-Third-Edition/tree/master/chapter2`.

Installing additional Python packages using pip

Nowadays, a lot of operating systems come with Python as a standard component. Most Linux distributions and UNIX-based systems, such as FreeBSD, NetBSD, OpenBSD, or 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 preinstalled system version of Python is often Python 2.7, which is fairly outdated.

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 very limited, and they are mostly outdated compared to PyPI. This is the reason why `pip` should always be used to obtain new packages in the latest version, as recommended by the **Python Packaging Authority** (**PyPA**). Although it is an independent package, starting from version 2.7.9 and 3.4 of CPython, it is bundled with every new release by default. Installing the new package is as simple as this:

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

Among other features, `pip` allows specific versions of packages to be forced (using the `pip install package-name==version` syntax) and upgraded to the latest version available (using the `--upgrade` switch). The full usage description for most of the command-line tools presented in the book can be easily obtained simply by running the command with the `-h` or `--help` switch, but here is an example session that demonstrates the most commonly used options:

```
$ pip show pip
Name: pip
Version: 18.0
Summary: The PyPA recommended tool for installing Python packages.
Home-page: https://pip.pypa.io/
Author: The pip developers
Author-email: pypa-dev@groups.google.com
License: MIT
Location: /Users/swistakm/.envs/epp-3rd-ed/lib/python3.7/site-packages
Requires:
Required-by:

$ pip install 'pip>=18.0'
Requirement already satisfied: pip>=18.0 in (...)/lib/python3.7/site-
packages (18.0)

$ pip install --upgrade pip
Requirement already up-to-date: pip in (...)/lib/python3.7/site-packages
(18.0)
```

In some cases, `pip` may not be available by default. From Python 3.4 onward (and also Python 2.7.9), it can always be bootstrapped using the `ensurepip` module:

```
$ python -m ensurepip
Looking in links:
/var/folders/z6/3m2r6jgd04q0m7yq29c6lbzh0000gn/T/tmp784u9bct
Requirement already satisfied: setuptools in /Users/swistakm/.envs/epp-3rd-
```

```
ed/lib/python3.7/site-packages (40.4.3)
Collecting pip
Installing collected packages: pip
Successfully installed pip-10.0.1
```

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

`pip` may be used to install system-wide packages. On UNIX-based and Linux systems, this will require superuser privileges, so the actual invocation will be as follows:

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

Note that this is not required on Windows since it does not provide the Python interpreter by default, and Python on Windows is usually installed manually by the user without superuser privileges.

Installing system-wide packages directly from PyPI is not recommended, and should be avoided. This may seem like a contradiction to the previous statement that using `pip` is a PyPA recommendation, but there are some serious reasons for that. As we explained 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 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.

Doing such things on the local computer for development purposes only is also not a good excuse. Recklessly using `pip` that way is almost always asking for trouble, and will eventually lead to issues that are very hard to debug. This does not mean that installing packages from PyPI is a strictly forbidden thing, but it should be always done consciously and with an understanding of the related risk.

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 version 1.x but, Project Y needs 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.
- 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 *frozen*, 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 is impossible to maintain their dependencies without any 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 virtual environments. These focus on isolating the Python interpreter and the packages available within it. Such environments are very easy to set up, and are very often just enough to ensure proper isolation during the development of small projects and packages.

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, it is still very easy to run into issues that occur only on selected systems or even specific distributions of such systems (for example, Ubuntu versus Gentoo). This is very common in large and complex projects, especially if they depend on compiled Python extensions or internal components of the hosting 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 its libraries and crucial system components, either with classical system virtualization tools (for example, VMWare, Parallels, and VirtualBox) or container systems (for example, Docker and Rocket). Some of the available solutions that provide that kind of isolation are explained later in this chapter.

Let's take a look at Python's `venv` in the next section.

Python's venv

There are several ways to isolate Python at runtime. The simplest and most obvious, although hardest to maintain, is to manually change the `PATH` and `PYTHONPATH` environment variables and/or move the Python binary to a different, customized place where we want to store our project's dependencies, in order to affect the way that it discovers available packages. Fortunately, there are several tools available that can help in maintaining the virtual environments and packages that are installed for these environments. These are mainly `virtualenv` and `venv`. What they do under the hood is, in fact, the same that we would do manually. The actual strategy depends on the specific tool implementation, but generally they are more convenient to use and can provide additional benefits.

To create new virtual environment, you can simply use the following command:

```
python3.7 -m venv ENV
```

Here, `ENV` should be replaced by the desired name for the new environment. This will create a new `ENV` directory in the current working directory path. Inside, it will contain a few new directories:

- `bin/`: This is where the new Python executable and scripts/executables provided by other packages are stored.
- `lib/` and `include/`: These directories contain the supporting library files for new Python inside the virtual environment. The new packages will be installed in `ENV/lib/pythonX.Y/site-packages/`.

Once the new environment has been created, it needs to be activated in the current shell session using UNIX's source command:

```
source ENV/bin/activate
```

This changes the state of the current shell sessions 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)` string at its beginning. To illustrate this, here is an example session that creates a new environment and activates it:

```
$ python -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 depends completely on its state, as stored on a filesystem. It does not provide any additional abilities to track what packages should be installed in it. These virtual environments are also not portable, and should not be moved to another system/machine. This means that the new virtual environment needs to be created from scratch for each new application deployment. Because of this, there is a good practice that's used by `venv` users to store all project dependencies in the `requirements.txt` file (this is the naming convention), as shown in the following code:

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

# strict version names are best for reproducibility
eventlet==0.17.4
graceful==0.1.1

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

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

With such files, all dependencies can be easily installed using `pip`, because it accepts the requirements file as its output:

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

What needs to be remembered is that the requirements file is not always the ideal solution, because it does not define the exact list of dependencies, only those that are to be installed. So, the whole project can work without problems in some development environments but will fail to start in others if the requirements file is outdated and does not reflect the actual state of the environment. There is, of course, the `pip freeze` command, which prints all packages in the current environment, but it should not be used blindly. It will output everything, even packages that are not used in the project but are installed only for testing.

> **Note for Windows users**
> For Windows users, `venv` 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 better match development conventions on that operating system. The commands that are used for activating/deactivating the environment are also different; you need to use `ENV/Scripts/activate.bat` and `ENV/Scripts/deactivate.bat` instead of using `source` on `activate` and `deactivate` scripts.

> **Deprecated pyvenv script**
> The Python `venv` module provides an additional `pyvenv` command-line script; since Python 3.6, it has been marked as deprecated and its usage is officially discouraged, as the `pythonX.Y -m venv` command is explicit about what version of Python will be used to create new environments, unlike the `pyvenv` script.

The differences between `venv` versus `virtualenv` are discussed in the next section.

venv versus virtualenv

`virtualenv` was one of the most popular tools to create lightweight virtual environments long before the creation of the `pyenv` standard library module. Its name simply stands for virtual environment. It's not a part of the standard Python distribution, so it needs to be obtained using `pip`. If you want to use it, it is one of the packages that is worth installing system-wide (using `sudo` on Linux and UNIX-based systems).

I would recommend using the `venv` module instead of `virtualenv` whenever it is possible. This should be your default choice for projects targeting Python versions 3.4 and higher. Using `venv` in Python 3.3 may be a little inconvenient due to the lack of built-in support for `setuptools` and `pip`. For projects targeting a wider spectrum of Python runtimes (including alternative interpreters and 2.x branch), it seems that `virtualenv` is the best choice.

In the next section, we'll take a look at system-level environment isolation.

System-level environment isolation

In most cases, software implementation can iterate quickly because developers reuse a lot of existing components. *Don't Repeat Yourself* – this is a popular rule and motto of many programmers. Using other packages and modules to include them in the code base is only a part of that culture. What can also be considered under *reused components* are binary libraries, databases, system services, third-party APIs, and so on. Even whole operating systems should be considered as 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 (starting from the lowest):

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

Of course, such stacks can be even simpler, but it is very unlikely. In fact, big applications are often so complex that it is hard to distinguish single layers. 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 for complexity, and it seems that code simply follows the second law of thermodynamics.

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 RDBMS, such as PostgreSQL, they are usually available in different versions on different systems. 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 the production environment. Forcing a programmer to work on something else rather than their beloved system of choice is impossible.

The problem lies in the fact that portability is still a big challenge. Not all services will work exactly the same in production environments as they do on the developer's machines, and this is very unlikely to change. Even Python can behave differently on different systems, despite how much work is put in to make it cross-platform. Usually, this is well-documented and happens only in places that depend directly on system calls, but 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 an application environment. This is usually achieved by leveraging different types of system virtualization tools. Virtualization, of course, reduces performance; but with modern computers that have hardware support for virtualization, the performance loss is usually negligible. On the other hand, the list of possible gains is very long:

- The development environment can exactly match the system version and services 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 the development environment
- 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, **File System in User Space** (**FUSE**), which is not available in Windows

In the next section, we'll take a look at virtual development environments using Vagrant.

Virtual development environments using Vagrant

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, Mac OS, 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 `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 VM and VM's host
- Ports that need to be forwarded between VM and its host

The syntax language for `Vagrantfile` is Ruby. The example configuration file provides a good template to start the project and has an excellent documentation, so the knowledge of this language is not required. 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 VM using the default provider and base box image. 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 `Vagrantfile` for the Python 3.7 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
# -*- mode: ruby -*-
# vi: set ft=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|
    # Display the VirtualBox GUI when booting the machine
    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 python3.7 -y
  SHELL
end
```

In the preceding example, we have set an additional provision of system packages with simple shell script. When you feel that `Vagrantfile` is ready, you can run your virtual machine using the following command:

```
vagrant up
```

The initial start 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 VM is brought up, and the amount of time depends on the choice of provider, image, and your system's performance. Usually, this takes only a couple of seconds. Once the new Vagrant environment is up and running, developers can connect to it through SSH using the following shorthand:

```
vagrant ssh
```

This can be done anywhere in the project source tree below the location of `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 VM 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 `Vagrantfile`) is available on the VM's filesystem under `/vagrant/`. This directory is automatically synchronized with your host filesystem, so you can normally work in the IDE or editor of your choice run on the host, and can treat the SSH session to your Vagrant VM just like a normal local Terminal session.

Let's take a look at virtual environments using Docker in the next section.

Virtual environments using Docker

Containers are an 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. OS is shared between containers and the host, so it theoretically requires less overhead than in full virtualization. Such a container contains only application code and its system-level dependencies, but, from the perspective of processes running inside, it looks like a completely isolated system environment.

Software containers got their popularity mostly thanks to Docker, which is one of the available implementations. Docker allows to describe its container in the form of a simple text document called `Dockerfile`. Containers from such definitions can be built and stored. It also supports incremental changes, so if new things are added to the container then it does not need to be recreated from scratch.

Let's compare containerization and virtualization in the next section

Containerization versus virtualization

Different tools, such as Docker and Vagrant, seem to overlap in features – but the main difference between them is the reason why these tools were built. Vagrant, as we mentioned earlier, is built primarily as a tool for development. It allows us to bootstrap the whole virtual machine with a single command, but does not allow us to simply pack such an environment as a complete deliverable artifact and deploy or release it. Docker, on the other hand, is built exactly for that purpose – preparing complete containers that can be sent and deployed to production as a whole package. If implemented well, this can greatly improve the process of product deployment. Because of that, using Docker and similar solutions (Rocket for example) during development only makes more sense if such containers are also to be used in the deployment process on production.

Due to some implementation nuances, the environments that are based on containers may sometimes behave differently than environments based on virtual machines. 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 rather than 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.

Let's write our first Dockerfile in the next section.

Writing your first Dockerfile

Every Docker-based environment starts with Dockerfile. Dockerfile is a format 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.
- `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 specified commands on top of previous layers, and 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. If no entry point is specified anywhere in the image layers, then Docker defaults to `/bin/sh -c`.
- `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>", ...]`, although 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 Dockerfile, let's assume that we want to *dockerize* the built-in Python web server available through the `http.server` module with some predefined static files that this server should serve. The structure of our project files could be as follows:

```
.
├── Dockerfile
├── README
└── static
    ├── index.html
    └── picture.jpg
```

Locally, you could run that Python's `http.server` on a default HTTP port with the following simple command:

```
python3.7 -m http.server --directory static/ 80
```

This example is of course, very trivial, and using Docker for it is using a sledgehammer to crack a nut. So, just for the purpose of this example, let's pretend that we have a lot of code in the project that generates these static files. We would like to deliver only these static files, and not the code that generates them. Let's also assume that the recipients of our image know how to use Docker but don't know how to use Python.

So, what we want to achieve is the following:

- Hide some complexity from the user—especially the fact that we use Python and the HTTP server that's built-in into Python
- Package Python3.7 executable with all its dependencies and all static files primarily available in our project directory
- Provide some defaults to run the server on port `80`

With all these requirements, our Dockerfile could take the following form:

```
# Let's define base image.
# "python" is official Python image.
# The "slim" versions are sensible starting
# points for other lightweight Python-based images
FROM python:3.7-slim

# In order to keep image clean let's switch
# to selected working directory. "/app/" is
# commonly used for that purpose.
WORKDIR /app/

# These are our static files copied from
# project source tree to the current working
# directory.
COPY static/ static/

# We would run "python -m http.server" locally
# so lets make it an entry point.
ENTRYPOINT ["python3.7", "-m", "http.server"]

# We want to serve files from static/ directory
# on port 80 by default so set this as default arguments
# of the built-in Python HTTP server
CMD ["--directory", "static/", "80"]
```

Let's take a look at how to run containers in the next section.

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 we presented in the previous section, and we want to tag our image with the name `webserver`. The `docker build` command invocation will be following, and its output may be as follows:

```
$ docker build -t webserver .
Sending build context to Docker daemon 4.608kB
Step 1/5 : FROM python:3.7-slim
3.7-slim: Pulling from library/python
802b00ed6f79: Pull complete
cf9573ca9503: Pull complete
b2182f7db2fb: Pull complete
37c0dde21a8c: Pull complete
a6c85c69b6b4: Pull complete
Digest:
sha256:b73537137f740733ef0af985d5d7e5ac5054aadebfa2b6691df5efa793f9fd6d
Status: Downloaded newer image for python:3.7-slim
 ---> a3aec6c4b7c4
Step 2/5 : WORKDIR /app/
 ---> Running in 648a5bb2d9ab
Removing intermediate container 648a5bb2d9ab
 ---> a2489d084377
Step 3/5 : COPY static/ static/
 ---> 958a04fa5fa8
Step 4/5 : ENTRYPOINT ["python3.7", "-m", "http.server", "--bind", "80"]
 ---> Running in ec9f2a63c472
Removing intermediate container ec9f2a63c472
 ---> 991f46cf010a
Step 5/5 : CMD ["--directory", "static/"]
 ---> Running in 60322d5a9e9e
Removing intermediate container 60322d5a9e9e
 ---> 40c606a39f7a
Successfully built 40c606a39f7a
Successfully tagged webserver:latest
```

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

```
$ docker images
REPOSITORY      TAG           IMAGE ID          CREATED           SIZE
webserver       latest        40c606a39f7a      2 minutes ago     143MB
python          3.7-slim      a3aec6c4b7c4      2 weeks ago       143MB
```

> **The shocking size of container images**
> The 143 MB of image for a simple Python image may seem 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.

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 -p 80:80 webserver
```

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

- `-it`: These are actually two concatenated options: `-i` and `-t`. `-i` (like *interactive*) keeps STDIN open, even if the container process is detached, and `-t` (like `tty`) allocates pseudo-TTY for the container. In short, thanks to these two options, we will be able to see live logs from `http.server` 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 container when it exits.

- `-p 80:80`: Tells Docker to publish the container's `port 80` by binding `port 80` on the host's interface.

Setting up complex environments

While the basic usage of Docker is pretty straightforward for basic setups, it can be 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 be really complicated when you have one service that needs to communicate with others. Single docker containers should only contain one running process.

This means that you really shouldn't put any additional process supervision tools, such as Supervisor and Circus, and should 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.

The best tool that you can use in both simple and complex scenarios is Compose. Compose is usually distributed with Docker, but in some Linux distributions (for example, Ubuntu), it may not be available by default, and must be installed as a separate package from the packages repository. 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 directory. An example of such a file for our previous project could be as follows:

```yaml
version: '3'

services:
  webserver:
    # this tell Compose to build image from
    # local (.) directory
    build: .
    # this is equivalent to "-p" option of
    # the "docker build" command
    ports:
    - "80:80"

    # this is equivalent to "-t" option of
    # the "docker build" 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`
- `docker-compose down`

Useful Docker 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 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 thoughtfully in order to use it efficiently and overcome some of the problems that are inevitable.

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 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:

1. **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 only 5 MB in size, and provides an elegant package manager that allows you to keep your images compact, too.

2. **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:

```
# Here we use bare alpine to illustrate
# package management as it lacks Python
# by default. For Python projects in general
# the 'python:3.7-alpine' is probably better
# choice.
FROM alpine:3.7

# Add python3 package as alpine image lacks it by default
RUN apk add python3

# Run multiple commands in single RUN instruction
# so space can be reclaimed after the 'apk del py3-pip'
# command because image layer is committed only after
# whole whole instruction.
```

```
RUN apk add py3-pip && \
    pip3 install django && \
    apk del py3-pip

# (...)
```

Addressing services inside of a 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:

```
version: '3'

services:
  webserver:
    build: .
    ports:
    - "80:80"
    tty: true

  database:
    image: postgres
    restart: always
```

The preceding configuration defines two services:

- `webserver`: This is a main application service container with images built from the local Dockerfile
- `database`: This is a PostgreSQL database container from an official `postgres` Docker image

We assume that the `webserver` 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, Compose is a tool that was designed exactly for such scenarios, so it will make it a lot more easier for us.

Whenever you start your environment with the `docker-compose up` command, 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 `webserver` service can use the `database:5432` address to communicate with its database (5432 is the default PostgreSQL port), and any other service in that Compose applicant will be able to access the HTTP endpoint of the `webserver` service under the `http://webserver:80` address.

Even though the service hostnames in Compose are easily predictable, it isn't good practice to hardcode any addresses in your application or its configuration. The best approach would be to provide them as environment variables that can be read by an 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'

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

  database:
    image: postgres
    restart: always
```

Communicating between multiple 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 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`:

```
version: '3'

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

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

  database:
    image: postgres
    restart: always
```

Such external networks are not managed by 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:

```
version: '3'

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

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

Let's take a look at popular productivity tools in the next section.

Popular productivity tools

Productivity tool is bit of a vague term. On one hand, almost every open source code package that has been released and is available online is a kind of productivity booster – it provides ready-to-use solutions to some problem, so that no one needs to spend time on it (ideally speaking). On the other hand, you could say that the whole of Python is about productivity—and both are undoubtedly true. Almost everything in this language and community surrounding it seems to be designed in order to make software development as productive as possible.

This creates a positive feedback loop. Since writing code is fun and easy, a lot of programmers use their free time to create tools that make it even easier and 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 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.

The most important productivity tools, `pip` and `venv`, were already discussed earlier in this chapter. Some of them have packages for specific problems, such as profiling and testing, which have their own chapters in this book. 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 – ipython, bpython, ptpython, and so on

Python programmers spend a lot of time in interactive interpreter sessions. It is 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 it 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, macOS) may be already preconfigured to provide a default startup script. It is commonly found in the users, 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.

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.join(os.environ['HOME'],
'.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.

Setting up the PYTHONSTARTUP environment variable

If you are running Linux or macOS, the simplest way is to create the 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, Bash and Korn shell 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 only a few can find time for. Fortunately, there are a few custom Python shell implementations that immensely improve the experience of interactive sessions in Python.

IPython

IPython (`https://ipython.readthedocs.io/en/stable/overview.html`) provides an extended Python command shell. Among the features that it provides, the most interesting ones are as follows:

- Dynamic object introspection
- System shell access from the prompt
- Profiling direct support
- Debugging facilities

Now, IPython is a part of the larger project called Jupyter, which provides interactive notebooks with live code that can be written in many different languages.

bpython

bpython (`https://bpython-interpreter.org/`) advertises itself as a fancy interface to the Python interpreter. Here are some of the accented features on the projects page:

- In-line syntax highlighting
- Readline-like autocomplete with suggestions displayed as you type

- Expected parameter list for any Python function
- Auto-indentation
- Python 3 support

ptpython

ptpython (`https://github.com/jonathanslenders/ptpython/`) is another approach to the topic of advanced Python shells. What is interesting about this project is that core prompt utilities implementation is available as a separate package, called `prompt_toolkit` (from the same author). This allows us to easily create various aesthetically pleasing interactive command-line interfaces.

It is often compared to bpython in functionalities, but the main difference is that it enables compatibility mode with IPython and its syntax enables additional features, such as `%pdb`, `%cpaste`, or `%profile`.

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 for easier experimentation with your code and inspection of its internal state. The simplest module that allows for 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 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 more fancy 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, all of the interactive shells that were 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:

```
# 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())
```

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` statements 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` (refer to `https://docs.python.org/3/library/pdb.html`). It can be invoked from the command line on the existing script, so Python will enter post-mortem debugging if the program exits abnormally:

```
python -m pdb script.py
```

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 just behaves abnormally, but does not exit unexpectedly. In such cases, custom breakpoints can be set on a specific line of code using this single-line idiom:

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

This will cause the Python interpreter to start the debugger session on this line during runtime.

pdb is very useful for tracing issues, and at first glance it may look very familiar 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 (bdb), 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 and additional information.

The debugger session in pdb is also very simple and does not provide additional features such as tab completion or code highlighting. Fortunately, there are a few packages available on PyPI that provide such features from alternative Python shells, as mentioned in the previous section. The most notable examples are as follows:

- ipdb: This is a separate package based on ipython
- ptpdb: This is a separate package based on ptpython
- bpdb: This is bundled with bpython

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. 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.

Now that you all of these tools under your tool belt, you are well-prepared for the next few chapters, where we will discuss multiple features of modern Python syntax.

In the next chapter, we will focus on the best practices for writing code in Python (language idioms) and provide a summary of selected elements of Python syntax that may be new for intermediate Python users, or familiar for those experienced with older versions of Python.

We will also take a look at internal CPython type implementations and their computational complexities as a rationale for provided idioms.

2
Section 2: Python Craftsmanship

This section provides an overview of the current landscape of Python development, from the perspective of the software craftsman – someone who programs for a living and must know their tools and language inside-out. The reader will learn what the newest exciting elements of Python syntax are and how to reliably and consistently deliver quality software.

The following chapters are included in this section:

- `Chapter 3`, *Modern Syntax Elements – Below the Class Level*
- `Chapter 4`, *Modern Syntax Elements – Above the Class Level*
- `Chapter 5`, *Elements of Metaprogramming*
- `Chapter 6`, *Choosing Good Names*
- `Chapter 7`, *Writing a Package*
- `Chapter 8`, *Deploying the Code*
- `Chapter 9`, *Python Extensions in Other Languages*

3
Modern Syntax Elements - Below the Class Level

Python has evolved a lot in the last few years. From the earliest version to the current one (3.7 at this time), a lot of enhancements have been made to make the language clearer, cleaner, and easier to write. Python basics have not changed drastically, but the tools it provides are now a lot more ergonomic.

As Python evolves, your software should too. Taking great care over how your program is written weighs heavily on how it will evolve. Many programs end up being ditched and rewritten from scratch because of their obtuse syntax, unclear APIs, or unconventional standards. Using new features of a programming language that make your code more expressive and readable increases the maintainability of your software and so prolongs its lifetime.

This chapter presents the most important elements of modern Python syntax, and tips on their usage. We will also discuss some implementation details of the built-in Python types that have various implications on your code performance, but we won't be digging too much into optimization techniques. The code performance tips for speed improvement or memory usage will be covered later in `Chapter 13`, *Optimization – Principles and Profiling Techniques*, and `Chapter 14`, *Optimization – Some Powerful Techniques*.

In this chapter, we will cover the following topics:

- Python's built-in types
- Supplemental data types and containers
- Advanced syntax
- Functional-style features of Python
- Function and variable annotations
- Other syntax elements you may not know yet

Technical requirements

The code files for this chapter can be found at `https://github.com/PacktPublishing/ Expert-Python-Programming-Third-Edition/tree/master/chapter3`.

Python's built-in types

Python provides a great set of data types. This is true for both numeric types and also collections. Regarding the numeric types, there is nothing special about their syntax. There are, of course, some differences for defining literals of every type and some (maybe) not well-known details regarding operators, but there isn't a lot that could surprise you in Python regarding the syntax for numeric types. Things change when it comes to collections and strings. Despite the *there should be only one way to do something* mantra, the Python developer is really left with plenty of choices. Some of the code patterns that seem intuitive and simple to beginners are often considered non-Pythonic by experienced programmers, because they are either inefficient or simply too verbose.

Such Pythonic patterns for solving common problems (many programmers call these idioms) may often seem like only aesthetics. You couldn't be more wrong in thinking that. Most of the idioms are driven by the fact that Python is implemented internally and how the built-in structures and modules work. Knowing more about such details is essential for a good understanding of the language. Unfortunately, the community itself is not free from myths and stereotypes about how things in Python work. Only by digging deeper by yourself will you be able to tell which of the popular statements about Python are really true.

Let's look at strings and bytes.

Strings and bytes

The topic of strings may provide some confusion for programmers that used to program only in Python 2. In Python 3, there is only one datatype capable of storing textual information. It is `str`, or simply string. It is an immutable sequence that stores Unicode code points. This is the major difference from Python 2, where `str` represented byte strings – something that is now handled by the `bytes` objects (but not exactly in the same way).

Strings in Python are sequences. This single fact should be enough to include them in a section covering other container types. But they differ from other container types in one important detail. Strings have very specific limitations on what type of data they can store, and that is Unicode text.

`bytes`, and its mutable alternative, `bytearray`, differs from `str` by allowing only bytes as a sequence value, and bytes in Python are integers in the `0 <= x < 256` range. This may be a bit confusing at the beginning, because, when printed, they may look very similar to strings:

```
>>> print(bytes([102, 111, 111]))
b'foo'
```

The `bytes` and `bytearray` types allow you to work with raw binary data that may not always have to be textual (for example, audio/video files, images, and network packets). The true nature of these types is revealed when they are converted into other sequence types, such as `list` or `tuple`:

```
>>> list(b'foo bar')
[102, 111, 111, 32, 98, 97, 114]
>>> tuple(b'foo bar')
(102, 111, 111, 32, 98, 97, 114)
```

A lot of Python 3 controversy was about breaking the backwards compatibility for string literals and how Python deals with Unicode. Starting from Python 3.0, every string literal without any prefix is Unicode. So, literals enclosed by single quotes (`'`), double quotes (`"`), or groups of three quotes (single or double) without any prefix represent the `str` data type:

```
>>> type("some string")
<class 'str'>
```

In Python 2, the Unicode literals required a `u` prefix (like `u"some string"`). This prefix is still allowed for backwards compatibility (starting from Python 3.3), but does not hold any syntactic meaning in Python 3.

Byte literals were already presented in some of the previous examples, but let's explicitly present their syntax for the sake of consistency. Bytes literals are enclosed by single quotes, double quotes, or triple quotes, but must be preceded with a `b` or `B` prefix:

```
>>> type(b"some bytes")
<class 'bytes'>
```

Note that Python does not provide a syntax for `bytearray` literals. If you want to create a `bytearray` value, you need to use a `bytes` literal and a `bytearray()` type constructor:

```
>>> bytearray(b'some bytes')
bytearray(b'some bytes')
```

It is important to remember that Unicode strings contain **abstract** text that is independent from the byte representation. This makes them unable to be saved on the disk or sent over the network without encoding them to binary data. There are two ways to encode string objects into byte sequences:

- Using the `str.encode(encoding, errors)` method, which encodes the string using a registered codec for encoding. Codec is specified using the `encoding` argument, and, by default, it is `'utf-8'`. The second errors, argument specifies the error handling scheme. It can be `'strict'` (default), `'ignore'` , `'replace'` , `'xmlcharrefreplace'`, or any other registered handler (refer to the built-in `codecs` module documentation).
- Using the `bytes(source, encoding, errors)` constructor, which creates a new bytes sequence. When the source is of the `str` type, then the `encoding` argument is obligatory and it does not have a default value. The usage of the `encoding` and `errors` arguments is the same as for the `str.encode()` method.

Binary data represented by `bytes` can be converted into a string in an analogous way:

- Using the `bytes.decode(encoding, errors)` method, which decodes the bytes using the codec registered for encoding. The arguments of this method have the same meaning and defaults as the arguments of `str.encode()`.
- Using the `str(source, encoding, error)` constructor, which creates a new string instance. Similar to the `bytes()` constructor, the `encoding` argument in the `str()` call has no default value and must be provided if the bytes sequence is used as a source.

> **Naming – bytes versus byte string**
> Due to changes made in Python 3, some people tend to refer to
> the `bytes` instances as byte strings. This is mostly due to historic reasons
> – `bytes` in Python 3 is the sequence type that is the closest one to
> the `str` type from Python 2 (but not the same). Still, the `bytes` instance is
> a sequence of bytes and also does not need to represent textual data. So, in
> order to avoid any confusion, it is advised to always refer to them as
> either `bytes` or byte sequence, despite their similarities to strings. The
> concept of strings is reserved for textual data in Python 3, and this is now
> always `str`.

Let's look into the implementation details of strings and bytes.

Implementation details

Python strings are immutable. This is also true for byte sequences. This is an important fact, because it has both advantages and disadvantages. It also affects the way strings should be handled in Python efficiently. Thanks to immutability, strings can be used as dictionary keys or `set` collection elements because, once initialized, they will never change their value. On the other hand, whenever a modified string is required (even with only tiny modification), a completely new instance needs to be created. Fortunately, `bytearray`, as a mutable version of `bytes`, does not have such an issue. Byte arrays can be modified in-place (without creating new objects) through item assignments and can be dynamically resized, exactly like lists – using appends, pops, inserts, and so on.

Let's discuss string concatenation in the next section.

String concatenation

The fact that Python strings are immutable imposes some problems when multiple string instances need to be joined together. As we stated previously, concatenating immutable sequences results in the creation of a new sequence object. Consider that a new string is built by repeated concatenation of multiple strings, as follows:

```
substrings = ["These ", "are ", "strings ", "to ", "concatenate."]
s = ""
for substring in substrings:
    s += substring
```

This will result in quadratic runtime costs in the total string length. In other words, it is highly inefficient. For handling such situations, the `str.join()` method is available. It accepts iterables of strings as the argument and returns joined strings. The call to `join()` of the `str` type can be done in two forms:

```
# using empty literal
s = "".join(substrings)

# using "unbound" method call
str.join("", substrings)
```

The first form of the `join()` call is the most common idiom. The string that provides this method will be used as a separator between concatenated substrings. Consider the following example:

```
>>> ','.join(['some', 'comma', 'separated', 'values'])
'some,comma,separated,values'
```

It is worth remembering that just because it is faster (especially for large lists), it does not mean that the `join()` method should be used in every situation where two strings need to be concatenated. Despite being a widely recognized idiom, it does not improve code readability. And readability counts! There are also some situations where `join()` may not perform as well as ordinary concatenation with a + operator. Here are some examples:

- If the number of substrings is very small and they are not contained already by some iterable variable (existing list or tuple of strings) – in some cases the overhead of creating a new sequence just to perform concatenation can overshadow the gain of using `join()`.
- When concatenating short literals – thanks to some interpreter-level optimizations, such as **constant folding** in CPython (see the following subsection), some complex literals (not only strings), such as `'a'` + `'b'` + `'c'`, can be translated into a shorter form at compile time (here `'abc'`). Of course, this is enabled only for constants (literals) that are relatively short.

Ultimately, if the number of strings to concatenate is known beforehand, the best readability is ensured by proper string formatting either using the `str.format()` method, the `%` operator, or f-string formatting. In code sections where the performance is not critical or the gain from optimizing string concatenation is very little, string formatting is recommended as the best alternative to concatenation.

Constant folding, the peephole optimizer, and the AST optimizer

CPython uses various techniques to optimize your code. The first optimization takes place as soon as source code is transformed into the form of the abstract syntax tree, just before it is compiled into byte code. CPython can recognize specific patterns in the abstract syntax tree and make direct modifications to it. The other kind of optimizations are handled by the peephole optimizer. It implements a number of common optimizations directly on Python's byte code. As we mentioned earlier, constant folding is one such feature. It allows the interpreter to convert complex literal expressions (such as `"one" + " " + "thing"`,`" "` `* 79`, or `60 * 1000`) into a single literal that does not require any additional operations (concatenation or multiplication) at runtime.

Until Python 3.5, all constant folding was done in CPython only by the peephole optimizer. For strings, the resulting constants were limited in length by a hardcoded value. In Python 3.5, this value was equal to 20. In Python 3.7, most of the constant folding optimizations are handled earlier on the abstract syntax tree level. These particular details are a curiosity rather than a thing that can be relied on in your day-to-day programming. Information about other interesting optimizations performed by AST and peephole optimizers can be found in the `Python/ast_opt.c` and `Python/peephole.c` files of Python's source code.

Let's take a look at string formatting with f-strings.

String formatting with f-strings

F-strings are one of the most beloved new Python features that came with Python 3.6. It's also one of the most controversial features of that release. The f-strings or **formatted string literals** that were introduced by the PEP 498 document add a new way to format strings in Python. Before Python 3.6, there were two basic ways to format strings:

- Using % formatting for example `"Some string with included % value" % "other"`
- Using the `str.format()` method for example `"Some string with included {other} value".format(other="other")`

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 the text that has to be formatted. In the `str.format()` method, the text substitutions refer to arguments and keyword arguments that are passed to the formatting method. You can use either anonymous substitutions that will translate to consecutive argument indexes, explicit argument indexes, or keyword names.

This means that the same string can be formatted in different ways:

```
>>> from sys import version_info
>>> "This is Python {}.{}".format(*version_info)
'This is Python 3.7'

>>> "This is Python {0}.{1}".format(*version_info)
'This is Python 3.7'

>>> "This is Python {major}.{minor}".format(major=version_info.major,
minor=version_info.minor)
'This is Python 3.7'
```

What makes f-strings special is that replacement fields can be any Python expression, and it will be evaluated at runtime. Inside of strings, you have access to any variable that is available in the same namespace as the formatted literal. With f-strings, the preceding examples could be written in the following way:

```
>>> from sys import version_info
>>> f"This is Python {version_info.major}.{version_info.minor}"
'This is Python 3.7'
```

The ability to use expressions as replacement fields make 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 that prints the first ten powers of the number 10 using f-strings and aligns 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 is almost like a separate mini-language inside Python. The best reference for it is the official documentation which you can find under `https://docs.python.org/3/library/string.html`. Another useful internet resource for that topic is `https://pyformat.info/`, which presents the most important elements of this specification using practical examples.

Let's take a look at containers in the next section.

Containers

Python provides a good selection of built-in data containers that allow you to efficiently solve many problems if you choose them wisely. Types that you should already know of are those that have dedicated literals:

- Lists
- Tuples
- Dictionaries
- Sets

Python is, of course, not limited to these four containers, and it extends the list of possible choices through its standard library. In many cases, solutions to some problems may be as simple as making a good choice for the data structure to hold your data. This part of the book aims to ease such decisions by providing deeper insight of the possible options.

Lists and tuples

Two of the most basic collection types in Python are lists and tuples, and they both represent sequences of objects. The basic difference between them should be obvious for anyone who has spent more than a few hours with Python; lists are dynamic, so they can change their size, while tuples are immutable (cannot be modified after they are created).

Lists and tuples in Python have various optimizations that make allocations/deallocations of small objects fast. They are also the recommended datatypes for structures where the position of the element is information by itself. For example, a tuple may be a good choice for storing a pair of (*x*, *y*) coordinates. Implementation details regarding tuples are not interesting. The only important thing about them in the scope of this chapter is that `tuple` is **immutable** and thus **hashable**. A detailed explanation of this section will be covered later in the *Dictionaries* section. More interesting than tuples is its dynamic counterpart – lists. In the next section, we will discuss how it really works, and how to deal with it efficiently.

Implementation details

Many programmers easily confuse Python's `list` type with the concept of linked lists which are found often in standard libraries of other languages, such as C, C++, or Java. In fact, CPython lists are not lists at all. In CPython, lists are implemented as variable length arrays. This should be also true for other implementations, such as Jython and IronPython, although such implementation details are often not documented in these projects. The reasons for such confusion is clear. This datatype is named **list** and also has an interface that could be expected from any linked list implementation.

Why it is important and what does it mean? Lists are one of the most popular data structures, and the way in which they are used greatly affects every application's performance. CPython is the most popular and used implementation, so knowing its internal implementation details is crucial.

Lists in Python are contiguous arrays of references to other objects. The pointer to this array and the length is stored in the list's head structure. This means that every time an item is added or removed, the array of references needs to be resized (reallocated). Fortunately, in Python, these arrays are created with exponential over allocation, so not every operation requires an actual resize of the underlying array. This is how the amortized cost of appending and popping elements can be low in terms of complexity. Unfortunately, some other operations that are considered *cheap* in ordinary linked lists have relatively high computational complexity in Python:

- Inserting an item at an arbitrary place using the `list.insert` method has complexity $O(n)$
- Deleting an item using `list.delete` or using the `del` operator `har` has complexity $O(n)$

At least retrieving or setting an element using an index is an operation where cost is independent of the list's size, and the complexity of these operations is always $O(1)$.

Let's define n as the length of a list. Here is a full table of average time complexities for most of the list operations:

Operation	Complexity
Copy	$O(n)$
Append	$O(1)$
Insert	$O(n)$
Get item	$O(1)$
Set item	$O(1)$

Delete item	O(*n*)
Iteration	O(*n*)
Get slice of length *k*	O(*k*)
Del slice	O(*n*)
Set slice of length *k*	O(*k+n*)
Extend	O(*k*)
Multiply by *k*	O(*nk*)
Test existence (`element in list`)	O(*n*)
`min()`/`max()`	O(*n*)
Get length	O(*1*)

For situations where a real linked list or doubly linked list is required, Python provides a `deque` type in the `collections` built-in module. This is a data structure that allows us to append and pop elements at each side with *O*(1) complexity. This is a generalization of stacks and queues, and should work fine anywhere where a doubly linked list is required.

List comprehensions

As you probably know, writing a piece of code such as this can be tedious:

```
>>> evens = []
>>> for i in range(10):
...     if i % 2 == 0:
...         evens.append(i)
...
>>> evens
[0, 2, 4, 6, 8]
```

This may work for C, but it actually makes things slower for Python for the following reasons:

- It makes the interpreter work on each loop to determine what part of the sequence has to be changed
- It makes you keep a counter to track what element has to be processed
- It requires additional function lookups to be performed at every iteration because `append()` is a list's method

A list comprehension is a better pattern for these kind of situations. It allows us to define a list by using a single line of code:

```
>>> [i for i in range(10) if i % 2 == 0]
[0, 2, 4, 6, 8]
```

This form of writing is much shorter and involves fewer elements. In a bigger program, this means less bugs and code that is easier to understand. This is the reason why many experienced Python programmers will consider such forms as being more readable.

> **List comprehensions and internal array resize**
>
> There is a myth among some Python programmers that list comprehensions can be a workaround for the fact that the internal array representing the list object must be resized with every few additions. Some say that the array will be allocated once in just the right size. Unfortunately, this isn't true.
>
> The interpreter, during evaluation of the comprehension, can't know how big the resulting container will be, and it can't preallocate the final size of the array for it. Due to this, the internal array is reallocated in the same pattern as it would be in the `for` loop. Still, in many cases, list creation using comprehensions is both cleaner and faster than using ordinary loops.

Other idioms

Another typical example of a Python idiom is the use of `enumerate()`. This built-in function provides a convenient way to get an index when a sequence is iterated inside of a loop. Consider the following piece of code as an example of tracking the element index without the `enumerate()` function:

```
>>> i = 0
>>> for element in ['one', 'two', 'three']:
...     print(i, element)
...     i += 1
...
0 one
1 two
2 three
```

This can be replaced with the following code, which is shorter and definitely cleaner:

```
>>> for i, element in enumerate(['one', 'two', 'three']):
...     print(i, element)
...
0 one
1 two
2 three
```

If you need to aggregate elements of multiple lists (or any other iterables) in the one-by-one fashion, you can use the built-in `zip()`. This is a very common pattern for uniform iteration over two same-sized iterables:

```
>>> for items in zip([1, 2, 3], [4, 5, 6]):
...     print(items)
...
(1, 4)
(2, 5)
(3, 6)
```

Note that the results of `zip()` can be reversed by another `zip()` call:

```
>>> for items in zip(*zip([1, 2, 3], [4, 5, 6])):
...     print(items)
...
(1, 2, 3)
(4, 5, 6)
```

One important thing you need to remember about the `zip()` function is that it expects input iterables to be the same size. If you provide arguments of different lengths, then it will trim the output to the shortest argument, as shown in the following example:

```
>>> for items in zip([1, 2, 3, 4], [1, 2]):
...     print(items)
...
(1, 1)
(2, 2)
```

Another popular syntax element is sequence unpacking. It is not limited to lists and tuples, and will work with any sequence type (even strings and byte sequences). It allows us to unpack a sequence of elements into another set of variables as long as there are as many variables on the left-hand side of the assignment operator as the number of elements in the sequence. If you paid attention to the code snippets, then you might have already noticed this idiom when we were discussing the `enumarate()` function.

The following is a dedicated example of that syntax element:

```
>>> first, second, third = "foo", "bar", 100
>>> first
'foo'
>>> second
'bar'
>>> third
100
```

Unpacking also allows us to capture multiple elements in a single variable using starred expressions as long as it can be interpreted unambiguously. Unpacking can also be performed on nested sequences. This can come in handy, especially when iterating on some complex data structures built out of multiple sequences. Here are some examples of more complex sequence unpacking:

```
>>> # starred expression to capture rest of the sequence
>>> first, second, *rest = 0, 1, 2, 3
>>> first
0
>>> second
1
>>> rest
[2, 3]
>>> # starred expression to capture middle of the sequence
>>> first, *inner, last = 0, 1, 2, 3
>>> first
0
>>> inner
[1, 2]
>>> last
3
>>> # nested unpacking
>>> (a, b), (c, d) = (1, 2), (3, 4)
>>> a, b, c, d
(1, 2, 3, 4)
```

Dictionaries

Dictionaries are one of most versatile data structures in Python. The `dict` type allows you to map a set of unique keys to values, as follows:

```
{
    1: ' one',
    2: ' two',
    3: ' three',
}
```

Dictionary literals are a very basic thing, and you should already know about them. Python allows programmers to also create a new dictionary using comprehensions, similar to the list comprehensions mentioned earlier. Here is a very simple example that maps numbers in a range from 0 to 99 to their squares:

```
squares = {number: number**2 for number in range(100)}
```

What is important is that the same benefits of using list comprehensions apply to dictionary comprehensions. So, in many cases, they are more efficient, shorter, and cleaner. For more complex code, when many `if` statements or function calls are required to create a dictionary, the simple `for` loop may be a better choice, especially if it improves readability.

For Python programmers new to Python 3, there is one important note about iterating over dictionary elements. The `keys()`, `values()`, and `items()` dictionary methods are no longer return lists. Also, their counterparts, `iterkeys()`, `itervalues()`, and `iteritems()`, which returned iterators instead, are missing in Python 3. Now, the `keys()`, `values()`, and `items()` methods return special view objects:

- `keys()`: This returns the `dict_keys` object which provides a view on all keys of the dictionary
- `values()`: This returns the `dict_values` object which provides a view on all values of the dictionary
- `items()`: This returns the `dict_items` object, providing views on all `(key, value)` two-tuples of the dictionary

View objects provide a view on the dictionary content in a dynamic way so that every time the dictionary changes, the views will reflect these changes, as shown in this example:

```
>>> person = {'name': 'John', 'last_name': 'Doe'}
>>> items = person.items()
>>> person['age'] = 42
>>> items
dict_items([('name', 'John'), ('last_name', 'Doe'), ('age', 42)])
```

View objects join the behavior of lists returned by the implementation of old methods with iterators that have been returned by their `iter` counterparts. Views do not need to redundantly store all values in memory (like lists do), but are still allowed to access their length (using the `len()` function) and testing for membership (using the `in` keyword). Views are, of course, iterable.

The last important thing about views is that both view objects returned by the `keys()` and `values()` methods ensure the same order of keys and values. In Python 2, you could not modify the dictionary content between these two calls if you wanted to ensure the same order of retrieved keys and values. `dict_keys` and `dict_values` are now dynamic, so even if the content of the dictionary changes between the `keys()` and `values()` calls, the order of iteration is consistent between these two views.

Implementation details

CPython uses hash tables with pseudo-random probing as an underlying data structure for dictionaries. It seems like a very deep implementation detail, but it is very unlikely to change in the near future, so it is also a very interesting fact for the Python programmer.

Due to this implementation detail, only objects that are **hashable** can be used as a dictionary key. An object is hashable if it has a hash value that never changes during its lifetime, and can be compared to different objects. Every Python built-in type that is immutable is also hashable. Mutable types, such as list, dictionaries, and sets, are not hashable, and so they cannot be used as dictionary keys. Protocol that defines if a type is hashable consists of two methods:

- `__hash__`: This provides the hash value (as an integer) that is needed by the internal `dict` implementation. For objects that are instances of user-defined classes, it is derived from their `id()`.
- `__eq__`: This compares if two objects have the same value. All objects that are instances of user-defined classes compare as unequal by default, except for themselves.

Two objects that are compared as equal must have the same hash value. The reverse does not need to be true. This means that collisions of hashes are possible – two objects with the same hash may not be equal. It is allowed, and every Python implementation must be able to resolve hash collisions CPython uses **open addressing** to resolve such collisions. The probability of collisions greatly affects dictionary performance, and, if it is high, the dictionary will not benefit from its internal optimizations.

While three basic operations, adding, getting, and deleting an item, have an average time complexity equal to $O(1)$, their amortized worst case complexities are a lot higher. It is $O(n)$, where n is the current dictionary size. Additionally, if user-defined class objects are used as dictionary keys and they are hashed improperly (with a high risk of collisions), this will have a huge negative impact on the dictionary's performance. The full table of CPython's time complexities for dictionaries is as follows:

Operation	Average complexity	Amortized worst case complexity
Get item	$O(1)$	$O(n)$
Set item	$O(1)$	$O(n)$
Delete item	$O(1)$	$O(n)$
Copy	$O(n)$	$O(n)$
Iteration	$O(n)$	$O(n)$

It is also important to know that the n number in worst case complexities for copying and iterating the dictionary is the maximum size that the dictionary ever achieved, rather than the size at the time of operation. In other words, iterating over the dictionary that once was huge but greatly shrunk in time may take a surprisingly long time. In some cases, it may be better to create a new dictionary object from a dictionary that needs to be shrunk if it has to be iterated often instead of just removing elements from it.

Weaknesses and alternatives

For a very long time, one of the most common pitfalls regarding dictionaries was expecting that they preserve the order of elements in which new keys were added. The situation has changed a bit in Python 3.6, and the problem was finally solved in Python 3.7 on the level of language specification.

But, before we dig deeper into the situation of Python 3.6 and later releases, we need to make a small detour and examine the problem as if we were still stuck in the past, when the only Python releases available were older than 3.6. In the past, you could have a situation where the consecutive dictionary keys also had hashes that were consecutive values too. And, for a very long time, this was the only situation when you could expect that you would iterate over dictionary elements in the same order as they were added to the dictionary. The easiest way to present this is by using integer numbers, as hashes of integer numbers are the same as their value:

```
>>> {number: None for number in range(5)}.keys()
dict_keys([0, 1, 2, 3, 4])
```

Using other datatypes that hash differently could show that the order is not preserved. Here is an example that was executed in CPython 3.5:

```
>>> {str(number): None for number in range(5)}.keys()
dict_keys(['1', '2', '4', '0', '3'])
>>> {str(number): None for number in reversed(range(5))}.keys()
dict_keys(['2', '3', '1', '4', '0'])
```

As shown in the preceding code, for CPython 3.5 (and also earlier versions), the resulting order is both dependent on the hashing of the object and also on the order in which the elements were added. This is definitely not what can be relied on, because it can vary with different Python implementations.

So, what about Python 3.6 and later releases? Starting from Python 3.6, the CPython interpreter uses a new compact dictionary representation that has a noticeably smaller memory footprint and also preserves order as a side effect of that new implementation. In Python 3.6, the order preserving nature of dictionaries was only an implementation detail, but in Python 3.7, it has been officially declared in the Python language specification. So, starting from Python 3.7, you can finally rely on the item insertion order of dictionaries.

In parallel to the CPython implementation of dictionaries, Python 3.6 introduced another change in the syntax that is related to the order of items in dictionaries. As defined in the PEP 486 "*Preserving the order of **kwargs in a function*" document, the order of keyword arguments collected using the `**kwargs` syntax must be the same as presented in function call. This behavior can be clearly presented with the following example:

```
>>> def fun(**kwargs):
...     print(kwargs)
...
>>> fun(a=1, b=2, c=3)
{'a': 1, 'b': 2, 'c': 3}
>>> fun(c=1, b=2, a=3)
{'c': 1, 'b': 2, 'a': 3}
```

However the preceding changes can be used effectively only in the newest releases of Python. So, what should you do if you have a library that must work on older versions of Python too, and some parts of its code requires order-preserving dictionaries? The best option is to be clear about your expectations regarding dictionary ordering and use a type that explicitly preserves the order of elements.

Fortunately, the Python standard library provides an ordered dictionary type called `OrderedDict` in the `collections` module. The constructor of this type accepts iterable as the initialization argument. Each element of that argument should be a pair of a dictionary key and value, as in the following example:

```
>>> from collections import OrderedDict
>>> OrderedDict((str(number), None) for number in range(5)).keys()
odict_keys(['0', '1', '2', '3', '4'])
```

It also has some additional features, such as popping items from both ends using the `popitem()` method, or moving the specified element to one of the ends using the `move_to_end()` method. A full reference on that collection is available in the Python documentation (refer to `https://docs.python.org/3/library/collections.html`). Even if you target only Python in version 3.7 or newer, which guarantees the preservation of the item insertion order, the `OrderedDict` type is still useful. It allows you to make your intention clear. If you define your variable with `OrderedDict` instead of a plain `dict`, it becomes obvious that, in this particular case, the order of inserted items is important.

The last interesting note is that, in very old code bases, you can find `dict` as a primitive set implementation that ensures uniqueness of elements. While this will give proper results, you should avoid such use of that type unless you target Python versions lower than 2.3. Using dictionaries in this way is wasteful in terms of resources. Python has a built-in `set` type that serves this purpose. In fact, it has very similar internal implementation to dictionaries in CPython, but offers some additional features, as well as specific set-related optimizations.

Sets

Sets are a very robust data structure that are mostly useful in situations where the order of elements is not as important as their uniqueness. They are also useful if you need to efficiently check efficiency if the element is contained in a collection. Sets in Python are generalizations of mathematic sets, and are provided as built-in types in two flavors:

- `set()`: This is a mutable, non-ordered, finite collection of unique, immutable (hashable) objects
- `frozenset()`: This is an immutable, hashable, non-ordered collection of unique, immutable (hashable) objects

The immutability of `frozenset()` objects makes it possible for them to be included as dictionary keys and also other `set()` and `frozenset()` elements. A plain mutable `set()` object cannot be used within another `set()` or `frozenset()`. Attempting to do so will raise a `TypeError` exception, as in the following example:

```
>>> set([set([1,2,3]), set([2,3,4])])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
```

On the other hand, the following set initializations are completely correct, and do not raise exceptions:

```
>>> set([frozenset([1,2,3]), frozenset([2,3,4])])
{frozenset({1, 2, 3}), frozenset({2, 3, 4})}
>>> frozenset([frozenset([1,2,3]), frozenset([2,3,4])])
frozenset({frozenset({1, 2, 3}), frozenset({2, 3, 4})})
```

Mutable sets can be created in three ways:

- Using a `set()` call that accepts optional iterables as the initialization argument, such as `set([0, 1, 2])`
- Using a set comprehension such as `{element for element in range(3)}`
- Using set literals such as `{1, 2, 3}`

Note that using literals and comprehensions for sets requires extra caution, because they are very similar in form to dictionary literals and comprehensions. Also, there is no literal for empty set objects – empty curly brackets `{}` are reserved for empty dictionary literals.

Implementation details

Sets in CPython are very similar to dictionaries. As a matter of fact, they are implemented like dictionaries with dummy values, where only keys are actual collection elements. Sets also exploit this lack of values in mapping for additional optimizations.

Thanks to this, sets allow very fast additions, deletions, and checks for element existence with the average time complexity equal to $O(1)$. Still, since the implementation of sets in CPython relies on a similar hash table structure, the worst case complexity for these operations is still $O(n)$, where n is the current size of a set.

Other implementation details also apply. The item to be included in a set must be hashable, and, if instances of user-defined classes in the set are hashed poorly, this will have a negative impact on their performance.

Despite their conceptual similarity to dictionaries, sets in Python 3.7 do not preserve the order of elements in specification, or as a detail of CPython implementation.

Let's take a look at the supplemental data types and containers.

Supplemental data types and containers

In the *Python's built-in types* section, we concentrated mostly on those data types that have dedicated literals in the Python syntax. These were also the types that are implemented at the interpreter-level. However, Python's standard library offers a great collection of supplemental data types that can be effectively used in places where the basic built-in types show their shortcomings, or places where the nature of the data requires specialized handling (for example, in the presentation of time and dates).

The most common are data containers that are found in the collections, and we have already briefly mentioned two of them: `deque` and `OrderedDict`. However, the landscape of data structures available for Python programmers is enormous and almost every module of the Python standard library defines some specialized types for handling the data of different problem domains.

In this section, we will focus only on the types that can be considered as general-purpose.

Specialized data containers from the collections module

Every data structure has its shortcomings. There is no single collection that can suit every problem, and four basic types of them (tuple, list, set, and dictionary) is still not a wide range of choices. These are the most basic and important collections that have a dedicated literal syntax. Fortunately, Python provides far more options in its standard library through the `collections` built-in module. Here are the most important universal data containers provided by this module:

- `namedtuple()`: This is a factory function for creating tuple subclasses whose indexes can be accessed as named attributes
- `deque`: This is a double-ended queue—a list-like generalization of stacks and queues with fast appends and pops on both ends
- `ChainMap`: This is a dictionary-like class to create a single view of multiple mappings

- `Counter`: This is a dictionary subclass for counting hashable objects
- `OrderedDict`: This is a dictionary subclass that preserves the order that the entries were added in
- `defaultdict`: This is a dictionary subclass that can supply missing values using a user-defined factory function

> More details on selected collections from the collections module and some advice on where it is worth using them is provided in `Chapter 14`, *Optimization – Some Powerful Techniques*.

Symbolic enumeration with the enum module

One of the special handy types found in the Python standard 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`:

```
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 `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 the previous example rewritten with the use of `auto` in it:

```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 a finite number of values/choices. For instance, they can be used to define statues 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 RuntimeError(
                "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 bit manipulation of numbers is very common, like C. In Python, this can be done in a more expressive and convenient way using `FlagEnum`:

```
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 operations (the `|` and `&` operators) and test for flag membership with the `in` keyword. Here are some examples for 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.

The next section will describe various advanced syntax.

Advanced syntax

It is hard to objectively tell which element of language syntax is advanced. For the purpose of this chapter, we will consider advanced syntax elements to be the elements that do not directly relate to any specific built-in datatypes, and which are relatively hard to grasp in the beginning. The most common Python features that may be hard to understand are the following:

- Iterators
- Generators
- Decorators
- Context managers

Iterators

An **iterator** is nothing more than a container object that implements the iterator protocol. This protocol consists of two methods:

- `__next__`: This returns the next item of the container
- `__iter__`: This returns the iterator itself

Iterators can be created from a sequence using the `iter` built-in function. Consider the following example:

```
>>> i = iter('abc')
>>> next(i)
'a'
>>> next(i)
'b'
>>> next(i)
'c'
>>> next(i)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
StopIteration
```

When the sequence is exhausted, a `StopIteration` exception is raised. It makes iterators compatible with loops, since they catch this exception as a signal to end the iteration. If you create a custom iterator, you need to provide objects with the implementation of __next__, which iterates the state of the object, and the __iter__ method, which returns the iterable.

Both methods are often implemented inside of the same class. The following is an example of the `CountDown` class, which allows us to iterate numbers toward `0`:

```
class CountDown:
    def __init__(self, step):
        self.step = step

    def __next__(self):
        """Return the next element."""
        if self.step <= 0:
            raise StopIteration
        self.step -= 1
        return self.step

    def __iter__(self):
        """Return the iterator itself."""
        return self
```

The preceding class implementation allows it to iterate over itself. This means that once you iterate over its content, the iterable is exhausted and cannot be iterated anymore:

```
>>> count_down = CountDown(4)
>>> for element in count_down:
...     print(element)
... else:
...     print("end")
...
3
2
1
0
end
>>> for element in count_down:
...     print(element)
... else:
...     print("end")
...
end
```

If you want your iterator to be reusable, you can always split its implementation into two classes in order to separate the iteration state and actual iterator objects, as in the following example:

```
class CounterState:
    def __init__(self, step):
        self.step = step

    def __next__(self):
```

```
            """Move the counter step towards 0 by 1."""
            if self.step <= 0:
                raise StopIteration
            self.step -= 1
            return self.step


class CountDown:
    def __init__(self, steps):
        self.steps = steps

    def __iter__(self):
        """Return iterable state"""
        return CounterState(self.steps)
```

If you separate your iterator from its state, you will ensure that it can't be exhausted:

```
>>> count_down = CountDown(4)
>>> for element in count_down:
...     print(element)
... else:
...     print("end")
...
3
2
1
0
end
>>> for element in count_down:
...     print(element)
... else:
...     print("end")
...
3
2
1
0
end
```

Iterators themselves are a low-level feature and concept, and a program can live without them. However, they provide the base for a much more interesting feature: generators.

Generators and yield statements

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:

```
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:

```
>>> fib = fibonacci()
>>> next(fib)
1
>>> next(fib)
1
>>> next(fib)
2
>>> [next(fib) for i in range(10)]
[3, 5, 8, 13, 21, 34, 55, 89, 144, 233]
```

Our `fibonacci()` function returns a `generator` object, a special iterator, which 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 pseudocode.

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.

The `tokenize` module from the standard library, for instance, generates tokens out of a stream of text working on them in a line-by-line fashion:

```
>>> import io
>>> import tokenize
>>> code = io.StringIO("""
... if __name__ == "__main__":
... print("hello world!")
... """)
>>> tokens = tokenize.generate_tokens(code.readline)
>>> next(tokens)
TokenInfo(type=56 (NL), string='\n', start=(1, 0), end=(1, 1), line='\n')
>>> next(tokens)
TokenInfo(type=1 (NAME), string='if', start=(2, 0), end=(2, 2), line='if
__name__ == "__main__":\n')
>>> next(tokens)
TokenInfo(type=1 (NAME), string='__name__', start=(2, 3), end=(2, 11),
line='if __name__ == "__main__":\n')
>>> next(tokens)
TokenInfo(type=53 (OP), string='==', start=(2, 12), end=(2, 14), line='if
__name__ == "__main__":\n')
>>> next(tokens)
TokenInfo(type=3 (STRING), string='"__main__"', start=(2, 15), end=(2, 25),
line='if __name__ == "__main__":\n')
>>> next(tokens)
TokenInfo(type=53 (OP), string=':', start=(2, 25), end=(2, 26), line='if
__name__ == "__main__":\n')
>>> next(tokens)
TokenInfo(type=4 (NEWLINE), string='\n', start=(2, 26), end=(2, 27),
line='if __name__ == "__main__":\n')
>>> next(tokens)
TokenInfo(type=5 (INDENT), string=' ', start=(3, 0), end=(3, 4), line='
print("hello world!")\n')
```

Here, we can see that `open.readline` iterates over the lines of the file and `generate_tokens` iterates over them in a pipeline, doing some additional work. Generators can also help in breaking the complexity of your code, and increasing the efficiency of some data transformation algorithms if they can be divided into separate processing steps. Thinking of each processing step as an `iterator` and then combining them into a high-level function is a great way to avoid big, ugly, and unreadable functions. Moreover, this can provide live feedback to the whole processing chain.

In the following example, each function defines a transformation over a sequence. They are then chained and applied together. Each call processes one element and returns its result:

```python
def capitalize(values):
    for value in values:
         yield value.upper()


def hyphenate(values):
    for value in values:
        yield f"-{value}-"


def leetspeak(values):
    for value in values:
        if value in {'t', 'T'}:
            yield '7'
        elif value in {'e', 'E'}:
            yield '3'
        else:
            yield value


def join(values):
    return "".join(values)
```

Once you split your data processing pipeline into several independent steps, you can combine them in different ways:

```python
>>> join(capitalize("This will be uppercase text"))
'THIS WILL BE UPPERCASE TEXT'
>>> join(leetspeak("This isn't a leetspeak"))
"7his isn'7 a l337sp3ak"
>>> join(hyphenate("Will be hyphenated by words".split()))
'-Will--be--hyphenated--by--words-'
>>> join(hyphenate("Will be hyphenated by character"))
'-W--i--l--l-- --b--e-- --h--y--p--h--e--n--a--t--e--d-- --b--y-- --c--h--
a--r--a--c--t--e--r-'
```

> **Keep the code simple, not the data**
> It is better to have a lot of simple iterable functions that work over sequences of values than a complex function that computes the result for one value at a time.

Another important feature that's available in Python regarding generators is the ability to interact with the code that's called with the `next()` function. The `yield` statement becomes an expression, and some value can be passed through it to the decorator with a new generator method, named `send()`:

```python
def psychologist():
    print('Please tell me your problems')
    while True:
        answer = (yield)
        if answer is not None:
            if answer.endswith('?'):
                print("Don't ask yourself too much questions")
            elif 'good' in answer:
                print("Ahh that's good, go on")
            elif 'bad' in answer:
                print("Don't be so negative")
```

Here is an example session with our `psychologist()` function:

```python
>>> free = psychologist()
>>> next(free)
Please tell me your problems
>>> free.send('I feel bad')
Don't be so negative
>>> free.send("Why I shouldn't ?")
Don't ask yourself too much questions
>>> free.send("ok then i should find what is good for me")
Ahh that's good, go on
```

The `send()` method acts similarly to the `next()` function, but makes the `yield` statement return the value passed to it inside of the function definition. The function can, therefore, change its behavior depending on the client code. Two other methods are available to complete this behavior: `throw()` and `close()`. They allow us to inject exceptions into the generator:

- `throw()`: This allows the client code to send any kind of exception to be raised.
- `close()`: This acts in the same way, but raises a specific exception, `GeneratorExit`. In this case, the generator function must raise `GeneratorExit` again, or `StopIteration`.

> Generators are the basis of other concepts that are available in Python, such as coroutines and asynchronous concurrency, which are covered in `Chapter 15`, *Concurrency*.

Decorators

Decorators were added in Python to make function and method wrapping easier to read and understand. A decorator is simply any function that receives a function and returns an enhanced one. The original use case was to be able to define the methods as class methods or static methods on the head of their definition. Without the decorator syntax, it would require a rather sparse and repetitive definition:

```
class WithoutDecorators:
    def some_static_method():
        print("this is static method")
    some_static_method = staticmethod(some_static_method)
    def some_class_method(cls):
        print("this is class method")
    some_class_method = classmethod(some_class_method)
```

Dedicated decorator syntax code is shorter and easier to understand:

```
class WithDecorators:
    @staticmethod
    def some_static_method():
        print("this is static method")
    @classmethod
    def some_class_method(cls):
        print("this is class method")
```

Let's take a look at the general syntax and possible implementations of decorators.

General syntax and possible implementations

The decorator is generally a named callable object (`lambda` expressions are not allowed) that accepts a single argument when called (it will be the decorated function) and returns another callable object. **Callable** is used here instead of a function with premeditation. 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 a syntactic sugar. Consider the following decorator usage:

```
@some_decorator
def decorated_function():
    pass
```

This can always be replaced by an explicit decorator call and function reassignment:

```
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.

> **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 the user tries to call an object that's been decorated this way. This part of the decorator syntax creates a field for some interesting experimentation.

As a function

There are many ways to write custom decorators, but the simplest way is to write a function that returns a sub-function that wraps the original function call.

The generic patterns is as follows:

```
def mydecorator(function):
    def wrapped(*args, **kwargs):
        # do some stuff before the original
        # function gets called
        result = function(*args, **kwargs)
        # do some stuff after function call and
        # return the result
        return result
    # return wrapper as a decorated function
    return wrapped
```

As a class

While decorators can almost always be implemented using functions, there are some situations when using user-defined classes is a better option. This is often true when the decorator needs complex parameterization, or if it depends on a specific state.

The generic pattern for a non-parameterized decorator defined as a class is as follows:

```
class DecoratorAsClass:
    def __init__(self, function):
        self.function = function

    def __call__(self, *args, **kwargs):
        # do some stuff before the original
        # function gets called
        result = self.function(*args, **kwargs)
        # do some stuff after function call and
        # return the result
        return result
```

Parametrizing decorators

In real usage scenarios, there is often a need to use decorators that can be parameterized. When the function is used as a decorator, then the solution is simple – a second level of wrapping has to be used. Here is a simple example of the decorator that repeats the execution of a decorated function the specified number of times every time it is called:

```
def repeat(number=3):
    """Cause decorated function to be repeated a number of times.
    Last value of original function call is returned as a result.
    :param number: number of repetitions, 3 if not specified
    """
    def actual_decorator(function):
        def wrapper(*args, **kwargs):
            result = None
            for _ in range(number):
                result = function(*args, **kwargs)
            return result
        return wrapper
    return actual_decorator
```

The decorator that's defined this way can accept parameters:

```
>>> @repeat(2)
... def print_my_call():
...     print("print_my_call() called!")
...
>>> print_my_call()
print_my_call() called!
print_my_call() called!
```

Note that, even if the parameterized decorator has default values for its arguments, the parentheses after its name is required. The correct way to use the preceding decorator with default arguments is as follows:

```
>>> @repeat()
... def print_my_call():
...     print("print_my_call() called!")
...
>>> print_my_call()
print_my_call() called!
print_my_call() called!
print_my_call() called!
```

Missing these parentheses will result in the following error when the decorated function is called:

```
>>> @repeat
... def print_my_call():
...     print("print_my_call() called!")
...
>>> print_my_call()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: actual_decorator() missing 1 required positional
argument: 'function'
```

Introspection preserving decorators

A common mistake when using decorators is not preserving function metadata (mostly docstring and original name) when using decorators. All the previous examples have this issue. They create a new function by composition and return a new object without any respect to the identity of the original decorated object. This makes the debugging of functions decorated that way harder and will also break most of the auto-documentation tools that you may want to use, because the original docstrings and function signatures are no longer accessible.

Let's see this in detail. Let's assume that we have some dummy decorator that does nothing and some other functions decorated with it:

```python
def dummy_decorator(function):
    def wrapped(*args, **kwargs):
        """Internal wrapped function documentation."""
        return function(*args, **kwargs)
    return wrapped


@dummy_decorator
def function_with_important_docstring():
    """This is important docstring we do not want to lose."""
```

If we inspect `function_with_important_docstring()` in the Python interactive session, we can see that it has lost its original name and docstring:

```python
>>> function_with_important_docstring.__name__
'wrapped'
>>> function_with_important_docstring.__doc__
'Internal wrapped function documentation.'
```

A proper solution to this problem is to use the `wraps()` decorator, provided by the `functools` module:

```python
from functools import wraps


def preserving_decorator(function):
    @wraps(function)
    def wrapped(*args, **kwargs):
        """Internal wrapped function documentation."""
        return function(*args, **kwargs)
    return wrapped


@preserving_decorator
def function_with_important_docstring():
    """This is important docstring we do not want to lose."""
```

With the decorator defined in such a way, all the important function metadata is preserved:

```
>>> function_with_important_docstring.__name__
'function_with_important_docstring.'
>>> function_with_important_docstring.__doc__
'This is important docstring we do not want to lose.'
```

Let's see the usage of decorators in the next section.

Usage and useful examples

Since decorators are loaded by the interpreter when the module is first read, their usage should be limited to wrappers that can be generically applied. If a decorator is tied to the method's class or to the signature of the function that it enhances, it should be refactored into a regular callable to avoid complexity. Often, it is good practice to group them in dedicated modules that reflect their area of use so that it will be easier to maintain them.

The common patterns for decorators are the following:

- Argument checking
- Caching
- Proxy
- Context provider

Argument checking

Checking the arguments that a function receives or returns can be useful when it is executed in a specific context. For example, if a function is to be called through XML-RPC, Python will not be able to directly provide its full signature like it does in statically-typed languages. This feature is needed to provide introspection capabilities when the XML-RPC client asks for the function signatures.

> **The XML-RPC protocol**
> The XML-RPC protocol is a lightweight **Remote Procedure Call** protocol that uses XML over HTTP to encode its calls. It is often used instead of SOAP for simple client-server exchanges. Unlike SOAP, which provides a page that lists all callable functions (WSDL), XML-RPC does not have a directory of available functions. An extension of the protocol that allows the discovery of the server API was proposed, and Python's `xmlrpc` module implements it (refer to `https://docs.python.org/3/library/xmlrpc.server.html`).

A custom decorator can provide this type of signature. It can also makes sure that what goes in and comes out respects the defined signature parameters:

```python
rpc_info = {}


def xmlrpc(in_=(), out=(type(None),)):
    def _xmlrpc(function):
        # registering the signature
        func_name = function.__name__
        rpc_info[func_name] = (in_, out)
        def _check_types(elements, types):
            """Subfunction that checks the types."""
            if len(elements) != len(types):
                raise TypeError('argument count is wrong')
            typed = enumerate(zip(elements, types))
            for index, couple in typed:
               arg, of_the_right_type = couple
                if isinstance(arg, of_the_right_type):
                    continue
                raise TypeError(
                    'arg #%d should be %s' % (index,
                        of_the_right_type))

        # wrapped function
        def __xmlrpc(*args):  # no keywords allowed
            # checking what goes in
            checkable_args = args[1:]  # removing self
            _check_types(checkable_args, in_)
            # running the function
            res = function(*args)
            # checking what goes out
            if not type(res) in (tuple, list):
                checkable_res = (res,)
            else:
                checkable_res = res
            _check_types(checkable_res, out)

            # the function and the type
            # checking succeeded
            return res
        return __xmlrpc
    return _xmlrpc
```

The decorator registers the function into a global dictionary, and keeps a list of the types for its arguments and for the returned values. Note that this example was highly simplified, just to demonstrate the idea of argument-checking decorators.

A usage example is as follws:

```
class RPCView:
    @xmlrpc((int, int))  # two int -> None
    def accept_integers(self, int1, int2):
        print('received %d and %d' % (int1, int2))

    @xmlrpc((str,), (int,))  # string -> int
    def accept_phrase(self, phrase):
        print('received %s' % phrase)
        return 12
```

When it is read, this class definition populates the `rpc_infos` dictionary and can be used in a specific environment, where the argument types are checked:

```
>>> rpc_info
{'meth2': ((<class 'str'>,), (<class 'int'>,)), 'meth1': ((<class
 'int'>, <class 'int'>), (<class 'NoneType'>,))}
>>> my = RPCView()
>>> my.accept_integers(1, 2)
received 1 and 2
>>> my.accept_phrase(2)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 26, in __xmlrpc
  File "<input>", line 20, in _check_types
TypeError: arg #0 should be <class 'str'>
```

Caching

The caching decorator is quite similar to argument checking, but focuses on those functions whose internal state does not affect the output. Each set of arguments can be linked to a unique result. This style of programming is the characteristic of **functional programming**, and can be used when the set of input values is finite.

Therefore, a caching decorator can keep the output together with the arguments that were needed to compute it, and return it directly on subsequent calls.

This behavior is called **memoizing**, and is quite simple to implement as a decorator:

```python
"""This module provides simple memoization arguments
that is able to store cached return results of
decorated function for specified period of time.
"""
import time
import hashlib
import pickle

cache = {}


def is_obsolete(entry, duration):
    """Check if given cache entry is obsolete"""
    return time.time() - entry['time'] > duration


def compute_key(function, args, kw):
    """Compute caching key for given value"""
    key = pickle.dumps((function.__name__, args, kw))
    return hashlib.sha1(key).hexdigest()


def memoize(duration=10):
    """Keyword-aware memoization decorator

    It allows to memoize function arguments for specified
    duration time.
    """
    def _memoize(function):
        def __memoize(*args, **kw):
            key = compute_key(function, args, kw)

            # do we have it already in cache?
            if (
                key in cache and
                not is_obsolete(cache[key], duration)
            ):
                # return cached value if it exists
                # and isn't too old
                print('we got a winner')
                return cache[key]['value']

            # compute result if there is no valid
            # cache available
            result = function(*args, **kw)
            # store the result for later use
```

```
                cache[key] = {
                    'value': result,
                    'time': time.time()
                }
            return result
        return __memoize
    return _memoize
```

A `SHA` hash key is built using the ordered argument values, and the result is stored in a global dictionary. The hash is made using a pickle, which is a bit of a shortcut to freeze the state of all objects that are passed as arguments, ensuring that all arguments are good candidates. If a thread or a socket is used as an argument, a `PicklingError` will occur (refer to `https://docs.python.org/3/library/pickle.html`). The duration parameter is used to invalidate the cached value when too much time has passed since the last function call.

Here's an example of the memoize decorator usage (assuming that the previous snippet is stored in the `memoize` module):

```
>>> from memoize import memoize
>>> @memoize()
... def very_very_very_complex_stuff(a, b):
...     # if your computer gets too hot on this calculation
...     # consider stopping it
...     return a + b
...
>>> very_very_very_complex_stuff(2, 2)
4
>>> very_very_very_complex_stuff(2, 2)
we got a winner
4
>>> @memoize(1) # invalidates the cache after 1 second
... def very_very_very_complex_stuff(a, b):
...     return a + b
...
>>> very_very_very_complex_stuff(2, 2)
4
>>> very_very_very_complex_stuff(2, 2)
we got a winner
4
>>> cache
{'c2727f43c6e39b3694649ee0883234cf': {'value': 4, 'time':
1199734132.7102251)}
>>> time.sleep(2)
>>> very_very_very_complex_stuff(2, 2)
4
```

Caching expensive functions can dramatically increase the overall performance of a program, but it has to be used with care. The cached value could also be tied to the function instead of using a centralized dictionary itself to better manage the scope and life cycle of the cache. But, in any case, a more efficient decorator would use a specialized cache library and/or dedicated caching service based on advanced caching algorithms. `Memcached` is a well-known example of such a caching service and can be easily used in Python.

> Chapter 14, *Optimization – Some Powerful Techniques*, provides detailed information and examples for various caching techniques.

Proxy

Proxy decorators are used to tag and register functions with a global mechanism. For instance, a security layer that protects access to the code, depending on the current user, can be implemented using a centralized checker with an associated permission required by the callable:

```python
class User(object):
    def __init__(self, roles):
        self.roles = roles


class Unauthorized(Exception):
    pass


def protect(role):
    def _protect(function):
        def __protect(*args, **kw):
            user = globals().get('user')
            if user is None or role not in user.roles:
                raise Unauthorized("I won't tell you")
            return function(*args, **kw)
        return __protect
    return _protect
```

This model is often used in Python web frameworks to define the security over publishable resources. For instance, Django provides decorators to secure its access to views representing web resources.

Here's an example, where the current user is kept in a global variable. The decorator checks his or her roles when the method is accessed (the previous snippet is stored in the `users` module):

```
>>> from users import User, protect
>>> tarek = User(('admin', 'user'))
>>> bill = User(('user',))
>>> class RecipeVault(object):
...     @protect('admin')
...     def get_waffle_recipe(self):
...         print('use tons of butter!')
...
>>> my_vault = RecipeVault()
>>> user = tarek
>>> my_vault.get_waffle_recipe()
use tons of butter!
>>> user = bill
>>> my_vault.get_waffle_recipe()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in wrap
__main__.Unauthorized: I won't tell you
```

Context provider

A context provider decorator makes sure that the function can run in the correct context, or executes some code before and/or after executing a decorated function. In other words, it sets and unsets a specific execution environment. For example, when a data item has to be shared among several threads, a lock has to be used to ensure that it is protected from multiple access. This lock can be coded in a decorator as follows:

```
from threading import RLock
lock = RLock()


def synchronized(function):
    def _synchronized(*args, **kw):
        lock.acquire()
        try:
            return function(*args, **kw)
        finally:
            lock.release()
```

```
    return _synchronized


@synchronized
def thread_safe():  # make sure it locks the resource
    pass
```

Context decorators are often being replaced by the usage of context managers (the `with` statement) which are also described later in this chapter.

Context managers – the with statement

The `try...finally` statement is useful to ensure some cleanup code is run, even if an error is raised. There are many use cases for this, such as the following:

- Closing a file
- Releasing a lock
- Making a temporary code patch
- Running protected code in a special environment

The `with` statement factors out these use cases by providing a simple way to wrap a block of code with methods defined within the context manager. This allows us to call some code before and after block execution, even if this block raises an exception. For example, working with a file is often done like so:

```
>>> hosts = open('/etc/hosts')
>>> try:
...     for line in hosts:
...         if line.startswith('#'):
...             continue
...         print(line.strip())
... finally:
...     hosts.close()
...
127.0.0.1       localhost
255.255.255.255 broadcasthost
::1             localhost
```

> This example is specific to Linux, since it reads the host file located in the `/etc/` directory, but any text file could have been used here in the same way.

By using the `with` statement, it can be rewritten into the following code, which is shorter and cleaner:

```
>>> with open('/etc/hosts') as hosts:
...     for line in hosts:
...         if line.startswith('#'):
...             continue
...         print(line.strip())
...
127.0.0.1       localhost
255.255.255.255 broadcasthost
::1             localhost
```

In the preceding example, the `open()` function is used as a context manager that ensures that the file will be closed after executing the `for` loop, even if some exception occurs in the process.

Some other common items from the Python standard library that are compatible with this statement are classes from the `threading` module:

- `threading.Lock`
- `threading.RLock`
- `threading.Condition`
- `threading.Semaphore`
- `threading.BoundedSemaphore`

The general syntax and possible implementations

The general syntax for the `with` statement in the simplest form is as follows:

```
with context_manager:
    # block of code
    ...
```

Additionally, if the context manager provides a context variable, it can be stored locally using the `as` clause:

```
with context_manager as context:
    # block of code
    ...
```

Note that multiple context managers can be used at once, as follows:

```
with A() as a, B() as b:
    ...
```

This is equivalent to nesting them, as follows:

```
with A() as a:
    with B() as b:
        ...
```

As a class

Any object that implements the **context manager protocol** can be used as a context manager. This protocol consists of two special methods:

- `__enter__(self)`: This allows you to define what should happen before executing the code that is wrapped with context manager and returns context variable
- `__exit__(self, exc_type, exc_value, traceback)`: This allows you to perform additional cleanup operations after executing the code wrapped with context manager, and captures all exceptions that were raised in the process

In short, the execution of the `with` statement proceeds as follows:

1. The `__enter__` method is invoked. Any return value is bound to target the specified `as` clause.
2. The inner block of code is executed.
3. The `__exit__` method is invoked.

`__exit__` receives three arguments that are filled when an error occurs within the code block. If no error occurs, all three arguments are set to `None`. When an error occurs, the `__exit__()` method should not re-raise it, as this is the responsibility of the caller. It can prevent the exception being raised, though, by returning `True`. This is provided to allow for some specific use cases, such as the `contextmanager` decorator, which we will see in the next section. But, for most use cases, the right behavior for this method is to do some cleanup, as would be done by the `finally` clause. Usually, no matter what happens in the block, it does not return anything.

The following is an example of a dummy context manager that implements this protocol to better illustrate how it works:

```python
class ContextIllustration:
    def __enter__(self):
        print('entering context')

    def __exit__(self, exc_type, exc_value, traceback):
        print('leaving context')

        if exc_type is None:
            print('with no error')
        else:
            print(f'with an error ({exc_value})')
```

When run without exceptions raised, the output is as follows (the previous snippet is stored in the `context_illustration` module):

```
>>> from context_illustration import ContextIllustration
>>> with ContextIllustration():
...     print("inside")
...
entering context
inside
leaving context
with no error
```

When the exception is raised, the output is as follows:

```
>>> from context_illustration import ContextIllustration
>>> with ContextIllustration():
...     raise RuntimeError("raised within 'with'")
...
entering context
leaving context
with an error (raised within 'with')
Traceback (most recent call last):
  File "<input>", line 2, in <module>
RuntimeError: raised within 'with'
```

As a function – the contextlib module

Using classes seems to be the most flexible way to implement any protocol provided in the Python language, but may be too much boilerplate for many simple use cases. A module was added to the standard library to provide helpers that simplify the creation of custom context managers. The most useful part of it is the `contextmanager` decorator. It allows us to provide both __enter__ and __exit__ procedures of the context manager within a single function, separated by a `yield` statement (note that this makes the function a generator). The previous example, when written with this decorator like would look like the following:

```python
from contextlib import contextmanager


@contextmanager
def context_illustration():
    print('entering context')

    try:
        yield
    except Exception as e:
        print('leaving context')
        print(f'with an error ({e})')
        # exception needs to be reraised
        raise
    else:
        print('leaving context')
        print('with no error')
```

If any exception occurs, the function needs to re-raise it in order to pass it along. Note that the `context_illustration` module could have some arguments if needed. This small helper simplifies the normal class-based context manager API exactly like generators do with the class-based iterator API.

The four other helpers provided by this module are as follows:

- `closing(element)`: This returns the context manager that calls the element's `close()` method on exit. This is useful for classes that deal with streams and files.
- `supress(*exceptions)`: This suppresses any of the specified exceptions if they occur in the body of the `with` statement.
- `redirect_stdout(new_target)` and `redirect_stderr(new_target)`: These redirect the `sys.stdout` or `sys.stderr` output of any code within the block to another file or file-like object.

Let's take a look at the functional-style features of Python.

Functional-style features of Python

Programming paradigm is a very important concept that allows us to classify different programming languages. Programming paradigm defines a specific way of thinking about language execution models (definition of how work takes place) or about the structure and organization of the code. There are many programming paradigms, but they are usually grouped into two main categories:

- **Imperative paradigms**, in which the programmer is mostly concerned about the program state and the program itself is a definition of how the computer should manipulate its state to generate the expected result
- **Declarative paradigms**, in which the programmer is concerned mostly about a formal definition of the problem or properties of the desired result and not defining how this result should be computed

Due to its execution model and omnipresent classes and objects, the paradigms that are the most natural to Python are object-oriented programming and structured programming. These are also the two most common imperative programming paradigms among all modern programming languages. However Python is considered a multi-paradigm language and contains features that are common to both imperative and declarative languages.

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 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.

Let's discuss functional programming in the next section.

What is functional programming?

Functional programming is a paradigm where the program is mainly an evaluation of (mathematical) functions, and is not through a series of defined steps that change the state of the program. Purely functional programs avoid the change of state (side effects) and mutable data. In Python, functional programming is realized through the use of complex expressions and declarations of functions.

One of the best ways to better understand the general concept of functional programming is through 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 or 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 object-oriented programming, where class instances are objects that are used to encapsulate the state of application, and methods are functions bound to those objects that are supposed to manipulate the state of these objects.
- **Referential transparency:** When a function or expression is referentially transparent, it can be replaced with the value that corresponds to its inputs 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 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 which 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.
- **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 that is concerned only with pure functions, and 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. Python offers a large variety of features that, for years, were only accessible in purely functional languages, so it is possible to write substantial amounts of code in a functional way, even though Python isn't functional by itself.

Let's take a look at Lambda functions in the next section.

Lambda functions

Lambda functions are a very popular programming concept that is especially profound in functional programming. In other programming languages, lambda function 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).

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 through 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:

```
import math


def circle_area(radius):
    return math.pi * radius ** 2
```

The same function expressed as a lambda function would take the following form:

```
lambda radius: math.pi * radius ** 2
```

Lambda in functions are anonymous, but it doesn't mean they cannot be referred to by any 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 undistinguishable from other functions, except for some metadata attributes. The following transcripts from interactive interpreter sessions illustrates this:

```
>>> 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>'
```

Let's take a look at the `map()`, `filter()`, and `reduce()` functions in the next sections.

map(), filter(), and reduce()

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 data 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 additional imports.

`map(fun, 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 items as there is iterables on every map step. If iterables are of different sizes, `map()` will stop until 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 positive integers, including `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:

```
>>> list(map(print, range(5), range(4), range(5)))
0 0 0
1 1 1
2 2 2
3 3 3
```

`filter(function, 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 meet the predicate defined by the `function` argument. The following are examples of the `filter()` functions 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_with_s = filter(lambda animal: 's' in animal, animals)
>>> print(f"Animals with letter 's' are: {list(animals_with_s)}")
Animals with letter 's' are: ['snake', 'squirrel']
```

The `reduce(function, iterable)` works completely opposite to `map()`. Instead of taking items of `iterable` and mapping them to the `function` return values in a one-by-one fashion, it cumulatively performs operations specified by `function` to all `iterable` items. Let's consider following the 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 infinite sequence to a `list` type or trying to ordinarily loop over such a sequence will result in program that does not ever end. However the return values of `map()` and `filter()` are iterators, and we already learned in this chapter that we can obtain new values from iterators using the `next()` function. The common `range()` function we have used in previous examples unfortunately requires finite input value, but the `itertools` module provides a useful `count()` function that allows you to count from a specific number in any direction *ad infinitum*. The following example shows how all these functions can be used together to generate an infinite sequence in a declarative way:

```
>>> from itertools import count
>>> sequence = filter(
...     # We want to accept only values divisible by 3
...     # that are not divisible by 2
...     lambda square: square % 3 == 0 and square % 2 == 1,
...     map(
...         # and all numbers must be squares
...         lambda number: number ** 2,
...         # and we count towards infinity
...         count()
...     )
... )
>>> next(sequence)
9
>>> next(sequence)
81
>>> next(sequence)
225
>>> next(sequence)
441
```

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.

Let's take a look at partial objects and `partial()` functions.

Partial objects and partial() functions

Partial objects are loosely related to the concept of partial functions in mathematics. A prtial function is a generalization of a mathematical function in a way that isn't forced to map every possible input value (domain) to its results. In Python, partial objects can be used to slice the possible input domain 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` function 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 functions, 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 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 functions. The following example shows how various functions that have been presented in this chapter can be used to create a simple generator of infinite powers of the number two without any explicit function definition:

```
from functools import partial
from itertools import count
infinite_powers_of_2 = map(partial(pow, 2), count())
```

> 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 others, allow us to cycle containers, group their contents, split iterables in chunks, and chain multiple iterables into one and many more, and 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.

Let's take a look at generator expressions in the next section.

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 dict, set, and list literals. A generator expression is denoted with 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 iterators. They also allow `if` clauses to filter specific elements. This means that you can often replace complex `map()` and `filter()` constructions with more readable and compact generator expressions. Consider one of the previous complex `map()`/`filter()` examples compared with the equivalent generator expression:

```
sequence = filter(
    lambda square: square % 3 == 0 and square % 2 == 1,
    map(
        lambda number: number ** 2,
        count()
    )
)

sequence = (
    square for square
```

```
        in (number ** 2 for number in count())
        if square % 3 == 0 and square % 2 == 1
    )
```

Let's discuss function and variable annotations in the next section.

Function and variable annotations

Function annotation is one of the most unique features of Python 3. The official documentation states the following:

> *"Function annotations are completely optional metadata information about the types used by user-defined functions."*

However, they are not restricted to type hinting; also, there is no single feature in Python and its standard library that leverages such annotations. This is why this feature is unique – it does not have any syntactic meaning. Annotations can simply be defined for a function and can be retrieved in runtime, but that is all. What to do with them is left to the developers.

Let's take a look at their general syntax and possible uses in the following sections

The general syntax

A slightly modified example from the Python documentation shows how to define and retrieve function annotations:

```
>>> def f(ham: str, eggs: str = 'eggs') -> str:
...     pass
...
>>> print(f.__annotations__)
{'return': <class 'str'>, 'eggs': <class 'str'>, 'ham': <class 'str'>}
```

As we can see, parameter annotations are defined by the expression evaluating the value of the annotation, preceded by a colon. Return annotations are defined by the expression between the colon denoting the end of the `def` statement and literal `->` that follows the parameter list.

Once defined, annotations are available in the `__annotations__` attribute of the function object as a dictionary and can be retrieved during application runtime.

The fact that any expression can be used as the annotation and that it is located just near the default arguments allows us to create some confusing function definitions, as follows:

```
>>> def square(number: 0<=3 and 1=0) -> (\
...     +9000): return number**2
>>> square(10)
100
```

However, such usage of annotations serves no other purpose than obfuscation, and, even without them, it is relatively easy to write code that is hard to read and maintain.

The possible uses

While annotations have great potential, they are not widely used. An article explaining the new features that were added to Python 3 (refer to `https://docs.python.org/3/whatsnew/3.0.html`) says that the intent of this feature as follows:

> "*The intent is to encourage experimentation through metaclasses, decorators, or frameworks.*"

On the other hand, **PEP 3107**, which officially proposed function annotations, lists the following set of possible use cases:

- Providing typing information:
    - Type checking
    - Let IDEs show what types a function expects and returns
    - Function overloading/generic functions
    - Foreign language bridges
    - Adaptation
    - Predicate logic functions
    - Database query mapping
    - RPC parameter marshaling

- Other information:
    - Documentation for parameters and return values

Although the function annotations are as old as Python 3, it is still very hard to find any popular and actively maintained package that uses them for something else other than type checking. So, except static type checking, function annotations are still mostly only good for experimentation and playing – the initial purpose for their inclusion in the initial release of Python 3.

Static type checking with mypy

Static type checking is a technique that allows us to quickly find possible errors and quality defects in code before it is even executed. It's a natural feature of compiled languages with static typing. Python, of course, lacks such built-in features, but there are some third-party packages that allow us to perform static type analysis in Python in order to improve code quality. Function and variable annotations are currently best utilized as type hints for the exact purpose of static type checking. 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 typing modules (refer to *PEP 484 Type Hints*).

The best thing about `mypy` is that type hinting is completely optional. If you have a very large codebase, you are not forced to suddenly annotate all your code before you start to reap benefits from the static type checking. You can just start to gradually introduce type annotation in the most used code and get increasing benefits over time as your type annotations coverage increases. Also, `mypy` is supported by mainstream Python development in the form of a typeshed project. Typeshed (see `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`.

Let's look at some of the other syntax elements you may not know of yet.

Other syntax elements you may not know of yet

There are some elements of the Python syntax that are not popular and rarely used. This is because they either provide very little gain, or their usage is simply hard to memorize. Due to this, many Python programmers (even with years of experience) simply do not know about their existence. The most notable examples of such features are as follows:

- The `for ... else ...` clause
- Keyword-only arguments

The for ... else ... statement

Using the `else` clause after the `for` loop only allows us to execute a block of code if the loop ended **naturally,** without terminating with the `break` statement:

```
>>> for number in range(1):
...     break
... else:
...     print("no break")
...
>>> for number in range(1):
...     pass
... else:
...     print("no break")
...
no break
```

This comes in handy in some situations, because it helps in removing some **sentinel** variables that may be required if the user wants to store information if a `break` statement occurred. This makes the code cleaner, but can confuse programmers who are not familiar with such syntax. Some say that such meaning of the `else` clause is counterintuitive, but here is an easy tip that will help you remember how does it work – memorizing that `else` clause after the `for` loop simply means **no break**.

Keyword-only arguments

While the `for ... else ...` form of `for` loops is rather a curiosity that not many developers are eager to use, there is at least one lesser-known feature in Python syntax that should be used more often by every Python programmer. This feature is keyword-only arguments.

Keyword-only arguments is a feature that has been in Python for a very long time, but initially was only found in some built-in functions or extensions that were built with the use of the Python/C API. But, starting from Python 3.0, keyword-only arguments are an official element of language syntax that can be used in any function signature. In function signatures, every keyword argument defined after a single literal `*` argument will be marked as keyword-only. Being keyword-only means that you cannot pass a value as an positional argument.

In order to better understand what problem is being solved by keyword-only arguments, let's consider the following set of function stubs that have been defined without that feature:

```python
def process_order(order, client, suppress_notifications=False):
    ...


def open_order(order, client):
    ...


def archive_order(order, client):
    ...
```

The preceding API is pretty consistent. We can clearly see that every function takes exactly two of the same arguments that are probably crucial for every part of the program that needs to deal with orders. We can also see that the additional `suppress_notifications` argument in the `process_order()` function stands out. It has a default value, so it is probably a flag that can be switched on and off. We don't know what this program does, but from the API, we can guess how these functions could be used. The most simple example could be as follows:

```python
order = ...
client = ...

open_order(order, client)
process_order(order, client)
archive_order(order, client)
```

Everything seems clear and simple. However, a curious API designer would see that there is something disturbing in the API design that can become a problem in the future. If there is a need to suppress notifications in the `process_order()` function, the API user can do this in two ways:

```
process_order(order, client, suppress_notifications=True)
process_order(order, client, True)
```

The first usage is best, as it makes the semantics of the function call clear. Here, the two leftmost arguments (order and client) are best when presented as positional arguments, because they have dedicated meaningful variable names, and it also seems that their position is conventional to the API. The meaning of the `suppress_notifications` argument will be totally lost if we present it as a plain literal `True` value.

What is more worrisome is that such lax constraints on API usage puts the API designer in a rather uncomfortable position where he/she must be extremely cautious when extending the existing interfaces. Let's imagine that there is new requirement to suppress payment on demand; we should be able to do this by adding a new argument named `suppress_payment`. Signature change is rather simple:

```
def process_order(
    order, client,
    suppress_notifications=False,
    suppress_payment=False,
):
    ...
```

For us, the intended usage is clear – both `suppress_notifications` and `suppress_payment` should be provided to the function as keyword arguments and not positional arguments. But, what is clear to us doesn't have to be clear to our users. It is just a matter of time until we start seeing function calls like the following:

```
process_order(order, client, True)
process_order(order, client, False)
process_order(order, client, False, False)
process_order(order, client, True, False)
process_order(order, client, False, True)
process_order(order, client, True, True)
```

This pattern is dangerous for yet another reason. Imagine that someone less familiar with the general design of the API added a new argument, not at the end of the argument list but just before other arguments that were supposed to be used as keywords. Such a mistake would invalidate all existing function calls where keywords arguments were wrongly passed positionally.

In large projects, it is extremely hard to protect your code from such misuse. And, without enough protection, every misused call to your functions will, over the years, create a large amount of debt that can greatly reduce your effectiveness. The best way to protect your function signatures from this kind of erosion is by explicitly stating which arguments should be used as keywords. In the discussed example, this approach would look as follows:

```python
def process_order(
    order, client,
    *,
    suppress_notifications=False,
    suppress_payment=False,
):
    ...
```

Summary

This chapter covered various best syntax practices that do not directly relate to Python classes and object-oriented programming. We started by dissecting the syntax for basic built-in types as well as the technical details of their implementation in the CPython interpreter.

After organizing our basic knowledge about Python built-in types, we finally discussed concepts that are truly advanced parts of Python programming language: iterators, generators, decorators, and context managers. Of course, we couldn't make this part completely class-less, as everything in Python is an object, and even elements of syntax that are not object-oriented have their underlying language protocols defined at the class-level object anatomy. So, in order to fulfill the title of this chapter, we then moved our focus to another major aspect of Python programming – the features of language that allow us to program in a functional style.

In order to end this chapter with a lighter tone, we've looked at a few lesser known, but still important and useful, features of Python language.

In the next chapter, we will use everything we learned so far to better understand the object-oriented features of Python. We will look more closely at the concept of language protocols and about method resolution order. We will see that, in Python, every paradigm has its place, and we will discover how object-oriented elements of the language allow for its plasticity.

4

Modern Syntax Elements - Above the Class Level

In this chapter, we will focus on modern syntax elements of Python with regard to classes and object-oriented programming. However, we will not cover the topic of object-oriented design patterns, as they will be discussed in detail in `Chapter 17`, *Useful Design Patterns*. Here, we will perform an overview of the most advanced Python syntax elements that will allow you to improve the code of your classes.

The Python class model as we know it evolved greatly during the history of Python 2. For a long time, we lived in a world where two implementations of the object-oriented programming paradigm coexisted in the same language. These two models were simply referred to as **old-style** and **new-style classes**. Python 3 ended this dichotomy, so that only the model known as the new-style class is available to developers. It is still important to know how both of them worked in Python 2, because it will help you in porting old code and writing backward compatible applications. Knowing how the object model changed will also help you to understand why it is designed that way right now. This is the reason why the following chapter will have many notes about old Python 2 features despite this book being targeted to the latest Python 3 releases.

The following topics will be discussed in this chapter:

- Protocols of the Python language
- Reducing boilerplate with data classes
- Subclassing built-in types
- Accessing methods from superclasses
- Slots

Technical requirements

Code files for this chapter can be found at `https://github.com/PacktPublishing/Expert-Python-Programming-Third-Edition/tree/master/chapter4`.

The protocols of the Python language – dunder methods and attributes

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**. It is simply a speech shorthand for double underscores.

The most common and obvious example of such dunder methods is `__init__()`, which is used for class instance initialization:

```
class CustomUserClass:
    def __init__(self, initiatization_argument):
        ...
```

These methods, either alone or when defined in specific combination, constitute the so-called language protocols. If an object implements specific language protocols, it becomes compatible with specific parts of the Python language syntax. The following is the table of the most important protocols within the Python language:

Protocol name	Methods	Description
Callable protocol	`__call__()`	Allows objects to be called with the parentheses syntax: `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 over using the `for` keyword:
		`for value in instance:`
		`...`

		Allows objects to be indexed with square bracket syntax and queried for length using a built-in function:
Sequence protocol	`__len__()`, `__getitem__()`	`item = instance[index]` `length = len(instance)`

These are the most important language protocols from the perspective of this chapter. The full list is, of course, a lot longer. For instance, Python provides over 50 dunder methods that allow us to emulate numeric values. Each of these methods is correlated to some specific mathematical operator, and so could be considered a separate language protocol. The full list of all the dunder methods can be found in the official documentation of the Python data model (see `https://docs.python.org/3/reference/datamodel.html`).

Language protocols are the foundation of the concept of interfaces in Python. One implementation of Python interfaces is in abstract base classes that allow us to define an arbitrary set of attributes and methods as an interface definition. These definitions of interfaces in the form of abstract classes can be later used to test whether or not the given object is compatible with a specific interface. The `collections.abc` module from the Python standard library provides a collection of abstract base classes that refer to the most common Python language protocol. You'll find more information about interfaces and abstract base classes in the *Interfaces* section of `Chapter 17`, *Useful Design Patterns*.

The same dunder convention is also used for specific attributes of custom user functions and is used to store various metadata about Python objects. These attributes are as follows:

- `__doc__`: A writable attribute that holds the function's documentation. It is, by default, populated by the `docstring` function.
- `__name__`: A writable attribute that holds the function's name.
- `__qualname__`: A writable attribute that holds the function's **qualified name**. The qualified name is a full dotted path to the object (with class names) in the global scope of the module where the object is defined.
- `__module__`: A writable attribute that holds the name of the module that function belongs to.
- `__defaults__`: A writable attribute that holds the default argument values if the function has any.
- `__code__`: A writable attribute that holds the function's compile code object.
- `__globals__`: A read-only attribute that holds the reference to the dictionary of global variables for that function's scope. The global scope for a function is the namespace of the module where this function is defined.

- `__dict__`: A writable attribute that holds a dictionary of function attributes. Functions in Python are first-class objects, so they can have any arbitrary arguments defined, just like any other object.
- `__closure__`: A read-only attribute that holds a tuple of cells with the function's free variables. Closure cells allow you to create parametrized function decorators.
- `__annotations__`: A writable attribute that holds the function's argument and return annotations.
- `__kwdefaults__`: A writable attribute that holds the default argument values for keyword-only arguments if the function has any.

Let's see how to reduce the boilerplate with data classes.

Reducing boilerplate with data classes

Before we dive deeper into details of Python classes, we will take a small detour. We will discuss a relatively new addition to the Python language, which are data classes. The `dataclasses` module, introduced in Python 3.7, provides a decorator and function that allows you to easily add generated special methods to your own classes.

Consider the following example. We are building a program that does some geometric computation and want to have 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 to achieve that goal. 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:

```
>>> 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 involves a lot of repetitive code that could be avoided. If your program uses many similar simple classes 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:

```
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 annotations of the `Vector` class attribute and automatically creates the `__init__()`, `__repr__()`, and `__eq__()` methods. The default equality comparison assumes that 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 sets. You can do this by simply adding a `frozen=True` argument to the `dataclass` decorator, as in the following example:

```
@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.

The final piece of useful information we will cover about data classes in this chapter is that you can define default values for specific attributes using the `field()` constructor. You can use both static values and constructors of other objects. Consider the following example:

```
>>> @dataclass
... class DataClassWithDefaults:
...     static_default: str = field(default="this is static default value")
...     factory_default: list = field(default_factory=list)
...
>>> DataClassWithDefaults()
DataClassWithDefaults(static_default='this is static default value',
factory_default=[])
```

The next section discusses subclassing built-in types.

Subclassing built-in types

Subclassing built-in types in Python is pretty straightforward. A built-in type, called `object` is a common ancestor for all built-in types, as well as for all user-defined classes that have no explicit parent class specified. Thanks to this, every time you need to implement a class that behaves almost like one of the built-in types, the best practice is to subtype it.

Now, we will look at the code for a class called `distinctdict`, which uses this technique. It will be a subclass of the usual Python `dict` type. This new class will behave, in most ways, like an ordinary Python `dict` type. But, instead of allowing multiple keys with the same value, when someone tries to add a new entry with an identical value, it raises a `ValueError` subclass with a help message.

As already stated, the built-in `dict` type is an object subclass:

```
>>> isinstance(dict(), object)
True
>>> issubclass(dict, object)
True
```

It means that we could easily define our own dictionary-based class as a direct subclass of that type, as follows:

```
class distinctdict(dict):
    ...
```

The previous approach would be totally valid, as a subclassing of `dict`, `list`, and `str` types has been allowed since Python 2.2. But, usually the better approach is to subclass one of the corresponding types from the collections module:

- `collections.UserDict`
- `collections.UserList`
- `collections.UserString`

These classes are usually easier to work with, as the underlying regular `dict`, `list`, and `str` objects are stored as data attributes of these classes.

The following is an example implementation of the `distinctdict` type that overrides part of the ordinary dictionary protocol to ensure that it contains only unique values:

```python
from collections import UserDict


class DistinctError(ValueError):
    """Raised when duplicate value is added to a distinctdict."""


class distinctdict(UserDict):
    """Dictionary that does not accept duplicate values."""
    def __setitem__(self, key, value):
        if value in self.values():
            if (
                (key in self and self[key] != value) or
                key not in self
            ):
                raise DistinctError(
                    "This value already exists for different key"
                )

        super().__setitem__(key, value)
```

The following is an example of using `distinctdict` in an interactive session:

```python
>>> my = distinctdict()
>>> my['key'] = 'value'
>>> my['other_key'] = 'value'
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 10, in __setitem__
DistinctError: This value already exists for different key
>>> my['other_key'] = 'value2'
>>> my
{'key': 'value', 'other_key': 'value2'}
>>> my.data
{'key': 'value', 'other_key': 'value2'}
```

If you take a look at your existing code, you may find a lot of classes that partially implement the protocols or functionalities of the built-in types. These classes could be faster and cleaner if implemented as subtypes of these types. The `list` type, for instance, manages the sequences of any type and you can use it every time your class works internally with a sequence or collection.

 The following is a simple example of the `Folder` class that subclasses the Python `list` type to represent and manipulate the contents of directories in a tree-like structure:

```
from collections import UserList


class Folder(UserList):
    def __init__(self, name):
        self.name = name

    def dir(self, nesting=0):
        offset = "  " * nesting
        print('%s%s/' % (offset, self.name))

        for element in self:
            if hasattr(element, 'dir'):
                element.dir(nesting + 1)
            else:
                print("%s  %s" % (offset, element))
```

Note that we have actually subclassed the `UserList` class from the `collections` module and not the bare `list` type. It is possible to subclass bare built-in types, such as `string`, `dict`, or `set`, but it is advisable to use their **user** counterparts from the `collections` module instead because they make subclassing a bit easier.

The following is an example use of our `Folder` class in an interactive session:

```
>>> tree = Folder('project')
>>> tree.append('README.md')
>>> tree.dir()
project/
  README.md
>>> src = Folder('src')
>>> src.append('script.py')
>>> tree.append(src)
>>> tree.dir()
project/
  README.md
  src/
    script.py
>>> tree.remove(src)
>>> tree.dir()
project/
  README.md
```

> **Built-in types cover most of the use cases**
>
> When you are about to create a new class that acts like a sequence or a mapping, think about its features and look over the existing built-in types. The `collections` module extends basic lists of built-in types with many useful containers. You will often end up using one of them without needing to create your custom subclasses.

Let's take a look at MRO in the next section.

MRO and accessing methods from superclasses

`super` is a built-in class that can be used to access an attribute belonging to an object's superclass.

> The Python official documentation lists `super` as a built-in function, but, it's a built-in class, even if it is used like a function:
> **>>> super**
> <class 'super'>
> **>>> isinstance(super, type)**

Its usage is a bit confusing if you are used to accessing a class attribute or method by calling the parent class directly and passing `self` as the first argument. This is a really old pattern, but still can be found in some code bases (especially in legacy projects). See the following code:

```
class Mama:  # this is the old way
    def says(self):
        print('do your homework')

class Sister(Mama):
    def says(self):
        Mama.says(self)
        print('and clean your bedroom')
```

Look particularly at the `Mama.says(self)` line. You can see here an explicit use of parent class. This means that the `says()` method belonging to `Mama` will be called. But, the instance on which it will be called is provided as the `self` argument, which is an instance of `Sister` in this case.

Instead, the `super` usage would be as follows:

```
class Sister(Mama):
    def says(self):
        super(Sister, self).says()
        print('and clean your bedroom')
```

Alternatively, you can also use the shorter form of the `super()` call:

```
class Sister(Mama):
    def says(self):
        super().says()
        print('and clean your bedroom')
```

The shorter form of `super` (without passing any arguments) is allowed inside the methods, but the usage of `super` is not limited to the body of methods. It can be used in any code area where the explicit call to the method of superclass implementation is required. Still, if `super` is not used inside the body of the method, then, all of its arguments are mandatory:

```
>>> anita = Sister()
>>> super(anita.__class__, anita).says()
do your homework
```

The final and most important thing that should be noted about super is that its second argument is optional. When only the first argument is provided, then `super` returns an unbounded type. This is especially useful when working with `classmethod`:

```
class Pizza:
    def __init__(self, toppings):
        self.toppings = toppings

    def __repr__(self):
        return "Pizza with " + " and ".join(self.toppings)

    @classmethod
    def recommend(cls):
        """Recommend some pizza with arbitrary toppings,"""
        return cls(['spam', 'ham', 'eggs'])
```

```
class VikingPizza(Pizza):
    @classmethod
    def recommend(cls):
        """Use same recommendation as super but add extra spam"""
        recommended = super(VikingPizza).recommend()
        recommended.toppings += ['spam'] * 5
        return recommended
```

Note that the zero-argument `super()` form is also allowed for methods decorated with the `classmethod` decorator. `super()`, if called without arguments in such methods, is treated as having only the first argument defined.

The use cases presented earlier are very simple to follow and understand, but when you face a multiple inheritance schema, it becomes hard to use `super`. Before explaining these problems, you need to first understand when `super` should be avoided and how the **Method Resolution Order** (**MRO**) works in Python.

Let's discuss old-style classes and super in Python 2.

Old-style classes and super in Python 2

`super()` in Python 2 works almost exactly the same as in Python 3. The only difference in its call signature is that the shorter, zero-argument form is not available, so at least one of the expected arguments must always be provided.

Another important thing for programmers to note who want to write cross-version compatible code is that `super` in Python 2 works only for new-style classes. The earlier versions of Python did not have a common ancestor for all classes in the form of an `object` type. The old behavior was left in every Python 2.x branch release for backward compatibility, so, in those versions, if the class definition has no ancestor specified, it is interpreted as an old-style class, and it cannot use `super`:

```
class OldStyle1:
    pass


class OldStyle2(OldStyle1):
    pass
```

The new-style class in Python 2 must explicitly inherit from the `object` type or other new-style class:

```
class NewStyleClass(object):
    pass


class NewStyleClassToo(NewStyleClass):
    pass
```

Python 3 no longer maintains the concept of old-style classes, so any class that does not inherit from any other class implicitly inherits from `object`. This means that explicitly stating that a class inherits from `object` may seem redundant. Standard good practice is to not include redundant code, but removing such redundancy in this case is a good approach only for projects that no longer target any of the Python 2 versions. Code that aims for cross-version compatibility of Python must always include `object` as an ancestor of base classes, even if this is redundant in Python 3. Not doing so will result in such classes being interpreted as old-style, and this will eventually lead to issues that are very hard to diagnose.

Let's understand Python's MRO in the next section

Understanding Python's Method Resolution Order

Python MRO is based on **C3**, the MRO built for the Dylan programming language (`http://opendylan.org`). The reference document, written by Michele Simionato, can be found at `http://www.python.org/download/releases/2.3/mro`. It describes how C3 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. The C3 algorithm is described in more detail later in this section.

The MRO change was made to resolve an issue introduced with the creation of a common base type (that is, `object` type). Before the change to the C3 linearization method, if a class had two ancestors (refer to *Figure 1*), the order in which methods were resolved was quite simple to compute and track only for simple cases that didn't use multiple inheritance model in a cascading way.

Here is an example of code, which, under Python 2, would not use C3 as an MRO:

```
class Base1:
    pass

class Base2:
    def method(self):
        print('Base2')

class MyClass(Base1, Base2):
    pass

>>> MyClass().method()
Base2
```

When `MyClass().method()` is called, the interpreter looks for the method in `MyClass`, then `Base1`, and then eventually finds it in `Base2`:
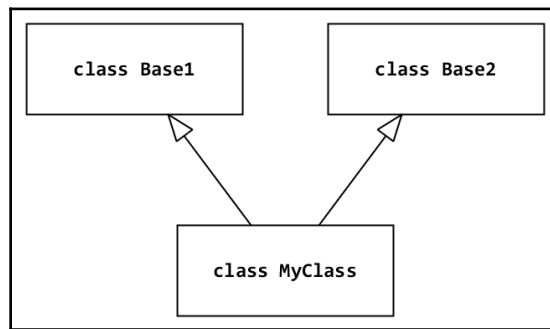
Figure 1: Classical hierarchy

When we introduce some `CommonBase` class at the top of our class hierarchy (both `Base1` and `Base2` will inherit from it, refer to *Figure 2*), things will get more complicated. 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. In some cases, the method that is executed may not be the one that is the closest in the inheritance tree.

Such an algorithm is still available in Python 2 for old-style classes. Here is an example of the old method resolution in Python 2 using old-style classes:

```
class CommonBase:
    def method(self):
        print('CommonBase')
```

```
class Base1(CommonBase):
    pass


class Base2(CommonBase):
    def method(self):
        print('Base2')


class MyClass(Base1, Base2):
    pass
```

The following transcript from the interactive session shows that `Base2.method()` will not be called despite the `Base2` class being closer in the class hierarchy to `MyClass` than `CommonBase`:

```
>>> MyClass().method()
CommonBase
```
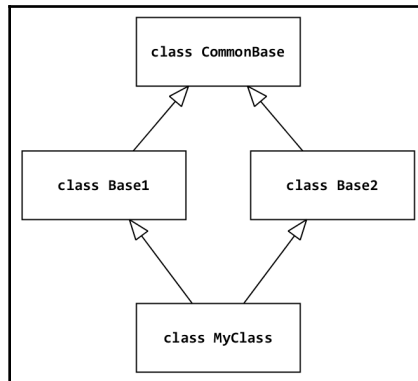

Figure 2: The Diamond class hierarchy

Such an inheritance scenario is extremely uncommon, so this is more a problem of theory than practice. The standard library does not structure the inheritance hierarchies in this way, and many developers think that it is bad practice. But, with the introduction of `object` at the top of the types hierarchy, the multiple inheritance problem pops up on the C side of the language, resulting in conflicts when doing subtyping. You should also note that every class in Python 3 has now got the same common ancestor. Since making it work properly with the existing MRO involved too much work, a new MRO was a simpler and quicker solution.

So, the same example run under Python 3 gives a different result:

```python
class CommonBase:
    def method(self):
        print('CommonBase')


class Base1(CommonBase):
    pass


class Base2(CommonBase):
    def method(self):
        print('Base2')


class MyClass(Base1, Base2):
    pass
```

And here is the usage example showing that C3 serialization will pick the method of the closest ancestor:

```python
>>> MyClass().method()
Base2
```

> **TIP**
>
> Note that the preceding behavior cannot be replicated in Python 2 without the `CommonBase` class explicitly inheriting from `object`. Reasons as to why it may be useful to specify `object` as a class ancestor in Python 3, even if this is redundant, were already mentioned in the previous section—*Old-style classes and super in Python 2*.

The Python MRO is based on a recursive call over the base classes. To summarize the Michele Simionato paper referenced at the beginning of this section, the C3 symbolic notation applied to our 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.

So, a synthetic description would be, as Simionato says:

> *"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 `merge` algorithm is responsible for removing the duplicates and preserving the correct ordering. It is described in the paper 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.*

The `head` is the first element of a list and the `tail` contains the rest of the elements. For example, in `(Base1, Base2, ..., BaseN)`, `Base1` is the `head`, and `(Base2, ..., BaseN)` is the `tail`.

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 a hierarchy disambiguation, when a class is involved in several lists.

So the result is as follows:

```
def L(klass):
    return [k.__name__ for k in klass.__mro__]


>>> L(MyClass)
['MyClass', 'Base1', 'Base2', 'CommonBase', 'object']
```

> **TIP**
>
> The `__mro__` attribute of a class (which is read-only) stores the result of the linearization computation. Computation is done when the class definition is loaded.
> You can also call `MyClass.mro()` to compute and get the result. This is another reason why classes in Python 2 should be taken with an extra case. While old-style classes in Python 2 have some defined order in which methods are resolved, they do not provide the `__mro__` attribute and the `mro()` method. So, despite the order of resolution, it is wrong to say that they have MRO. In most cases, whenever someone refers to MRO in Python, it means that they are referring to the C3 algorithm described in this section.

Let's now discuss some of the shortcomings faced by programmers.

Super pitfalls

Now, back to the `super()` call. If you deal with multiple inheritance hierarchy, it can become problematic. This is mainly due to the initialization of classes. In Python, the initialization methods (that is, the `__init__()` methods) of base classes are not implicitly called in ancestor classes if ancestor classes override `__init__()`. In such cases, you need to call superclass methods explicitly, and this can sometimes lead to initialization problems.

In this section, we will discuss a few examples of such problematic situations.

Mixing super and explicit class calls

In the following example, taken from James Knight's website (`http://fuhm.net/super-harmful`), a `C` class that calls initialization methods of its parent classes using the `super().__init__()` method will make the call to the `B.__init__()` class to be called twice:

```python
class A:
    def __init__(self):
        print("A", end=" ")
        super().__init__()


class B:
    def __init__(self):
        print("B", end=" ")
        super().__init__()


class C(A, B):
    def __init__(self):
        print("C", end=" ")
        A.__init__(self)
        B.__init__(self)
```

Here is the output:

```python
>>> print("MRO:", [x.__name__ for x in C.__mro__])
MRO: ['C', 'A', 'B', 'object']
>>> C()
C A B B <__main__.C object at 0x0000000001217C50>
```

In the preceding transcript we see that initialization of class C invokes the B.__init__() method twice. To avoid such issues, super should be used in the whole class hierarchy. The problem is that sometimes, a part of such complex hierarchy may be located in a third-party code. Many other related pitfalls on the hierarchy calls introduced by multiple inheritances can be found on James's page.

Unfortunately, you cannot be sure that external packages use super() in their code. Whenever you need to subclass some third-party class, it is always a good approach to take a look inside its code and the code of other classes in the MRO. This may be tedious, but, as a bonus, you get some information about the quality of code provided by such a package and more understanding of its code. You may learn something new that way.

Heterogeneous arguments

Another issue with super usage occurs if methods of classes within the class hierarchy use inconsistent argument sets. How can a class call its base class an __init__() code if it doesn't have the same signature? This leads to the following problem:

```
class CommonBase:
    def __init__(self):
        print('CommonBase')
        super().__init__()


class Base1(CommonBase):
    def __init__(self):
        print('Base1')
        super().__init__()


class Base2(CommonBase):
    def __init__(self, arg):
        print('base2')
        super().__init__()


class MyClass(Base1 , Base2):
    def __init__(self, arg):
        print('my base')
        super().__init__(arg)
```

An attempt to create a `MyClass` instance will raise `TypeError` due to a mismatch of the parent classes' `__init__()` signatures:

```
>>> MyClass(10)
my base
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in __init__
TypeError: __init__() takes 1 positional argument but 2 were given
```

One solution would be to use arguments and keyword arguments packing with `*args` and `**kwargs` magic so that all constructors pass along all the parameters, even if they do not use them:

```
class CommonBase:
    def __init__(self, *args, **kwargs):
        print('CommonBase')
        super().__init__()

class Base1(CommonBase):
    def __init__(self, *args, **kwargs):
        print('Base1')
        super().__init__(*args, **kwargs)

class Base2(CommonBase):
    def __init__(self, *args, **kwargs):
        print('base2')
        super().__init__(*args, **kwargs)

class MyClass(Base1 , Base2):
    def __init__(self, arg):
        print('my base')
        super().__init__(arg)
```

With this approach, the parent class signatures will always match:

```
>>> _ = MyClass(10)
my base
Base1
base2
CommonBase
```

This is an awful fix though, because it makes all constructors accept any kind of parameters. It leads to weak code, since anything can be passed and gone through. Another solution is to use the explicit `__init__()` calls of specific classes in `MyClass`, but this would lead to the first pitfall.

In the next section, we will discuss the best practices.

Best practices

To avoid all the aforementioned problems, and until Python evolves in this field, we need to take into consideration the following points:

- **Multiple inheritance should be avoided**: It can be replaced with some design patterns presented in `Chapter 17`, *Useful Design Patterns*.
    - **Super usage has to be consistent**: In a class hierarchy, `super` should be used everywhere or nowhere. Mixing `super` and classic calls is a confusing practice. People tend to avoid `super` to render their code more explicit.
- **Explicitly inherit from an object in Python 3 if you target Python 2 too**: Classes without any ancestor specified are recognized as old-style classes in Python 2. Mixing old-style classes with new-style classes should be avoided in Python 2.
- **Class hierarchy has to be looked over when a parent class method is called**: To avoid any problems, every time a parent class method is called, a quick glance at the MRO involved (with `__mro__`) is necessary.

Let's take a look at the access patterns for advanced attributes.

Advanced attribute access patterns

When many C++ and Java programmers first learn Python, they are surprised by Python's lack of a `private` keyword. The nearest concept is **name mangling**. Every time an attribute is prefixed by __, it is renamed by the interpreter on the fly:

```
class MyClass:
    __secret_value = 1
```

Accessing the `__secret_value` attribute by its initial name will raise an `AttributeError` exception:

```
>>> 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'
>>> dir(MyClass)
['_MyClass__secret_value', '__class__', '__delattr__', '__dict__',
```

```
    '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
    '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__',
    '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
    '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__']
>>> instance_of._MyClass__secret_value
1
```

This feature is provided to avoid name collision under inheritance, as the attribute is renamed with the class name as a prefix. It is not a real privacy lock, since the attribute can be accessed through its composed name. This feature could be used to protect the access of some attributes, but, in practice, __ should never be used. When an attribute is not public, the convention to use is a _ prefix. This does not invoke any name mangling algorithm, but just documents the attribute as a private element of the class and is the prevailing style.

Other mechanisms are available in Python that nicely separate the public part of the class together with the private code. The descriptors and properties are interesting features of Python that allow us to cleanly provide this kind of separation.

Descriptors

A descriptor lets you customize what should be done when you refer to an attribute of an object.

Descriptors are the base of a complex attribute access in Python. They are used internally to implement properties, methods, class methods, static methods, and the super type. They are classes that define how attributes of another class can be accessed. In other words, a class can delegate the management of an attribute to another one.

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, I 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 this protocol are, in fact, called by the object's
special __getattribute__() method (do not confuse it with __getattr__(), which has
a different purpose) on every attribute lookup. 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 __dict__ lookup,
and __dict__ lookup takes precedence over non-data descriptors.

To make it clearer, here is an example from the official Python documentation that shows
how descriptors work on real code:

```
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


class MyClass(object):
    x = RevealAccess(10, 'var "x"')
    y = 5
```

Here is an example of using it in the interactive session:

```
>>> m = MyClass()
>>> m.x
Retrieving var "x"
10
>>> m.x = 20
Updating var "x"
>>> m.x
Retrieving var "x"
20
>>> m.y
5
```

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. Although the case for the descriptor's __del__ method is not shown in the preceding example, it should be obvious now: it 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. They 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 without the need for subclassing.

Real-life example – lazily evaluated attributes

One example usage of descriptors may be to delay 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 the global application context. The other case is when such initialization is simply expensive, but it is not known whether it will be used anyway when the class is imported. Such a descriptor could be implemented as follows:

```python
class InitOnAccess:
    def __init__(self, klass, *args, **kwargs):
        self.klass = klass
        self.args = args
        self.kwargs = kwargs
        self._initialized = None

    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
```

Here is an example usage:

```python
>>> class MyClass:
...     lazily_initialized = InitOnAccess(list, "argument")
...
>>> m = MyClass()
>>> m.lazily_initialized
initialized!
['a', 'r', 'g', 'u', 'm', 'e', 'n', 't']
>>> m.lazily_initialized
cached!
['a', 'r', 'g', 'u', 'm', 'e', 'n', 't']
```

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:

```
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
```

Such an implementation is similar to using the `property` decorator (described later), but the function that is wrapped with it is executed only once and then the class attribute is replaced with a value returned by that function property. That technique is often useful when there's a need to fulfil 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:

```
import OpenGL.GL as gl
from OpenGL.GL import shaders


class lazy_class_attribute(object):
    def __init__(self, function):
        self.fget = function
```

```python
    def __get__(self, obj, cls):
        value = self.fget(obj or 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. For inexperienced developers, the altered class behavior might be very confusing and unexpected, because descriptors affect the very basic part of class behavior. 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

The properties provide a built-in descriptor type that knows how to link an attribute to a set of methods. `property` 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:

```
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 in module chapter3:
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's class (`Rectangle`) `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
```

Slots

An interesting feature that is very rarely used by developers is slots. They allow you to set a static attribute list for a given class with the `__slots__` attribute, and skip the creation of the `__dict__` dictionary in each instance of the class. They were intended to save memory space for classes with very few attributes, since `__dict__` is not created at every instance.

Besides this, they can help to design classes whose signature needs to be frozen. For instance, if you need to restrict the dynamic features of the language over a class, defining slots can help:

```
>>> class Frozen:
...     __slots__ = ['ice', 'cream']
...
>>> '__dict__' in dir(Frozen)
False
>>> 'ice' in dir(Frozen)
True
>>> frozen = Frozen()
>>> frozen.ice = True
>>> frozen.cream = None
>>> frozen.icy = True
Traceback (most recent call last): File "<input>", line 1, in <module>
AttributeError: 'Frozen' object has no attribute 'icy'
```

This feature should be used carefully. When a set of available attributes is limited using `__slots__`, it is much harder to add something to the object dynamically. Some techniques, such as monkey patching, will not work with instances of classes that have slots defined. Fortunately, the new attributes can be added to the derived classes if they do not have their own slots defined:

```
>>> class Unfrozen(Frozen):
...     pass
...
>>> unfrozen = Unfrozen()
>>> unfrozen.icy = False
>>> unfrozen.icy
False
```

Summary

In this chapter, we discussed modern Python syntax elements related to class models and object-oriented programming.

We started with an explanation of the language protocol concept and simple ways to implement those protocols. We discussed the subclassing of built-in types and how to call methods from superclasses. After that, we moved on to more advanced concepts of object-oriented programming in Python. These were useful syntax features that focus on instance attribute access: descriptors and properties. We demonstrated how they can be used to create cleaner and more maintainable code.

In the next chapter, we will explore the vast topic of metaprogramming in Python. We will reuse some of the syntax features that we've learned so far to show various metaprogramming techniques.

5
Elements of Metaprogramming

Metaprogramming is one of the most complex and powerful approaches to programming in Python. Metaprogramming tools and techniques have evolved with Python; so, before we dive into this topic, it is important for you to know all the elements of modern Python syntax well. We discussed them in the two previous chapters. If you read them carefully, then you should know enough to fully understand the contents of this chapter.

In this chapter, we will explain what metaprogramming really is and present a few practical approaches to metaprogramming in Python.

In this chapter, we will cover the following topics:

- What is metaprogramming?
- Decorators
- Metaclasses
- Code generation

Technical requirements

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

- `macropy`
- `falcon`
- `hy`

You can install these packages using the following command:

```
python3 -m pip install <package-name>
```

The code files for this chapter can be found at `https://github.com/PacktPublishing/Expert-Python-Programming-Third-Edition/tree/master/chapter5`.

What is metaprogramming?

Maybe there is a good academic definition of metaprogramming that we can cite here, but this is a book that is more about good software craftsmanship than about computer science theory. This is why we will use the following simple definition:

> *"Metaprogramming is a technique of writing computer programs that can treat themselves as data, so they can introspect, generate, and/or modify itself while running."*

Using this definition, we can distinguish between two major approaches to metaprogramming in Python.

The first approach 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 fly. Python really provides a lot of tools in this area. This feature of the Python language is used by IDEs (such as PyCharm) to provide real-time code analysis and name suggestions. The easiest possible metaprogramming tools in Python that utilized language introspection are decorators that allow for adding extra functionality to the existing functions, methods, or classes. Next are special methods of classes that allow you to interfere with class instance process creation. The most powerful are metaclasses, which allow programmers to even completely redesign Python's implementation of object-oriented programming.

The second approach allows programmers to work directly with code, either in its raw (plain text) format or in 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.

Decorators – a method of metaprogramming

The decorator syntax was explained in `Chapter 3`, *Modern Syntax Elements – Below the Class Level*, as a syntactic sugar for the following simple pattern:

```python
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. As a result, a new function (or anything else) is created based on the previous function object with the same name. This decoration may be a complex operation that performs some code introspection or decorated function to give different results depending on how the original function was implemented. All this means is that decorators can be considered as a metaprogramming tool.

This is good news. The basics of decorators are relatively easy to grasp and in most cases make code shorter, easier to read, and also cheaper to maintain. Other metaprogramming tools that are available in Python are more difficult to understand and master. Also, they might not make the code simple at all.

We'll take a look at class decorators in the next section.

Class decorators

One of the lesser known syntax features of Python are the class decorators. Their syntax and implementation is exactly the same as function decorators, as we mentioned in Chapter 3, *Modern Syntax Elements – Below the Class Level*. The only difference is that they are expected to return a class instead of the function object. Here is an example class decorator that modifies the `__repr__()` method to return the printable object representation, which is shortened to some arbitrary number of characters:

```
def short_repr(cls):
    cls.__repr__ = lambda self: super(cls, self).__repr__()[:8]
    return cls

@short_repr
class ClassWithRelativelyLongName:
    pass
```

The following is what you will see in the output:

```
>>> ClassWithRelativelyLongName()
<ClassWi
```

Of course, the preceding snippet is not an example of good code by any means. Still, it shows how multiple language features that are explained in the previous chapter can be used together, for example:

- Not only instances but also class objects can be modified at runtime

- Functions are descriptors too, so they can be added to the class at runtime because the actual method binding is performed on the attribute lookup as part of the descriptor protocol
- The `super()` call can be used outside of a class definition scope as long as proper arguments are provided
- Finally, class decorators can be used on class definitions

The other aspects of writing function decorators apply to the class decorators as well. Most importantly, they can use closures and be parametrized. Taking advantage of these facts, the previous example can be rewritten into the following more readable and maintainable form:

```
def parametrized_short_repr(max_width=8):
    """Parametrized decorator that shortens representation"""
    def parametrized(cls):
        """Inner wrapper function that is actual decorator"""
        class ShortlyRepresented(cls):
            """Subclass that provides decorated behavior"""
            def __repr__(self):
                return super().__repr__()[:max_width]

        return ShortlyRepresented

    return parametrized
```

The major drawback of using closures in class decorators this way is that the resulting objects are no longer instances of the class that was decorated but instances of the subclass that was created dynamically in the decorator function. Among others, this will affect the class's __name__ and __doc__ attributes, as follows:

```
@parametrized_short_repr(10)
class ClassWithLittleBitLongerLongName:
    pass
```

Such usage of class decorators will result in the following changes to the class metadata:

```
>>> ClassWithLittleBitLongerLongName().__class__
<class 'ShortlyRepresented'>
>>> ClassWithLittleBitLongerLongName().__doc__
'Subclass that provides decorated behavior'
```

Unfortunately, this cannot be fixed as simply as we explained in the *Introspection Preserving Decorators* section of `Chapter 3`, *Modern Syntax Elements – Below the Class Level.* In class decorators, you can't simply use the additional `wraps` decorator to preserve the original class type and metadata. This makes use of the class decorators in this form limited in some circumstances. They can, for instance, break 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. 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:

```
class SomeConcreteClass(MixinClass, SomeBaseClass):
    pass
```

Mixins 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, mixins 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. Multiple inheritance makes code more complex and hard to reason about. This is why class decorators may be a good replacement for mixin classes.

Let's take a look at the use of __new__() to override the instance creation process.

Using __new__() for overriding the instance creation process

The special method __new__() is a static method that's responsible for creating class instances. It is special-cased, so there is no need to declare it as static using the `staticmethod` decorator. This __new__(cls, [,...]) method is called prior to the __init__() initialization method. Typically, the implementation of overridden __new__() invokes its superclass version using `super().__new__()` with suitable arguments and modifies the instance before returning it.

The following is an example class with the overridden __new__() method implementation in order to count the number of class instances:

```
class InstanceCountingClass:
    instances_created = 0
    def __new__(cls, *args, **kwargs):
        print('__new__() called with:', cls, args, kwargs)
        instance = super().__new__(cls)
        instance.number = cls.instances_created
        cls.instances_created += 1

        return instance

    def __init__(self, attribute):
        print('__init__() called with:', self, attribute)
        self.attribute = attribute
```

Here is the log of the example interactive session that shows how our InstanceCountingClass implementation works:

```
>>> from instance_counting import InstanceCountingClass
>>> instance1 = InstanceCountingClass('abc')
__new__() called with: <class '__main__.InstanceCountingClass'> ('abc',) {}
__init__() called with: <__main__.InstanceCountingClass object at
0x101259e10> abc
>>> instance2 = InstanceCountingClass('xyz')
__new__() called with: <class '__main__.InstanceCountingClass'> ('xyz',) {}
__init__() called with: <__main__.InstanceCountingClass object at
0x101259dd8> xyz
>>> instance1.number, instance1.instances_created
(0, 2)
>>> instance2.number, instance2.instances_created
(1, 2)
```

The __new__() method should usually return an 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 creation/initialization behavior of immutable class instances like some of Python's built-in types, as shown in the following code:

```
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__()
```

Let's review these in the following interactive session:

```
>>> 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 can 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 programmers' expectations. Even overridden initialization of immutable types can be replaced with more predictable and well-established design patterns like the Factory Method, which is described in Chapter 17, *Useful Design Patterns*.

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

Metaclass is a Python feature that is considered by many as one of the most difficult things to understand in this language and 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.

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 that define object instances 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. Here is a simple diagram that should make this clear:
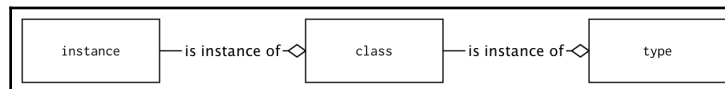
Figure 1: How classes are typed

In Python, it is possible to substitute the metaclass for a class object with our own type. Usually, the new metaclass is still the subclass of the `type` class (refer to *Figure 2*) because not doing so would make the resulting classes highly incompatible with other classes in terms of inheritance:
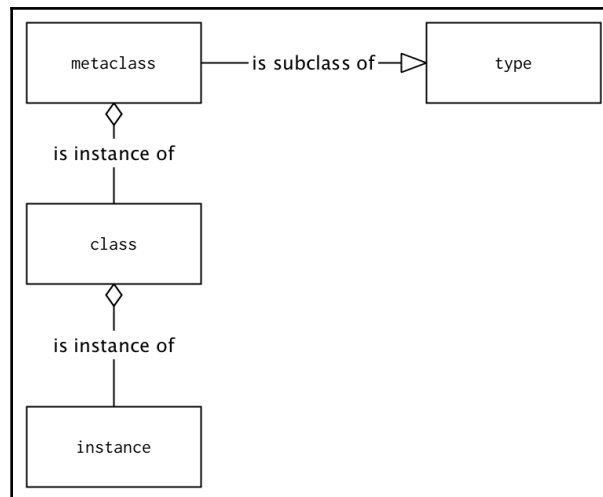
Figure 2: Usual implementation of custom metaclasses

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:

```
def method(self):
    return 1

MyClass = type('MyClass', (object,), {'method': method})
```

This is equivalent to the explicit definition of the class with the `class` keyword:

```
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:

```
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 call signature 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:

```
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 can have the following different purposes:

- `__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 a `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`, but it can be overridden to return any other mapping type. Note that it does not accept `namespace` as an argument because, before calling it, the namespace does not exist. Example usage of that method will be explained later in the *New Python 3 syntax for metaclasses* section.
- `__init__(cls, name, bases, namespace, **kwargs)`: This is not seen popularly 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__()` was called, the class was already constructed and so this method can do less things 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 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 the beginning 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, "__new__ called")
        return super().__new__(mcs, name, bases, namespace)

    @classmethod
    def __prepare__(mcs, name, bases, **kwargs):
        print(mcs, "__prepare__ called")
        return super().__prepare__(name, bases, **kwargs)

    def __init__(cls, name, bases, namespace, **kwargs):
        print(cls, "__init__ called")
        super().__init__(name, bases, namespace)

    def __call__(cls, *args, **kwargs):
        print(cls, "__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 'RevealingMeta'> __prepare__ called
<class 'RevealingMeta'> __new__ called
```

```
<class 'RevealingClass'> __init__ called
>>> instance = RevealingClass()
<class 'RevealingClass'> __call__ called <class 'RevealingClass'> __new__
called <RevealingClass object at 0x1032b9fd0> __init__ called
```

Let's take a look at the new Python 3 syntax for metaclasses.

New Python 3 syntax for metaclasses

Metaclasses are not a new feature and have been available in Python since version 2.2. Anyway, the syntax of this changed significantly and this change is neither backward nor forward compatible. The new syntax is as follows:

```
class ClassWithAMetaclass(metaclass=type):
    pass
```

In Python 2, this must be written as follows:

```
class ClassWithAMetaclass(object):
    __metaclass__ = type
```

Class statements in Python 2 do not accept keyword arguments, so Python 3 syntax for defining `metaclasses` will raise the `SyntaxError` exception on import. It is still possible to write code using metaclasses that will run on both Python versions, but it requires some extra work. Fortunately, compatibility-related packages such as `six` provide simple and reusable solutions to this problem, such as the one shown in the following code:

```
from six import with_metaclass


class Meta(type):
    pass


class Base(object):
    pass


class MyClass(with_metaclass(Meta, Base)):
    pass
```

The other important difference is the lack of the `__prepare__()` hook in Python 2 metaclasses. Implementing such a function will not raise any exceptions under Python 2, but this is pointless because it will not be called in order to provide a clean namespace object. This is why packages that need to maintain Python 2 compatibility need to rely on more complex tricks if they want to achieve things that are a lot easier to implement using `__prepare__()`. For instance, the Django REST Framework (`http://www.django-rest-framework.org`) in version 3.4.7 uses the following approach to preserve the order in which attributes are added to a class:

```python
class SerializerMetaclass(type):
    @classmethod
    def _get_declared_fields(cls, bases, attrs):
        fields = [(field_name, attrs.pop(field_name))
                  for field_name, obj in list(attrs.items())
                  if isinstance(obj, Field)]
        fields.sort(key=lambda x: x[1]._creation_counter)

        # If this class is subclassing another Serializer, add
        # that Serializer's fields.
        # Note that we loop over the bases in *reverse*.
        # This is necessary in order to maintain the
        # correct order of fields.
        for base in reversed(bases):
            if hasattr(base, '_declared_fields'):
                fields = list(base._declared_fields.items()) + fields

        return OrderedDict(fields)

    def __new__(cls, name, bases, attrs):
        attrs['_declared_fields'] = cls._get_declared_fields(
            bases, attrs
        )
        return super(SerializerMetaclass, cls).__new__(
            cls, name, bases, attrs
        )
```

This is the workaround for the fact that the default namespace type, which is `dict`, does not guarantee to preserve the order of the key-value tuples in Python versions older than 3.7 (see the *Dictionaries* section of `Chapter 3`, *Modern Syntax Elements – Below the Class Level*). The `_creation_counter` attribute is expected to be in every instance of the `Field` class. This `Field.creation_counter` attribute is created in the same way as `InstanceCountingClass.instance_number` which was presented in the *Using __new__() for overriding the instance creation process* section. This is a rather complex solution that breaks a single responsibility principle by sharing its implementation across two different classes only to ensure a trackable order of attributes. In Python 3, this could be simpler because `__prepare__()` can return other mapping types, such as `OrderedDict`, as shown in the following code:

```
from collections import OrderedDict


class OrderedMeta(type):
    @classmethod
    def __prepare__(cls, name, bases, **kwargs):
        return OrderedDict()
    def __new__(mcs, name, bases, namespace):
        namespace['order_of_attributes'] = list(namespace.keys())
        return super().__new__(mcs, name, bases, namespace)


class ClassWithOrder(metaclass=OrderedMeta):
    first = 8
    second = 2
```

If you inspect `ClassWithOrder` in an interactive session, you'll see the following output:

```
>>> ClassWithOrder.order_of_attributes
['__module__', '__qualname__', 'first', 'second']
>>> ClassWithOrder.__dict__.keys()
dict_keys(['__dict__', 'first', '__weakref__', 'second',
'order_of_attributes', '__module__', '__doc__'])
```

The uses of metaclasses are given in the next section.

Metaclass usage

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.

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.

But there are situations where things cannot be easily done without them. 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.

Let's take a look at some of the limitations of metaclasses.

Metaclass pitfalls

Like some other advanced Python features, the 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:

```
>>> 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:

```
>>> Nonsense()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File
"/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/unittest/m
ock.py", line 917, in __call__
    return _mock_self._mock_call(*args, **kwargs)
  File
"/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/unittest/m
ock.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.

We'll discuss code generation in the next section.

Code generation

As we already mentioned, the 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 do some modifications to the 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 deeper by yourself. So, treat this section only as a short summary of possible starting points for further learning.

Let's take a look at the 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 the Python code. `object` should be a string or code object (see the `compile()` function) representing a single statement or 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` might be any mapping object; it 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 `expression` 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 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. Mode should be either `exec` (sequence of statements), `eval` (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 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()` can cost you your image as a professional developer, or even your job.

Even if used with a 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.

> **eval() and untrusted input**
>
> The signature of the `eval()` function might make you think that if you
> provide empty `globals` and `locals` namespaces and wrap it with
> proper `try ... except` statements, then it will be reasonably safe. There
> could be nothing more wrong. Ned Batcheler 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_dangerou`
> `s.html`). This is single proof that both `exec()` and `eval()` should never
> be used with untrusted input.

We'll take a look at abstract syntax tree in the next section.

Abstract syntax tree (AST)

The Python syntax is converted into **AST** 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:

```
>>> tree = ast.parse('def hello_world(): print("hello world!")')
>>> tree
<_ast.Module object at 0x00000000038E9588>
>>> ast.dump(tree)
"Module(
    body=[
```

```
         FunctionDef(
            name='hello_world',
            args=arguments(
                args=[],
                vararg=None,
                kwonlyargs=[],
                kw_defaults=[],
                kwarg=None,
                defaults=[]
            ),
            body=[
                Expr(
                    value=Call(
                        func=Name(id='print', ctx=Load()),
                        args=[Str(s='hello world!')],
                        keywords=[]
                    )
                )
            ],
            decorator_list=[],
            returns=None
         )
      ]
   )"
```

The output of `ast.dump()` in the preceding example was reformatted to increase the readability and better show the tree-like structure of the AST. 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 3*):
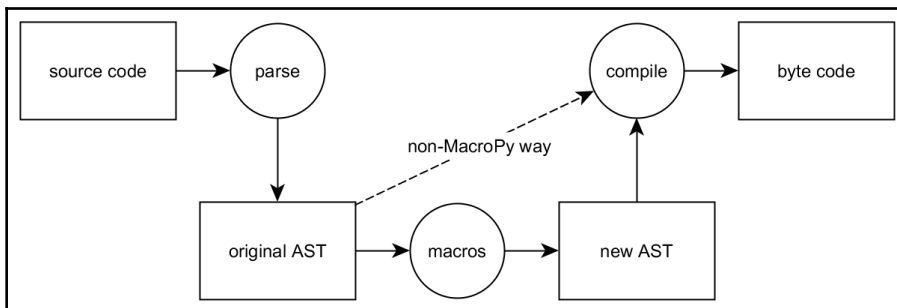
Figure 3: How MacroPy adds syntactic macros to Python modules on import

AST can also be created in a purely artificial manner, and there is no need to parse any source at all. This gives Python programmers the ability to create Python bytecode for custom domain-specific languages, or even completely implement other programming languages on top of Python VMs.

Import hooks

Taking advantage of MacroPy's ability to modify original ASTs would be as easy as using the `import macropy.activate` statement if it could somehow override the Python import behavior. Fortunately, Python provides a way to intercept imports 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 finder** object.

The details of implementing both path finders and meta path finders are extensively implemented 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 import machinery in Python is rather complex and any attempt to summarize it in a few paragraphs would inevitably fail. Here, we just noted that such things are possible.

We'll take a look at projects that use code generation patterns in the following sections.

Projects that use code generation patterns

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 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 to other micro-frameworks that aim for simplicity, such as Flask, Bottle, or web2py.

One of its features is it's 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 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 highlighted call to the `api.add_route()` method updates dynamically 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, and is written entirely in Python. Many similar projects that implement other code 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 the 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 byte code 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
```

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.

The next chapter will offer you a moment of rest as we focus on the topic of good naming practices.

6
Choosing Good Names

Most of the standard library was built keeping usability in mind. Python, in this case, can be compared to the pseudocode you might think about when working on a program. Most of the code can be read out loud. For instance, this snippet could be understood even by someone that is not a programmer:

```
my_list = []
if 'd' not in my_list:
    my_list.append('d')
```

The fact that Python code is so close to natural language is one of the reasons why Python is so easy to learn and use. When you are writing a program, the flow of your thoughts is quickly translated into lines of code.

This chapter focuses on the best practices to write code that is easy to understand and use, including:

- The usage of naming conventions, described in PEP 8
- The setting of naming best practices
- A short summary of popular tools that allow you to check for compliance with styling guides

In this chapter, we will cover the following topics:

- PEP 8 and naming best practices
- Naming styles
- The naming guide
- Best practices for arguments
- Class names
- Module and package names
- Useful tools

Technical requirements

Following are Python packages mentioned in this chapter that you can download from PyPI:

- `pylint`
- `pycodestyle`
- `flake8`

You can install these packages using following command:

```
python3 -m pip install <package-name>
```

Code files for this chapter can be found at `https://github.com/PacktPublishing/Expert-Python-Programming-Third-Edition/tree/master/chapter6`.

PEP 8 and naming best practices

PEP 8 (`http://www.python.org/dev/peps/pep-0008`) provides a style guide for writing Python code. Besides some basic rules, such as indentation, maximum line length, and other details concerning the code layout, PEP 8 also provides a section on naming conventions that most of the code bases follow.

This section provides only a quick summary of PEP 8, and a handy naming guide for each kind of Python syntax element. You should still consider reading the PEP 8 document as mandatory.

Why and when to follow PEP 8?

If you are creating a new software package that is intended to be open sourced, you should always follow PEP 8 because it is a widely accepted standard and is used in most of the open source projects written in Python. If you want to foster any collaboration with other programmers, then you should definitely stick to PEP 8, even if you have different views on the best code style guidelines. Doing so has the benefit of making it a lot easier for other developers to jump straight into your project. Code will be easier to read for newcomers because it will be consistent in style with most of the other Python open source packages.

Also, starting with full PEP 8 compliance saves you time and trouble in the future. If you want to release your code to the public, you will eventually face suggestions from fellow programmers to switch to PEP 8. Arguments as to whether it is really necessary to do so for a particular project tend to be never-ending flame wars that are impossible to win. This is the sad truth, but you may be eventually forced to be consistent with it or risk losing valuable contributors.

Also, restyling of the whole project's code base if it is in a mature state of development might require a tremendous amount of work. In some cases, such restyling might require changing almost every line of code. While most of the changes can be automated (indentation, newlines, and trailing whitespaces), such massive code overhaul usually introduces a lot of conflicts in every version control workflow that is based on branching. It is also very hard to review so many changes at once. These are the reasons why many open source projects have a rule that style-fixing changes should always be included in separate pull/merge requests or patches that do not affect any feature or bug.

Beyond PEP 8 – Team-specific style guidelines

Despite providing a comprehensive set of style guidelines, PEP 8 still leaves some freedom for the developers. Especially in terms of nested data literals and multiline function calls that require long lists of arguments. Some teams may decide that they require additional styling rules and the best option is to formalize them in some kind of document that is available for every team member.

Also, in some situations, it may be impossible or economically infeasible to be strictly consistent with PEP 8 in some old projects that had no style guide defined. Such projects will still benefit from formalization of the actual coding conventions even if they do not reflect the official set of PEP 8 rules. Remember, what is more important than consistency with PEP 8 is consistency within the project. If rules are formalized and available as a reference for every programmer, then it is way easier to keep consistency within a project and organization.

Let's take a look at the different naming styles in the next section.

Naming styles

The different naming styles used in Python are:

- `CamelCase`
- `mixedCase`
- `UPPERCASE` and `UPPER_CASE_WITH_UNDERSCORES`
- `lowercase` and `lower_case_with_underscores`
- `_leading` and `trailing_` underscores, and sometimes `__doubled__` underscores

Lowercase and uppercase elements are often a single word, and sometimes a few words concatenated. With underscores, they are usually abbreviated phrases. Using a single word is better. The leading and trailing underscores are used to mark the privacy and special elements.

These styles are applied to the following:

- Variables
- Functions and methods
- Properties
- Classes
- Modules
- Packages

Variables

There are the following two kinds of variables in Python:

- **Constants:** These define values that are not supposed to change during program execution
- **Public and private variables:** These hold the state of applications that can change during program execution

Constants

For constant global variables, an uppercase with an underscore is used. It informs the developer that the given variable represents a constant value.

There are no real constants in Python like those in C++, where `const` can be used. You can change the value of any variable. That's why Python uses a naming convention to mark a variable as a constant.

For example, the `doctest` module provides a list of option flags and directives (`http://docs.python.org/lib/doctest-options.html`) that are small sentences, clearly defining what each option is intended for, for example:

```
from doctest import IGNORE_EXCEPTION_DETAIL
from doctest import REPORT_ONLY_FIRST_FAILURE
```

These variable names seem rather long, but it is important to clearly describe them. Their usage is mostly located in the initialization code rather than in the body of the code itself, so this verbosity is not annoying.

Abbreviated names obfuscate the code most of the time. Don't be afraid of using complete words when an abbreviation seems unclear.

Some constants' names are also driven by the underlying technology. For instance, the `os` module uses some constants that are defined on the C side, such as the `EX_XXX` series, that defines UNIX exit code numbers. Same name code can be found, as in the following example, in the system's `sysexits.h` C headers files:

```
import os
import sys
sys.exit(os.EX_SOFTWARE)
```

Another good practice when using constants is to gather all of them at the top of a module that uses them. It is also common to combine them under new variables if they are flags or enumerations that allow for such operations, for example:

```
import doctest
TEST_OPTIONS = (doctest.ELLIPSIS |
                doctest.NORMALIZE_WHITESPACE |
                doctest.REPORT_ONLY_FIRST_FAILURE)
```

Let's take a look at the naming and usage of constants in the next section.

Naming and usage

Constants are used to define a set of values the program relies on, such as the default configuration filename.

A good practice is to gather all the constants in a single file in the package. That is how Django works, for instance. A module named `settings.py` provides all the constants as follows:

```
# config.py
SQL_USER = 'tarek'
SQL_PASSWORD = 'secret'
SQL_URI = 'postgres://%s:%s@localhost/db' % (
    SQL_USER, SQL_PASSWORD
)
MAX_THREADS = 4
```

Another approach is to use a configuration file that can be parsed with the `ConfigParser` module, or another configuration parsing tool. But some people argue that it is rather an overkill to use another file format in a language such as Python, where a source file can be edited and changed as easily as a text file.

For options that act like flags, a common practice is to combine them with Boolean operations, as the `doctest` and `re` modules do. The pattern taken from `doctest` is quite simple, as shown in the following code:

```
OPTIONS = {}

def register_option(name):
    return OPTIONS.setdefault(name, 1 << len(OPTIONS))

def has_option(options, name):
 return bool(options & name)

# now defining options
BLUE = register_option('BLUE')
RED = register_option('RED')
WHITE = register_option('WHITE')
```

This code allows for the following usage:

```
>>> # let's try them
>>> SET = BLUE | RED
>>> has_option(SET, BLUE)
True
>>> has_option(SET, WHITE)
False
```

When you define a new set of constants, avoid using a common prefix for them, unless the module has several independent sets of options. The module name itself is a common prefix.

Another good solution for option-like constants would be to use the `Enum` class from the built-in `enum` module and simply rely on the `set` collection instead of the binary operators. Details of the `enum` module usage and syntax were explained in the *Symbolic enumeration with enum module* section of `Chapter 3`, *Modern Syntax Elements - Below the Class Level.*

> Using binary bit-wise operations to combine options is common in Python. The inclusive OR (`|`) operator will let you combine several options in a single integer, and the AND (`&`) operator will let you check that the option is present in the integer (refer to the `has_option` function).

Let's discuss public and private variables in the following section.

Public and private variables

For global variables that are mutable and freely available through imports, a lowercase letter with an underscore should be used when they do not need to be protected. If a variable shouldn't be used and modified outside of its origin module we consider it a private member of that module. A leading underscore, in that case, can mark the variable as a private element of the package, as shown in the following code:

```
_observers = []
def add_observer(observer):
    _observers.append(observer)
def get_observers():
    """Makes sure _observers cannot be modified."""
    return tuple(_observers)
```

Variables that are located in functions, and methods, follow the same rules as public variables and are never marked as private since they are local to the function context.

For class or instance variables, you should use the private marker (the leading underscore) if making the variable a part of the public signature does not bring any useful information, or is redundant. In other words, if the variable is used only internally for the purpose of some other method that provides an actual public feature, it is better to make it private.

For instance, the attributes that are powering a property are good private citizens, as shown in the following code:

```
class Citizen(object):
    def __init__(self, first_name, last_name):
        self._first_name = first_name
        self._last_name = last_name

    @property
    def full_name(self):
        return f"{self._first_name} {self._last_name}"
```

Another example would be a variable that keeps some internal state that should not be disclosed to other classes. This value is not useful for the rest of the code, but participates in the behavior of the class:

```
class UnforgivingElephant(object):
    def __init__(self, name):
        self.name = name
        self._people_to_stomp_on = []

    def get_slapped_by(self, name):
        self._people_to_stomp_on.append(name)
        print('Ouch!')

    def revenge(self):
        print('10 years later...')
        for person in self._people_to_stomp_on:
            print('%s stomps on %s' % (self.name, person))
```

Here is what you'll see in an interactive session:

```
>>> joe = UnforgivingElephant('Joe')
>>> joe.get_slapped_by('Tarek')
Ouch!
>>> joe.get_slapped_by('Bill')
Ouch!
>>> joe.revenge()
10 years later...
Joe stomps on Tarek
Joe stomps on Bill
```

Let's take a look at naming styles for functions and methods in the next section.

Functions and methods

Functions and methods should be in lowercase with underscores. This rule was not always true in the old standard library modules. Python 3 did a lot of reorganization of the standard library, so most of the functions and methods have a consistent letter case. Still, for some modules such as `threading`, you can access the old function names that used `mixedCase` (for example, `currentThread`). This was left to allow easier backward compatibility, but if you don't need to run your code in older versions of Python, then you should avoid using these old names.

This way of writing methods was common before the lowercase norm became the standard, and some frameworks, such as Zope and Twisted, are also still using *mixedCase* for methods. The community of developers working with them is still quite large. So the choice between `mixedCase` and lowercase with an underscore is definitely driven by the libraries you are using.

As a Zope developer, it is not easy to stay consistent because building an application that mixes pure Python modules and modules that import Zope code is difficult. In Zope, some classes mix both conventions because the code base is still evolving and Zope developers try to adopt the common conventions accepted by so many.

A decent practice in this kind of library environment is to use `mixedCase` only for elements that are exposed in the framework, and to keep the rest of the code in PEP 8 style.

It is also worth noting that developers of the Twisted project took a completely different approach to this problem. The Twisted project, same as Zope, predates the PEP 8 document. It was started when there were no official guidelines for Python code style, so it had its own guidelines. Stylistic rules about the indentation, docstrings, line lengths, and so on could be easily adopted. On the other hand, updating all the code to match naming conventions from PEP 8 would result in completely broken backward compatibility. And doing that for such a large project as Twisted is infeasible. So Twisted adopted as much of PEP 8 as possible and left things such as `mixedCase` for variables, functions, and methods as part of its own coding standard. And this is completely compatible with the PEP 8 suggestion because it exactly says that consistency within a project is more important than consistency with PEP 8's style guide.

The private controversy

For private methods and functions, we usually use a single leading underscore. This is only a naming convention and has no syntactical meaning. But it doesn't mean that leading underscores have no syntactical meaning at all. When a method has two leading underscores, it is renamed on the fly by the interpreter to prevent a name collision with a method from any subclass. This feature of Python is called **name mangling**.

So some people tend to use a double leading underscore for their private attributes to avoid name collision in the subclasses, for example:

```python
class Base(object):
    def __secret(self):
        print("don't tell")

    def public(self):
        self.__secret()

class Derived(Base):
    def __secret(self):
        print("never ever")
```

From this you will see the following output:

```
>>> Base.__secret
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: type object 'Base' has no attribute '__secret'
>>> dir(Base)
['_Base__secret', ..., 'public']
>>> Base().public()
don't tell
>>> Derived().public()
don't tell
```

The original motivation for name mangling in Python was not to provide the same isolation primitive as a private keyword in C++ but to make sure that some base classes implicitly avoid collisions in subclasses, especially if they are intended to be used in multiple inheritance contexts (for example, as mixin classes). But using it for every attribute that isn't public obfuscates the code and makes it extremely hard to extend. This is not Pythonic at all.

For more information on this topic, an interesting thread occurred in the Python-Dev mailing list many years ago, where people argued on the utility of name mangling and its fate in the language. It can be found at `http://mail.python.org/pipermail/python-dev/2005-December/058555.html`.

Let's take a look at the naming styles for special methods.

Special methods

Special methods (`https://docs.python.org/3/reference/datamodel.html#special-method-names`) start and end with a double underscore and form so-called protocols of the language (see `Chapter 4`, *Modern Syntax Elements - Above the Class Level*). Some developers used to call them *dunder* methods as a portmanteau of double underscore. They are used for operator overloading, container definitions, and so on. For the sake of readability, they should be gathered at the beginning of class definitions, as shown in the following code:

```
class WeirdInt(int):
    def __add__(self, other):
        return int.__add__(self, other) + 1

    def __repr__(self):
        return '<weirdo %d>' % self

    # public API
    def do_this(self):
        print('this')

    def do_that(self):
        print('that')
```

No user-defined method should use this convention unless it explicitly has to implement one of the Python object protocols. So don't invent your own *dunder* methods such as this:

```
class BadHabits:
    def __my_method__(self):
        print('ok')
```

Let's discuss the naming styles for arguments in the next section.

Arguments

Arguments are in lowercase, with underscores if needed. They follow the same naming rules as variables because arguments are simply local variables that get their value as function input values. In the following example, `text` and `separator` are arguments of `one_line()` function:

```python
def one_line(text, separator=" "):
    """Convert possibly multiline text to single line"""
    return separator.join(text.split())
```

The naming style of properties is discussed in the next section.

Properties

The names of properties are in lowercase, or in lowercase with underscores. Most of the time they represent an object's state, which can be a noun or an adjective, or a small phrase when needed. In the following code example, the `Container` class is a simple data structure that can return copies of its contents through `unique_items` and `ordered_items` properties:

```python
class Container:
    _contents = []

    def append(self, item):
        self._contents.append(item)

    @property
    def unique_items(self):
        return set(self._contents)

    @property
    def ordered_items(self):
        return list(self._contents)
```

Let's take a look at the naming styles used for classes.

Classes

The names of classes are always in `CamelCase`, and may have a leading underscore when they are private to a module.

In object-oriented programming classes are used to encapsulate the application state. Attributes of objects are record of that state. Methods are used to modify that state, convert it into meaningful values or to produce side effects. This is why class names are often noun phrases and form a usage logic with the method names that are verb phrases. The following code example contains a `Document` class definition with a single `save()` method:

```python
class Document():
    file_name: str
    contents: str
    ...

    def save(self):
        with open(self.file_name, 'w') as file:
            file.write(self.contents)
```

Class instances often use the same noun phrases as the document but spelled with lowercase. So, actual `Document` class usage could be as follows:

```python
new_document = Document()
new_document.save()
```

Let's go through the naming styles for modules and packages.

Modules and packages

Besides the special module __init__, the module names are in lowercase. The following are some examples from the standard library:

- `os`
- `sys`
- `shutil`

The Python standard library does not use underscores for module names to separate words but they are used commonly in many other projects. When the module is private to the package, a leading underscore is added. Compiled C or C++ modules are usually named with an underscore and imported in pure Python modules. Package names follow the same rules, since they act more like structured modules.

Let's discuss the naming guide in the next section.

The naming guide

A common set of naming rules can be applied on variables, methods, functions, and properties. The names of classes and modules play a very important role in namespace construction and greatly affect code readability. This section contains a miniguide that will help you to define meaningful and readable names for your code elements.

Using the has/is prefixes for Boolean elements

When an element holds a Boolean value you can mark it with `is` and/or `has` syntax to make the variable more readable. In the following example, `is_connected` and `has_cache` are such identifiers that hold Boolean states of the `DB` class instances:

```
class DB:
    is_connected = False
    has_cache = False
```

Using plurals for variables that are collections

When an element is holding a sequence, it is a good idea to use a plural form. You can also do the same for various mapping variables and properties. In following example, `connected_users` and `tables` are class attributes that hold multiple values:

```
class DB:
    connected_users = ['Tarek']
    tables = {'Customer':['id', 'first_name', 'last_name']}
```

Using explicit names for dictionaries

When a variable holds a mapping, you should use an explicit name when possible. For example, if a `dict` holds a person's address, it can be named `persons_addresses`:

```
persons_addresses = {'Bill': '6565 Monty Road',
                     'Pamela': '45 Python street'}
```

Avoid generic names and redundancy

You should generally avoid using explicit type names `list`, `dict`, and `set` as parts of variable names even for local variables. Python now offers function and variable annotations and a typing hierarchy that allows you to easily mark an expected type for a given variable so there is no longer a need to describe object types in their names. It makes the code hard to read, understand, and use. Using a built-in name has to be avoided as well to avoid shadowing it in the current namespace. Generic verbs should also be avoided, unless they have a meaning in the namespace.

Instead, domain-specific terms should be used as follows:

```
def compute(data):  # too generic
    for element in data:
        yield element ** 2

def squares(numbers):  # better
    for number in numbers:
        yield number ** 2
```

There is also the following list of prefixes and suffixes that, despite being very common in programming, should be, in fact, avoided in function and class names:

- Manager
- Object
- Do, handle, or perform

The reason for this is that they are vague, ambiguous, and do not add any value to the actual name. Jeff Atwood, the co-founder of Discourse and Stack Overflow, has a very good article on this topic and it can be found on his blog at `http://blog.codinghorror.com/i-shall-call-it-somethingmanager/`

There is also a list of package names that should be avoided. Everything that does not give any clue about its content can do a lot of harm to the project in the long term. Names such as `misc`, `tools`, `utils`, `common`, or `core` have a very strong tendency to become endless bags of various unrelated code pieces of very poor quality that seem to grow in size exponentially. In most cases, the existence of such a module is a sign of laziness or lack of enough design efforts. Enthusiasts of such module names can simply forestall the future and rename them to `trash` or `dumpster` because this is exactly how their teammates will eventually treat such modules.

In most cases, it is almost always better to have more small modules even with very little content but with names that reflect well what is inside. To be honest, there is nothing inherently wrong with names such as `utils` and `common` and there is a possibility to use them responsibly. But reality shows that in many cases they instead become a stub for dangerous structural antipatterns that proliferate very fast. And if you don't act fast enough, you may not be able get rid of them ever. So the best approach is simply to avoid such risky organizational patterns and nip them in the bud.

Avoiding existing names

It is a bad practice to use names that shadow other names that already exist in the same context. It makes code reading and debugging very confusing. Always try to define original names, even if they are local to the context. If you eventually have to reuse existing names or keywords, use a trailing underscore to avoid name collision, for example:

```
def xapian_query(terms, or_=True):
    """if or_ is true, terms are combined with the OR clause"""
    ...
```

Note that the `class` keyword is often replaced by `klass` or `cls`:

```
def factory(klass, *args, **kwargs):
    return klass(*args, **kwargs)
```

Let's take a look at some of the best practices to keep in mind while working with arguments.

Best practices for arguments

The signatures of functions and methods are the guardians of code integrity. They drive its usage and build its APIs. Besides the naming rules that we have discussed previously, special care has to be taken for arguments. This can be done through the following three simple rules:

- Build arguments by iterative design.
- Trust the arguments and your tests.
- Use `*args` and `**kwargs` magic arguments carefully.

Building arguments by iterative design

Having a fixed and well-defined list of arguments for each function makes the code more robust. But this can't be done in the first version, so arguments have to be built by iterative design. They should reflect the precise use cases the element was created for, and evolve accordingly.

Consider the following example of the first versions of some `Service` class:

```
class Service:  # version 1
    def _query(self, query, type):
        print('done')

    def execute(self, query):
        self._query(query, 'EXECUTE')
```

If you want to extend the signature of the `execute()` method with new arguments in a way that preserves backward compatibility, you should provide default values for these arguments as follows:

```
class Service(object):  # version 2
    def _query(self, query, type, logger):
        logger('done')

    def execute(self, query, logger=logging.info):
        self._query(query, 'EXECUTE', logger)
```

The following example from an interactive session presents two styles of calling the `execute()` method of the updated `Service` class:

```
>>> Service().execute('my query')    # old-style call
>>> Service().execute('my query', logging.warning)
WARNING:root:done
```

Trusting the arguments and your tests

Given the dynamic typing nature of Python, some developers use assertions at the top of their functions and methods to make sure the arguments have proper content, for example:

```
def divide(dividend, divisor):
    assert isinstance(dividend, (int, float))
    assert isinstance(divisor, (int, float))
    return dividend / divisor
```

This is often done by developers who are used to static typing and feel that something is missing in Python.

This way of checking arguments is a part of the **Design by Contract** (**DbC**) programming style, where preconditions are checked before the code is actually run.

The two main problems in this approach are as follows:

- DbC's code explains how it should be used, making it less readable
- This can make it slower, since the assertions are made on each call

The latter can be avoided with the -O option of the Python interpreter. In that case, all assertions are removed from the code before the byte code is created, so that the checking is lost.

In any case, assertions have to be done carefully, and should not be used to bend Python to a statically typed language. The only use case for this is to protect the code from being called nonsensically. If you really want to have some kind of static typing in Python, you should definitely try MyPy or a similar static type checker that does not affect your code runtime and allows you to provide type definitions in a more readable form as function and variable annotations.

Using *args and **kwargs magic arguments carefully

The `*args` and `**kwargs` arguments can break the robustness of a function or method. They make the signature fuzzy, and the code often starts to become a small argument parser where it should not, for example:

```
def fuzzy_thing(**kwargs):
    if 'do_this' in kwargs:
        print('ok i did this')

    if 'do_that' in kwargs:
        print('that is done')

    print('ok')

>>> fuzzy_thing(do_this=1)
ok i did this
ok
>>> fuzzy_thing(do_that=1)
```

```
that is done
ok
>>> fuzzy_thing(what_about_that=1)
ok
```

If the argument list gets long and complex, it is tempting to add magic arguments. But this is more a sign of a weak function or method that should be broken into pieces or refactored.

When `*args` is used to deal with a sequence of elements that are treated the same way in the function, asking for a unique container argument such as an `iterator` is better, for example:

```
def sum(*args):  # okay
    total = 0
    for arg in args:
        total += arg
    return total

def sum(sequence):  # better!
    total = 0
    for arg in sequence:
        total += arg
    return total
```

For `**kwargs`, the same rule applies. It is better to fix the named arguments to make the method's signature meaningful, for example:

```
def make_sentence(**kwargs):
    noun = kwargs.get('noun', 'Bill')
    verb = kwargs.get('verb', 'is')
    adjective = kwargs.get('adjective', 'happy')
    return f'{noun} {verb} {adjective}'

def make_sentence(noun='Bill', verb='is', adjective='happy'):
    return f'{noun} {verb} {adjective}'
```

Another interesting approach is to create a container class that groups several related arguments to provide an execution context. This structure differs from `*args` or `**kwargs` because it can provide internals that work over the values, and can evolve independently. The code that uses it as an argument will not have to deal with its internals.

For instance, a web request passed on to a function is often represented by an instance of a class. This class is in charge of holding the data passed by the web server, as shown in the following code:

```
def log_request(request):  # version 1
    print(request.get('HTTP_REFERER', 'No referer'))

def log_request(request):  # version 2
    print(request.get('HTTP_REFERER', 'No referer'))
    print(request.get('HTTP_HOST', 'No host'))
```

Magic arguments cannot be avoided sometimes, especially in metaprogramming. For instance, they are indispensable in the creation of decorators that work on functions with any kind of signature.

Let's discuss class names in the next section.

Class names

The name of a class has to be concise, precise, and descriptive. A common practice is to use a suffix that informs about its type or nature, for example:

- **SQL**Engine
- **Mime**Types
- **String**Widget
- **Test**Case

For base or abstract classes, a **Base** or **Abstract** prefix can be used as follows:

- **Base**Cookie
- **Abstract**Formatter

The most important thing is to be consistent with the class attributes. For example, try to avoid redundancy between the class and its attributes' names as follows:

```
>>> SMTP.smtp_send()  # redundant information in the namespace
>>> SMTP.send()       # more readable and mnemonic
```

Let's take a look at module and package names in the next section.

Module and package names

The module and package names inform about the purpose of their content. The names are short, in lowercase, and usually without underscores, for example:

- `sqlite`
- `postgres`
- `sha1`

They are often suffixed with `lib` if they are implementing a protocol, as in the following:

```
import smtplib
import urllib
import telnetlib
```

When choosing a name for a module, always consider its content and limit the amount of redundancy within the whole namespace, for example:

```
from widgets.stringwidgets import TextWidget  # bad
from widgets.strings import TextWidget        # better
```

When a module is getting complex and contains a lot of classes, it is a good practice to create a package and split the module's elements into other modules.

The `__init__` module can also be used to put back some common APIs at the top level of the package. This approach allows you to organize the code into smaller components without reducing the ease of use.

Let's take a look at some of the useful tools used while working with naming conventions and styles.

Useful tools

Common conventions and practices used in a software project should always be documented. But having proper documentation for guidelines is often not enough to enforce that these guidelines are actually followed. Fortunately, you can use automated tools that can check sources of your code and verify if it meets specific naming conventions and style guidelines.

The following are a few popular tools:

- `pylint`: This is a very flexible source code analyzer
- `pycodestyle` and `flake8`: This is a small code style checker and a wrapper that adds to it some more useful features, such as static analysis and complexity measurement

Pylint

Besides some quality assurance metrics, Pylint allows for checking of whether a given source code is following a naming convention. Its default settings correspond to PEP 8 and a Pylint script provides a shell report output.

To install Pylint, you can use `pip` as follows:

```
$ pip install pylint
```

After this step, the command is available and can be run against a module, or several modules using wildcards. Let's try it on Buildout's `bootstrap.py` script as follows:

```
$ wget -O bootstrap.py https://bootstrap.pypa.io/bootstrap-buildout.py -q
$ pylint bootstrap.py
No config file found, using default configuration
************* Module bootstrap
C: 76, 0: Unnecessary parens after 'print' keyword (superfluous-parens)
C: 31, 0: Invalid constant name "tmpeggs" (invalid-name)
C: 33, 0: Invalid constant name "usage" (invalid-name)
C: 45, 0: Invalid constant name "parser" (invalid-name)
C: 74, 0: Invalid constant name "options" (invalid-name)
C: 74, 9: Invalid constant name "args" (invalid-name)
C: 84, 4: Import "from urllib.request import urlopen" should be placed at
the top of the module (wrong-import-position)

...


Global evaluation
-----------------
Your code has been rated at 6.12/10
```

Real Pylint's output is a bit longer and here it has been truncated for the sake of brevity.

Remember that Pylint can often give you false positive warnings that decrease the overall quality rating. For instance, an import statement that is not used by the code of the module itself is perfectly fine in some cases (for example, building top-level __init__ modules in a package). Always treat Pylint's output as a hint and not an oracle.

Making calls to libraries that are using `mixedCase` for methods can also lower your rating. In any case, the global evaluation of your code score is not that important. Pylint is just a tool that points you to places where there is the possibility for improvements.

It is always recommended to do some tuning of Pylint. In order to do so you need to create a `.pylinrc` configuration file in your project's root directory. You can do that using the following `-generate-rcfile` option of the `pylint` command:

```
$ pylint --generate-rcfile > .pylintrc
```

This configuration file is self-documenting (every possible option is described with comment) and should already contain every available Pylint configuration option.

Besides checking for compliance with some arbitrary coding standards, Pylint can also give additional information about the overall code quality, such as:

- Code duplication metrics
- Unused variables and imports
- Missing function, method, or class docstrings
- Too long function signatures

The list of available checks that are enabled by default is very long. It is important to know that some of the rules are very arbitrary and cannot always be easily applied to every code base. Remember that consistency is always more valuable than compliance to some arbitrary rules. Fortunately, Pylint is very tunable, so if your team uses some naming and coding conventions that are different from the ones assumed by default, you can easily configure Pylint to check for consistency with your own conventions.

pycodestyle and flake8

`pycodestyle` (formerly `pep8`) is a tool that has only one purpose; it provides only style checking against code conventions defined in PEP 8. This is the main difference from Pylint that has many more additional features. This is the best option for programmers that are interested in automated code style checking only for the PEP 8 standard, without any additional tool configuration, as in Pylint's case.

`pycodestyle` can be installed with `pip` as follows:

```
$ pip install pycodestyle
```

When run on the Buildout's `bootstrap.py` script, it will give the following short list of code style violations:

```
$ wget -O bootstrap.py https://bootstrap.pypa.io/bootstrap-buildout.py -q
$ pycodestyle bootstrap.py
bootstrap.py:118:1: E402 module level import not at top of file
bootstrap.py:119:1: E402 module level import not at top of file
bootstrap.py:190:1: E402 module level import not at top of file
bootstrap.py:200:1: E402 module level import not at top of file
```

The main difference from Pylint's output is its length. `pycodestyle` concentrates only on style, so it does not provide any other warnings, such as unused variables, too long function names, or missing docstrings. It also does not give a rating. And it really makes sense because there is no such thing as partial consistency or partial conformance. Any, even the slightest, violation of style guidelines makes the code immediately inconsistent.

The code of `pycodestyle` is simpler than Pylint's and its output is easier to parse, so it may be a better choice if you want to make your code style verification part of a continuous integration process. If you are missing some static analysis features, there is the `flake8` package that is a wrapper on `pycodestyle` and a few other tools that are easily extendable and provide a more extensive suite of features. These include the following:

- McCabe complexity measurement
- Static analysis via `pyflakes`
- Disabling whole files or single lines using comments

Summary

In this chapter, we have discussed the most common and widely accepted coding conventions. We started with the official Python style guide (the PEP 8 document). The official style guide was complemented by some naming suggestions that will make your future code more explicit. We have also seen some useful tools that are indispensable in maintaining the consistency and quality of your code.

All of this prepares us for the first practical topic of the book—writing and distributing your own packages. In the next chapter, we will learn how to publish our very own package on a public PyPI repository and also how to leverage the power of packaging ecosystems in your private organization.

7
Writing a Package

This chapter focuses on a repeatable process of writing and releasing Python packages. We will see how to shorten the time needed to set up everything before starting the real work. We will also learn how to provide a standardized way to write packages and ease the use of a test-driven development approach. We will finally learn how to facilitate the release process.

It is organized into the following four parts:

- A **common pattern** for all packages that describes the similarities between all Python packages, and how distutils and setuptools play a central role the packaging process.
- What are **namespace packages** and why they can be useful?
- How to register and upload packages in the **Python Package Index** (**PyPI**) with emphasis on security and common pitfalls.
- The **standalone executables** as an alternative way to package and distribute Python applications.

In this chapter, we will cover the following topics:

- Creating a package
- Namespace packages
- Uploading a package
- Standalone executables

Technical requirements

Here are Python packages mentioned in this chapter that you can download from PyPI:

- `twine`
- `wheel`
- `cx_Freeze`
- `py2exe`
- `pyinstaller`

You can install these packages using following command:

```
python3 -m pip install <package-name>
```

Code files for this chapter can be found at
`https://github.com/PacktPublishing/Expert-Python-Programming-Third-Edition/tree/master/chapter7`.

Creating a package

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.

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 in the packaging ecosystem and will help you to work with third-party code that is available on PyPI that you are probably already using.

Also, having your closed source project or its components available as source distribution packages can help you to deploy your code in different environments. The advantages of leveraging the Python packaging ecosystem in the code deployment process will be described in more detail in the next chapter. Here we will focus on proper tools and techniques to create such distributions.

We'll discuss the confusing state of Python package tools in the next section.

The confusing state of Python packaging tools

The state of Python packaging was very confusing for a long time and it took many years to bring organization to this topic. Everything started with the distutils package introduced in 1998, which was later enhanced by setuptools in 2003. These two projects started a long and knotted story of forks, alternative projects, and complete rewrites that tried to (once and for all) fix the Python packaging ecosystem. Unfortunately, most of these attempts never succeeded. The effect was quite the opposite. Each new project that aimed to supersede setuptools or distutils only added to the already huge confusion around packaging tools. Some of such forks were merged back to their ancestors (such as distribute which was a fork of setuptools) but some were left abandoned (such as distutils2).

Fortunately, this state is gradually changing. An organization called the **Python Packaging Authority** (**PyPA**) was formed to bring back the order and organization to the 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. Treat that site as the best source of information about packaging and complementary reading for this chapter. This guide also contains a detailed history of changes and new projects related to packaging. So it is worth reading it, even if you already know a bit about packaging, to make sure you still use the proper tools.

Stay away from other popular internet resources, such as **The Hitchhiker's Guide to Packaging**. It is old, not maintained, and mostly obsolete. It may be interesting only for historical reasons, and the Python Packaging User Guide is in fact a fork of this old resource.

Let's take a look at the effect of PyPA on Python packaging.

The current landscape of Python packaging thanks to PyPA

PyPA, besides providing an authoritative guide for packaging, also maintains packaging projects and a standardization process for new official aspects of Python packaging. All of PyPA's projects can be found under a single organization on GitHub: `https://github.com/pypa`.

Some of them were already mentioned in the book. The following are the most notable:

- `pip`
- `virtualenv`
- `twine`
- `warehouse`

Note that most of them were started outside of this organization and were moved under PyPA patronage when they become mature and widespread solutions.

Thanks to PyPA engagement, the progressive abandonment of the eggs format in favor of wheels for built distributions has already happened. Also thanks to the commitment of the PyPA community, the old PyPI implementation was finally totally rewritten in the form of the **Warehouse** project. Now, PyPI has got a modernized user interface and many long-awaited usability improvements and features.

In the next section, we'll take a look at some of the tools recommended while working with packages.

Tool recommendations

The Python Packaging User Guide gives a few suggestions on recommended tools for working with packages. They can be generally divided into the following two groups:

- Tools for installing packages
- Tools for package creation and distribution

Utilities from the first group recommended by PyPA were already mentioned in `Chapter 2`, *Modern Python Development Environments,* but let's repeat them here for the sake of consistency:

- Use `pip` for installing packages from PyPI.
- Use `virtualenv` or `venv` for application-level isolation of the Python runtime environment.

The Python Packaging User Guide recommendations of tools for package creation and distribution are as follows:

- Use `setuptools` to define projects and create **source distributions.**
- Use **wheels** in favor of **eggs** to create **built distributions.**
- Use `twine` to upload package distributions to PyPI.

Let's take a look at how to configure your project.

Project configuration

It should be obvious that the easiest way to organize the code of big applications is to split them into several packages. This makes the code simpler, easier to understand, maintain, and change. It also maximizes the reusability of your code. Separate packages act as components that can be used in various programs.

setup.py

The root directory of a package that has to be distributed contains a setup.py script. It defines all metadata as described in the distutils module. Package metadata is expressed as arguments in a call to the standard setup() function. Despite distutils being the standard library module provided for the purpose of code packaging, it is actually recommended to use the setuptools instead. The setuptools package provides several enhancements over the standard distutils module.

Therefore, the minimum content for this file is as follows:

```
from setuptools import setup

setup(
    name='mypackage',
)
```

name gives the full name of the package. From there, the script provides several commands that can be listed with the --help-commands option, as shown in the following code:

```
$ 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 independent from the project and also provides transparency about how your package was built/distributed to the users and other 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`, `sdist`, and `bdist_wheel` commands' defaults:

```
[global]
quiet=1

[sdist]
formats=zip,tar

[bdist_wheel]
universal=1
```

This example configuration will ensure that source distributions (`sdist` section) will always be created in two formats (ZIP and TAR) and the built `wheel` distributions (`bdist_wheel` section) will be created as universal wheels that are independent from the Python version. Also most of the output will be suppressed on every command by the global `--quiet` switch. Note that this option is included here only for demonstration purposes and it may not be a reasonable choice to suppress the output for every command by default.

MANIFEST.in

When building a distribution with the `sdist` command, the `distutils` module browses the package directory looking for files to include in the archive. By default `distutils` will include the following:

- All Python source files implied by the `py_modules`, `packages`, and `scripts` arguments
- All C source files listed in the `ext_modules` argument
- Files that match the glob pattern `test/test*.py`
- Files named `README`, `README.txt`, `setup.py`, and `setup.cfg`

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 is the default built-in collection strategy or one defined by custom extension, the `sdist` will create a `MANIFEST` file that lists all files and will include them in the final archive.

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 `setup.py` file). This template directs the `sdist` command on which files to include.

This `MANIFEST.in` template defines one inclusion or exclusion rule per line:

```
include HISTORY.txt
include README.txt
include CHANGES.txt
include CONTRIBUTORS.txt
include LICENSE
recursive-include *.txt *.py
```

The full list of the `MANIFEST.in` commands can be found in the official `distutils` documentation.

Most important metadata

Besides the name and the version of the package being distributed, the most important arguments that the `setup()` function can receive are as follows:

- `description`: This includes a few sentences to describe 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 MIME type of 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.
- `url`: This is the URL of the project.
- `license`: This is the name of the license (GPL, LGPL, and so on) under which the package is distributed.
- `packages`: This is a list of all package names in the package distribution; `setuptools` provides a small function called `find_packages` that can automatically find package names to include.
- `namespace_packages`: This is a list of namespace packages within package distribution.

Trove classifiers

PyPI and `distutils` provide 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 `solrq` project available on PyPI:

```
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 packages 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 667 classifiers available on PyPI that are grouped into the following nine major categories:

- Development status
- Environment
- Framework

- Intended audience
- License
- Natural language
- Operating system
- Programming language
- Topic

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/`.

Common patterns

Creating a package for distribution can be a tedious task for unexperienced developers. Most of the metadata that `setuptools` or `distuitls` accept in their `setup()` function call can be provided manually ignoring the fact that this metadata may be also available in other parts of the project. Here is an example:

```
from setuptools import setup

setup(
    name="myproject",
    version="0.0.1",
    description="mypackage project short description",
    long_description="""
        Longer description of mypackage project
        possibly with some documentation and/or
        usage examples
    """,
    install_requires=[
        'dependency1',
        'dependency2',
        'etc',
    ]
)
```

Some of the metadata elements are often found in different places in a typical Python project. For instance, content of long description is commonly included in the project's README file, and it is a good convention to put a version specifier in the *__init__* module of the package. Hardcoding such package metadata as `setup()` function arguments redundancy to the project that allows for easy mistakes and inconsistencies in future. Both `setuptools` and `distutils` cannot automatically pick metadata information from the project sources, so you need to provide it yourself. There are some common patterns among the Python community for solving the most popular problems such as dependency management, version/readme inclusion, and so on. It is worth knowing at least a few of them because they are so popular that they could be considered as packaging idioms.

Automated inclusion of version string from package

The PEP 440 *Version Identification and Dependency Specification* document specifies a standard for version and dependency specification. It is a long document that covers accepted version specification schemes and defines how version matching and comparison in Python packaging tools should work. If you are using or plan to use a complex project version numbering scheme, then you should definitely read this document carefully. If you are using a simple scheme that consists just of one, two, three, or more numbers separated by dots, then you don't have to dig into the details of PEP 440. If you don't know how to choose the proper versioning scheme, I greatly recommend following the semantic versioning scheme that was already briefly mentioned in `Chapter 1`, *Current Status of Python*.

The other problem related to code versioning is where to include that version specifier for a package or module. There is PEP 396 (Module Version Numbers) that deals exactly with this problem. PEP 396 is only an informational document and has a *deferred* status, so it is not a part of the official Python standards track. Anyway, it describes what seems to be a *de facto* standard now. According to PEP 396, if a package or module has a specific version defined, the version specifier should be included as a `__version__` attribute of package root `__init__.py` INI file or distributed module file. Another *de facto* standard is to also include the `VERSION` attribute that contains the tuple of the version specifier parts. This helps users to write compatibility code because such version tuples can be easily compared if the versioning scheme is simple enough.

So many packages available on PyPI follow both conventions. Their __init__.py files contain version attributes that look like the following:

```
# version as tuple for simple comparisons
VERSION = (0, 1, 1)
# string created from tuple to avoid inconsistency
__version__ = ".".join([str(x) for x in VERSION])
```

The other suggestion of PEP 396 is that the version argument provided in the setup() function of the setup.py script should be derived from __version__, or the other way around. The Python Packaging User Guide features multiple patterns for single-sourcing project versioning, and each of them has its own advantages and limitations. My personal favorite is rather long and is not included in the PyPA's guide, but has the advantage of limiting the complexity only to the setup.py script. This boilerplate assumes that the version specifier is provided by the VERSION attribute of the package's __init__ module and extracts this data for inclusion in the setup() call. Here is an excerpt from some imaginary package's setup.py script that illustrates this approach:

```
from setuptools import setup
import os


def get_version(version_tuple):
    # additional handling of a,b,rc tags, this can
    # be simpler depending on your versioning scheme
    if not isinstance(version_tuple[-1], int):
        return '.'.join(
            map(str, version_tuple[:-1])
        ) + version_tuple[-1]
    return '.'.join(map(str, version_tuple))

# path to the packages __init__ module in project
# source tree
init = os.path.join(
    os.path.dirname(__file__), 'src', 'some_package',
    '__init__.py'
)

version_line = list(
    filter(lambda l: l.startswith('VERSION'), open(init))
)[0]

# VERSION is a tuple so we need to eval 'version_line'.
# We could simply import it from the package but we
# cannot be sure that this package is importable before
# installation is done.
```

```
PKG_VERSION = get_version(eval(version_line.split('=')[-1]))

setup(
    name='some-package',
    version=PKG_VERSION,
    # ...
)
```

README file

The Python Package Index can display the project's README file or the value of `long_description` on the package page in the PyPI portal. PyPI is able to interpret the markup used in the `long_description` content and render it as HTML on the package page. The type of markup language is controlled through the `long_description_content_type` argument of the `setup()` call. For now, there are the following three choices for markup available:

- Plain text with `long_description_content_type='text/plain'`
- reStructuredText with `long_description_content_type='text/x-rst'`
- Markdown with `long_description_content_type='text/markdown'`

Markdown and reStructuredText are the most popular choices among Python developers, but some might still want to use different markup languages for various reasons. If you want to use something different as your markup language for your project's README, you can still provide it as a project description on the PyPI page in a readable form. The trick lies in using the `pypandoc` package to translate your other markup language into reStructuredText (or Markdown) while uploading the package to the Python Package Index. It is important to do it with a fallback to plain content of your README file, so the installation won't fail if the user has no `pypandoc` installed. The following is an example of a `setup.py` script that is able to read the content of the README file written in AsciiDoc markup language and translate it to reStructuredText before including a `long_description` argument:

```
from setuptools import setup
try:
    from pypandoc import convert

    def read_md(file_path):
        return convert(file_path, to='rst', format='asciidoc')

except ImportError:
    convert = None
    print(
        "warning: pypandoc module not found, "
```

```
            "could not convert Asciidoc to RST"
    )

    def read_md(file_path):
        with open(file_path, 'r') as f:
            return f.read()

README = os.path.join(os.path.dirname(__file__), 'README')

setup(
    name='some-package',
    long_description=read_md(README),
    long_description_content_type='text/x-rst',
    # ...
)
```

Managing dependencies

Many projects require some external packages to be installed in order to work properly. When the list of dependencies is very long, there comes a question as to how to manage it. The answer in most cases is very simple. Do not over-engineer it. Keep it simple and provide the list of dependencies explicitly in your setup.py script as follows:

```
from setuptools import setup
setup(
    name='some-package',
    install_requires=['falcon', 'requests', 'delorean']
    # ...
)
```

Some Python developers like to use requirements.txt files for tracking lists of dependencies for their packages. In some situations, you might find some reason for doing that, but in most cases, this is a relic of times where the code of that project was not properly packaged. Anyway, even such notable projects as Celery still stick to this convention. So if you are not willing to change your habits or you are somehow forced to use requirement files, then at least do it properly. Here is one of the popular idioms for reading the list of dependencies from the requirements.txt file:

```
from setuptools import setup
import os


def strip_comments(l):
    return l.split('#', 1)[0].strip()
```

```
def reqs(*f):
    return list(filter(None, [strip_comments(l) for l in open(
        os.path.join(os.getcwd(), *f)).readlines()]))

setup(
    name='some-package',
    install_requires=reqs('requirements.txt')
    # ...
)
```

In next section, you'll learn how to add custom commands to your setup script.

The custom setup command

`distutils` allows you to create new commands. A new command can be registered with an entry point, which was introduced by `setuptools` as a simple way to define packages as plugins.

An entry point is a named link to a class or a function that is made available through some APIs in `setuptools`. Any application can scan for all registered packages and use the linked code as a plugin.

To link the new command, the `entry_points` metadata can be used in the setup call as follows:

```
setup(
    name="my.command",
    entry_points="""
        [distutils.commands]
        my_command = my.command.module.Class
    """
)
```

All named links are gathered in named sections. When `distutils` is loaded, it scans for links that were registered under `distutils.commands`.

This mechanism is used by numerous Python applications that provide extensibility.

Let's see how to work with packages during the development stage.

Working with packages during development

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. Also, 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 the Python Package Index.

An alternative to the bare `setup.py` script when installing a package 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>
```

Uninstalling packages

Amazingly, `setuptools` and `distutils` lack 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.

setup.py develop or pip -e

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 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 the `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.

Let's take a look at namespace packages in the next section.

Namespace packages

The Zen of Python that you can read after writing `import this` in the interpreter session 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 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.

Why is it useful?

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` with an `sql` subpackage/submodule with the following file structure:

```
$ tree acme/
acme/
├── acme
│   ├── __init__.py
│   └── sql
│       └── __init__.py
└── setup.py

2 directories, 3 files
```

Whenever you want to add a new subpackage, let's say `templating`, you are forced to include it in the source tree of `acme` as follows:

```
$ tree acme/
acme/
├── acme
│   ├── __init__.py
│   ├── sql
│   │   └── __init__.py
│   └── templating
│       └── __init__.py
└── setup.py

3 directories, 4 files
```

Such an approach makes independent development
of `acme.sql` and `acme.templating` almost impossible. The `setup.py` script will also
have to specify all dependencies for every subpackage. So it is impossible (or at least very
hard) to have an installation of some of the `acme` components optional. Also, with enough
subpackages it is practically impossible to avoid dependency conflicts.

With namespace packages, you can store the source tree for each of these subpackages
independently as follows:

```
$ tree acme.sql/
acme.sql/
├── acme
│   └── sql
│       └── __init__.py
└── setup.py

2 directories, 2 files

$ tree acme.templating/
acme.templating/
├── acme
│   └── templating
│       └── __init__.py
└── setup.py

2 directories, 2 files
```

And you can also register them independently in PyPI or any package index you use. Users
can choose which of the subpackages they want to install from the `acme` namespace as
follows, but they never install the general `acme` package (it doesn't even have to exist):

```
$ pip install acme.sql acme.templating
```

Note that independent source trees are not enough to create namespace packages in
Python. You need a bit of additional work if you don't want your packages to not overwrite
each other. Also proper handling may be different depending on the Python language
version you target. Details of that are described in the next two sections.

PEP 420 - implicit namespace packages

If you use and target only Python 3, then there is good news for you. **PEP 420 (Implicit Namespace Packages)** introduced a new way to define namespace packages. It is part of the standards track and became an official part of the language since version 3.3. In short, every directory that contains Python packages or modules (including namespace packages too) is considered a namespace package if it does not contain the __init__.py file. So, the following are examples of file structures presented in the previous section:

```
$ tree acme.sql/
acme.sql/
├── acme
│   └── sql
│       └── __init__.py
└── setup.py

2 directories, 2 files

$ tree acme.templating/
acme.templating/
├── acme
│   └── templating
│       └── __init__.py
└── setup.py

2 directories, 2 files
```

They are enough to define that acme is a namespace package under Python 3.3 and later. Minimal setup.py for acme.templating package will look like following:

```
from setuptools import setup
setup(
    name='acme.templating',
    packages=['acme.templating'],
)
```

Unfortunately, the setuptools.find_packages() function does not support PEP 420 at the time of writing this book. This may change in the future. Also, a requirement to explicitly define a list of packages seems to be a very small price to pay for easy integration of namespace packages.

Namespace packages in previous Python versions

You can't use implicit namespace packages (PEP 420 layout) in Python versions older than 3.3. Still, the concept of namespace packages is very old and was commonly used for years in such mature projects such as Zope. It means that it is definitely possible to use namespace packages in older version of Python. Actually, there are several ways to define that the package should be treated as a namespace.

The simplest one is to create a file structure for each component that resembles an ordinary package layout without implicit namespace packages and leave everything to setuptools. So, the example layout for acme.sql and acme.templating could be the following:

```
$ tree acme.sql/
acme.sql/
├── acme
│   ├── __init__.py
│   └── sql
│       └── __init__.py
└── setup.py

2 directories, 3 files

$ tree acme.templating/
acme.templating/
├── acme
│   ├── __init__.py
│   └── templating
│       └── __init__.py
└── setup.py
2 directories, 3 files
```

Note that for both acme.sql and acme.templating, there is an additional source file, acme/__init__.py. This file must be left empty. The acme namespace package will be created if we provide its name as a value of the namespace_packages keyword argument of the setuptools.setup() function as follows:

```
from setuptools import setup

setup(
    name='acme.templating',
    packages=['acme.templating'],
    namespace_packages=['acme'],
)
```

Easiest does not mean best. The `setuptools` module in order to register a new namespace will call for the `pkg_resources.declare_namespace()` function in your `__init__.py` file. It will happen even if the `__init__.py` file is empty. Anyway, as the official documentation says, it is your own responsibility to declare namespaces in the `__init__.py` file, and this implicit behavior of `setuptools` may be dropped in the future. In order to be safe and *future-proof*, you need to add the following line to the `acme/__init__.py` file:

```
__import__('pkg_resources').declare_namespace(__name__)
```

This line will make your namespace package safe from potential future changes regarding namespace packages in the `setuptools` module.

Let's see how to upload a package in the next section.

Uploading a package

Packages would be useless without an organized way to store, upload, and download them. 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: `https://pypi.python.org/pypi`.

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. Details of such packaging usage with instructions on how to create your own package index will be explained in the next chapter. Here we focus mainly on open source uploads to PyPI, with only little mention on how to specify alternative repositories.

PyPI - Python Package Index

Python Package Index is, as already mentioned, the official source of open source package distributions. Downloading from it does not require any account or permission. The only thing you need is a package manager that can download new distributions from PyPI. Your preferred choice should be `pip`.

Let's see how to upload a package in the next section.

Uploading to PyPI - or other package index

Anyone can register and upload packages to PyPI provided that he or she has an account registered. 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.

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 upload source distribution, built distribution, and `wheel` package 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 in previous command calls and are forced to rebuild them on every upload. This may be inconvenient for large or complex projects where creation of the actual distribution may take a considerable amount of time. Another problem of `setup.py upload` is that it can use plain text HTTP or unverified HTTPS connections on some Python versions. 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 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/*
```

Let's see what `.pypric` is in the next section.

.pypirc

`.pypirc` is a configuration file that stores information about Python packages 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 and you should be prompted for it whenever it is necessary.

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`.

Let's take a look at the comparison between source packages and built packages.

Source packages versus built packages

There are generally the following 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 proper development toolchain in his/her environment. This consists mostly of the compiler and proper C header files. For such cases, the build distribution format may be better suited because it can provide already built extensions for specific platforms.

Let's take a look at what exactly the `sdist` command.

sdist

The `sdist` command is the simplest command available. It creates a release tree where everything that is needed to run the package is copied to. This tree is then archived in one or many archived 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 from 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, as follows. If you don't, `setuptools` module will assume default value of `version = '0.0.0'`:

```
from setuptools import setup

setup(name='acme.sql', version='0.1.1')
```

Every time a package is released, the version number should be increased so that the target system knows the package has changed.

Let's run the following `sdist` command for `acme.sql` package in `0.1.1` version:

```
$ python setup.py sdist
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)

$ ls dist/
acme.sql-0.1.1.tar.gz
```

> On Windows, the default archive type will be ZIP.

The version is used to mark the name of the archive, which 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 under Windows. That's why a package should always be distributed with a prebuilt distribution as well, when it is intended to be run on several platforms.

Let's take a look at what the `bdist` and `wheels` commands are in the next section.

bdist and wheels

To be able to distribute a prebuilt distribution, `distutils` provides the `build` command. This commands 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 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 `!#` prefix) and fixes the file mode so that it is executable.

Each of these steps is a command that can be called 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-compiler option yet in the `distutils` 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`). This **include** file is available from the time Python was built from the sources. For a packaged distribution, an extra package for your system distribution is probably required. At least in popular Linux distributions, it is often named `python-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 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. This can be configured in `distutils`.

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` on macOS as follows:

```
$ python setup.py bdist
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)

$ 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 another system, specific distribution archive as follows:

```
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)  22 239 752 192 bytes 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 a tree that can be copied directly into the Python tree. It mainly contains a folder that is copied into Python's `site-packages` folder. It may also contain cached bytecode files (`*.pyc` files on Python 2 and `__pycache__/*.pyc` on Python 3).

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 `distutils`. It allows creating 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 (`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 PyPA's recommendation, 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**. The process of building them is unfortunately not as straightforward as for Windows and macOS binary wheels. For these kind of wheels, PyPA maintains special Docker images that serve as a ready-to-use build environments. For sources of these images and more information, you can visit their official repository on GitHub: `https://github.com/pypa/manylinux`.

Let's take a look at standalone executables in the next section.

Standalone executables

Creating standalone executables is a commonly overlooked topic in materials that cover 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 from them 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 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 of Python applications, this takes the following form:

```
#!/usr/bin/env python
```

Such directive when used as a 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 `python3.4`, `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, bootstrapping of the Python environment on Windows can be a challenge even for experienced developers, so you cannot expect that nontechnical users will be able to do that by themselves.

The other thing to consider is the simple user experience in the desktop environment. Users usually expect that applications can be run from the desktop by simply clicking on them. Not every desktop environment will support that with Python applications distributed as a source.

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 when standalone executables are useful.

When standalone executables useful?

Standalone executables are useful in situations where simplicity of the user experience is more important than the user's ability to interfere with the applications 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 nontechnical 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

Let's take a look at some of the popular tools in the next section.

Popular tools

Python does not have any built-in support for building standalone executables. Fortunately, there are some community projects solving that problem with varied amounts 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 each one of them has 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. None of these tools, of course, requires deep interaction in your code, but if you start building standalone packages early, you can automate the whole process and save future integration time and costs. 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. Providing a standalone executable for such a project will be problematic and will take a lot of your time.

Let's take a look at PyInstaller in the next section.

PyInstaller

PyInstaller (`http://www.pyinstaller.org/`) 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

Supported versions of Python are Python 2.7 and Python 3.3, 3.4, and 3.5. 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 Vagrant, 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
2121 INFO: PyInstaller: 3.1
2121 INFO: Python: 2.7.10
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 was truncated in the preceding example for the sake of brevity. If run on Windows, the resulting structure of directories and files will be as follows:

```
$ tree /0066
│   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
            python27.dll
            select.pyd
            unicodedata.pyd
            _hashlib.pyd
```

The `dist/myscript` directory contains the built application that can now be distributed to the users. Note that 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
(...)

$ tree /f
├─────build
│    └─────myscript
│            myscript.exe.manifest
│            out00-Analysis.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 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 `pyinstaller` arguments specified earlier. This is very useful if
you have performed a lot of customizations to 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 building 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 and can be used only when
modifying `.spec` file.

PyInstaller is an extensive tool, which by its usage is very simple 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 (`http://cx-freeze.sourceforge.net/`) 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

Like PyInstaller, it 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. Assuming that we have the same application as featured in the *PyInstaller* section, the example usage is very simple as well:

```
$ cxfreeze myscript.py
copying C:\Python27\lib\site-packages\cx_Freeze\bases\Console.exe ->
D:\dev\app\dist\myscript.exe
copying C:\Windows\system32\python27.dll ->
D:\dev\app\dist\python27.dll
writing zip file D:\dev\app\dist\myscript.exe
(...)
copying C:\Python27\DLLs\bz2.pyd -> D:\dev\app\dist\bz2.pyd
copying C:\Python27\DLLs\unicodedata.pyd -> D:\dev\app\dist\unicodedata.pyd
Resulting structure of files is as follows:

 $ tree /f
│    myscript.py
│
└──────dist
        bz2.pyd
        myscript.exe
        python27.dll
        unicodedata.pyd
```

Instead of providing the 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:

```
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` seems a bit easier than PyInstaller's, and `distutils` integration is a very useful feature. 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.

Let's take a look at py2exe and py2app 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.

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. It is not an easy task to decompile the embedded code from such executable files, but it is definitely doable. What is even more important is that the results of such decompilation (if done with proper tools) might look strikingly similar to original sources.

This fact makes standalone Python executables not a viable solution for closed source projects where leaking of the application code could harm the organization. So, if your whole business can be copied simply by copying the source code of you application, then you should think of other ways to distribute the application. Maybe providing software as a service will be a better choice for you.

Making decompilation harder

As already said, there is no reliable way to secure applications from decompilation with the tools available at the moment. Still, there are some ways to make this process harder. But harder does not mean less probable. For some of us, the most tempting challenges are the hardest ones. And we all know that the eventual price in this challenge is very high—the code that you tried to secure.

Usually the process of decompilation consists of the following steps:

1. Extracting the project's binary representation of bytecode from standalone executables
2. Mapping of a binary representation to bytecode of a specific Python version
3. Translation of bytecode to AST
4. Re-creation of sources directly from AST

Providing the exact solutions for deterring developers from such reverse engineering of standalone executables would be pointless for obvious reasons. 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 Python runtime, but each one of them is riddled with many pitfalls and disadvantages. Mostly, they only defer what is anyway inevitable. Once your trick is broken, it renders all your additional efforts a waste of time and resources.

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 security stay airtight.

Summary

In this chapter, we have discussed the details of Python's packaging ecosystem. Now, after reading it, you should know which tools suit your packaging needs and also which types of distributions your project requires. You should also know the popular techniques for common problems and how to provide useful metadata to your project.

We also discussed the topic of standalone executables that are very useful in distributing desktop applications.

The next chapter will build on what we have learned here to show how to efficiently deal with code deployments in a reliable and automated way.

8
Deploying the Code

Even perfect code (if it exists) is useless if it is not able to run. So, in order to serve any purpose, our code needs to be installed on the target machine (computer) and executed. The process of making a specific version of your application or service available to end users is called deployment.

In the case of desktop applications, this seems to be simple as your job ends with providing a downloadable package with an optional installer, if necessary. It is the user's responsibility to download and install the package in their environment. Your responsibility is to make this process as easy and convenient as possible. Proper packaging is still not a simple task, but some tools were already explained in the previous chapter.

Surprisingly, things get more complicated when your code is not a standalone product. If your application only provides a service that is being sold to users, then it is your responsibility to run it on your own infrastructure. This scenario is typical for a web application or any *X as a service* product. In such a situation, the code is deployed to set off remote machines that are physically accessible to the developers. This is especially true if you are already a user of cloud computing services such as **Amazon Web Services** (**AWS**) or Heroku.

In this chapter, we will concentrate on the aspect of code deployment to remote hosts due to very high popularity of Python in the field of building various web-related services and products. Despite the high portability of this language, it has no specific quality that would make its code easily deployable. What matters the most is how your application is built and what processes you use to deploy it to the target environments. So, this chapter will focus on the following topics:

- What are the main challenges in deploying the code to remote environments?
- How to build applications in Python that are easily deployable
- How to reload web services without downtime
- How to leverage a Python packaging ecosystem in code deployment
- How to properly monitor and instrument code that runs remotely

Technical requirements

You can download various monitoring and log processing tools mentioned in this chapter from the following sites:

- Munin: `http://munin-monitoring.org`
- Logstash, Elasticsearch, and Kibana: `https://www.elastic.co`
- Fluentd: `https://www.fluentd.org`

Following are Python packages mentioned in this chapter that you can download from PyPI:

- `fabric`
- `devpi`
- `circus`
- `uwsgi`
- `gunicorn`
- `sentry_sdk`
- `statsd`

You can install these packages using the following command:

```
python3 -m pip install <package-name>
```

Code files for this chapter can be found at `https://github.com/PacktPublishing/Expert-Python-Programming-Third-Edition/tree/master/chapter8`.

The Twelve-Factor App

The main requirement for painless deployment is building your application in a way that ensures that this process will be simple and as streamlined as possible. This is mostly about removing obstacles and encouraging well-established practices. Following such common practices is especially important in organizations where only specific people are responsible for development (the developers team or Dev for short) and different people are responsible for deploying and maintaining the execution environments (the operations team or Ops for short).

All tasks related to server maintenance, monitoring, deployment, configuration, and so on are often put into one single bag called **operations**. Even in organizations that have no separate teams for operational tasks, it is common that only some of the developers are authorized to do deployment tasks and maintain the remote servers. The common name for such a position is DevOps. Also, it isn't such an unusual situation that every member of the development team is responsible for operations, so everyone in such a team can be called DevOps.

No matter how your organization is structured and what the responsibilities of each developer are, everyone should know how operations work and how code is deployed to the remote servers because, in the end, the execution environment and its configuration is a hidden part of the product you are building.

The following common practices and conventions are important mainly for the following reasons:

- At every company people quit and new ones are hired. By using best approaches, you are making it easier for fresh team members to jump into the project. You can never be sure that new employees are already familiar with common practices for system configuration and running applications in a reliable way, but you can at least make their fast adaptation more probable.
- In organizations where only some people are responsible for deployments, it simply reduces the friction between the operations and development teams.

A good source of such practices that encourage building easy deployable apps is a manifesto called the **Twelve-Factor App**. It is a general language-agnostic methodology for building software-as-a-service 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 its name says, the Twelve-Factor App consists of 12 rules:

- **Code base**: One code base tracked in revision control and many deploys
- **Dependencies**: Explicitly declare and isolate dependencies
- **Config**: Store configurations in the environment
- **Backing services**: Treat backing services as attached resources
- **Build, release, run**: Strictly separate build and run stages
- **Processes**: Execute the app as one or more stateless processes
- **Port binding**: Export services via port binding
- **Concurrency**: Scale out via the process model
- **Disposability**: Maximize robustness with fast startup and graceful shutdown

- **Dev/prod parity**: Keep development, staging, and production as similar as possible
- **Logs**: Treat logs as event streams
- **Admin processes**: Run administration/management tasks as one-off processes

Extending each of these rules here is a bit pointless because the official page of Twelve-Factor App methodology (`http://12factor.net/`) contains extensive rationale for each app factor with examples of tools for different frameworks and environments.

This chapter tries to stay consistent with the preceding manifesto, so we will discuss some of them in detail when necessary. The techniques and examples that are presented may sometimes slightly diverge from these 12 factors, but remember that these rules are not carved in stone. They are great as long as they serve the purpose. In the end, what matters is the working application (product) and not being compatible with some arbitrary methodology.

Let's take a look at the various deployment automation approaches in the next section.

Various approaches to deployment automation

With the advent of application containerization (Docker and similar technologies), modern software provisioning tools (for example, Puppet, Chef, Ansible, and Salt), and infrastructure management systems (for example, Terraform and SaltStack) development and operations teams have a variety of ways in which they can organize and manage their code deployments and configuration of remote systems. Each solution has pros and cons, so advanced automation tools should be chosen very wisely with respect to the favored development processes and methodologies.

Fast paced teams that use microservice architecture and deploy code often (maybe even simultaneously in parallel versions) will definitely favor container orchestration systems such as Kubernetes or use dedicated services provided by their cloud vendor (for example, AWS). Teams that build old-style big monolithic applications and run them on their own bare-metal servers might want to use more low-level automation and software provisioning systems. Actually, there is no rule and you can find teams of every size using every possible approach to software provisioning, code deployments, and application orchestration. The limiting factors here are resources and knowledge.

That's why it's really hard to briefly provide a set of common tools and solutions that would fit the needs and capabilities of every developer and every team. Because of that, in this chapter, we will focus only on a pretty simple approach to automation using Fabric. We could say that this is outdated. And that's probably true. What seem to be the *most modern* are container orchestrations systems in the style of Kubernetes that allow you to leverage Docker containers for fast, maintainable, scalable, and reproducible environments. But these systems have quite a steep learning curve and it's impossible to introduce them in just a few sections of a single chapter. Fabric, on the other hand, is very simple and easy to grasp so it is a really great tool to introduce someone to the concept of automation.

Let's look at the use of Fabric in deployment automation in the next section.

Using Fabric for deployment automation

For very small projects, it may be possible to deploy your code *by hand*, that is, by manually typing the sequence of commands through the remote shells that are necessary to install a new version of code and execute it on a remote shell. Anyway, even for an average-sized project, this is error-prone and tedious and should be considered a waste of most of the precious resource you have, your own time.

The solution for that is automation. The simple thumb rule could be the following:

> *If you needed to perform the same task manually at least twice, you should automate it so you won't need to do it for the third time.*

There are various tools that allow you to automate different things, including the following:

- Remote execution tools such as Fabric are used for on-demand automated execution of code on multiple remote hosts.
- Configuration management tools such as Chef, Puppet, CFEngine, Salt, and Ansible are designed for automatized configuration of remote hosts (execution environments). They can be used to set up backing services (databases, caches, and so on), system permissions, users, and so on. Most of them can be used also as a tool for remote execution (such as Fabric) but, depending on their architecture, this may be more or less convenient.

Configuration management solutions is a complex topic that deserves a separate book. The truth is that the simplest remote execution frameworks have the lowest entry barrier and are the most popular choice, at least for small projects. In fact, every configuration management tool that provides a way to declaratively specify configuration of your machines has a remote execution layer implemented somewhere deep inside.

Also, some configuration management tools may not be best suited for actual automated code deployment. One such example is Puppet, which really discourages the explicit running of any shell commands. This is why many people choose to use both types of solution to complement each other: configuration management for setting up a system-level environment and on-demand remote execution for application deployment.

Fabric (`http://www.fabfile.org/`) is so far the most popular solution used by Python developers to automate remote execution. It is a Python library and command-line tool for streamlining the use of SSH for application deployment or systems administration tasks. We will focus on it because it is relatively easy to start with. Keep in mind that, depending on your needs, it may not be the best solution to your problems. Anyway, it is a great example of a utility that can add some automation to your operations, if you don't have any yet.

You could, of course, automate all of the work using only Bash scripts but this is very tedious and error-prone. Python has more convenient ways for string processing and encourages code modularization. Fabric is in fact only a tool for gluing the execution of commands via SSH. It means that you still need to know how to use the command-line interface and its utilities in your remote environment.

So, if you want to strictly follow the Twelve-Factor App methodology, you should not maintain its code in the source tree of the deployed application.

Complex projects are, in fact, very often built from various components maintained as separate code bases, so this is another reason why it is a good approach to have one separate repository for all of the project component configurations and Fabric scripts. This makes deployment of different services more consistent and encourages good code reuse.

To start working with Fabric, you need to install the `fabric` package (using `pip`) and create a script named `fabfile.py`. That script is usually located in the root of your project. Note that `fabfile.py` can be considered a part of your project configuration.

But before we create our `fabfile` let's define some initial utilities that will help us to set up the project remotely. Here's a module that we will call `fabutils`:

```
import os

# Let's assume we have private package repository created
# using 'devpi' project
PYPI_URL = 'http://devpi.webxample.example.com'

# This is arbitrary location for storing installed releases.
# Each release is a separate virtual environment directory
# which is named after project version. There is also a
```

```
    # symbolic link 'current' that points to recently deployed
    # version. This symlink is an actual path that will be used
    # for configuring the process supervision tool for example,:
    # .
    # ├── 0.0.1
    # ├── 0.0.2
    # ├── 0.0.3
    # ├── 0.1.0
    # └── current -> 0.1.0/

    REMOTE_PROJECT_LOCATION = "/var/projects/webxample"

    def prepare_release(c):
        """ Prepare a new release by creating source distribution and
         uploading to out private package repository
        """
        c.local(f'python setup.py build sdist')
        c.local(f'twine upload --repository-url {PYPI_URL}')

    def get_version(c):
        """ Get current project version from setuptools """
        return c.local('python setup.py --version').stdout.strip()

    def switch_versions(c, version):
        """ Switch versions by replacing symlinks atomically """
        new_version_path = os.path.join(REMOTE_PROJECT_LOCATION, version)
        temporary = os.path.join(REMOTE_PROJECT_LOCATION, 'next')
        desired = os.path.join(REMOTE_PROJECT_LOCATION, 'current')

        # force symlink (-f) since probably there is a one already
        c.run(f"ln -fsT {new_version_path} {temporary}")
        # mv -T ensures atomicity of this operation
        c.run(f"mv -Tf {temporary} {desired}" )
```

An example of a final `fabfile` that defines a simple deployment procedure will look like this:

```
    from fabric import task
    from .fabutils import *

    @task
    def uptime(c):
        """
        Run uptime command on remote host - for testing connection.
        """
        c.run("uptime")

    @task
```

```
def deploy(c):
    """ Deploy application with packaging in mind """
    version = get_version(c)

    pip_path = os.path.join(
        REMOTE_PROJECT_LOCATION, version, 'bin', 'pip'
    )

    if not c.run(f"test -d {REMOTE_PROJECT_LOCATION}", warn=True):
        # it may not exist for initial deployment on fresh host
        c.run(f"mkdir -p {REMOTE_PROJECT_LOCATION}")

    with c.cd(REMOTE_PROJECT_LOCATION):
        # create new virtual environment using venv
        c.run(f'python3 -m venv {version}')

        c.run(f"{pip_path} install webxample=={version} --index-url
{PYPI_URL}")

    switch_versions(c, version)
    # let's assume that Circus is our process supervision tool
    # of choice.
    c.run('circusctl restart webxample')
```

Every function decorated with `@task` is now treated as an available subcommand to the `fab` utility provided with the `fabric` package. You can list all of the available subcommands using the `-l` or `--list` switch. The code is shown in the following snippet:

```
$ fab --list
Available commands:
    deploy  Deploy application with packaging in mind
    uptime  Run uptime command on remote host - for testing connection.
```

Now, you can deploy the application to the given environment type with only the following single shell command:

```
$ fab -H myhost.example.com deploy
```

Note that the preceding `fabfile` serves only illustrative purposes. In your own code, you might want to provide extensive failure handling and try to reload the application without the need to restart the web worker process. Also, some of the techniques presented here may not be obvious right now but will be explained later in this chapter. These include the following:

- Deploying an application using the private package repository
- Using Circus for process supervision on the remote host

In the next section, we will take a look at index mirroring in Python.

Your own package index or index mirror

These are three main reasons why you might want to run your own index of Python packages:

- The official Python Package Index does not have any availability guarantees. It is run by Python Software Foundation thanks to numerous donations. Because of that, it means that this site can be down at the most inconvenient time. You don't want to stop your deployment or packaging process in the middle due to PyPI outage.
- It is useful to have reusable components written in Python properly packaged, even for the closed source that will never be published publicly. It simplifies code base because packages that are used across the company for different projects do not need to be vendored. You can simply install them from the repository. This simplifies maintenance for such shared code and might reduce development costs for the whole company if it has many teams working on different projects.
- It is a very good practice to have your entire project packaged using `setuptools`. Then, deployment of the new application version is often as simple as running `pip install --update my-application`.

> **Code vendoring**
> Code vendoring is a practice of including sources of the external package in the source code (repository) of other projects. It is usually done when the project's code depends on a specific version of some external package that may also be required by other packages (and in a completely different version).
> For instance, the popular `requests` package uses some version of `urllib3` in its source tree because it is very tightly coupled to it and is very unlikely to work with any other version of `urllib3`. An example of a module that is particularly often used by others is `six`. It can be found in sources of numerous popular projects such as Django (`django.utils.six`), Boto (`boto.vendored.six`), or Matplotlib (`matplotlib.externals.six`).
> Although *vendoring* is practiced even by some large and successful open source projects, it should be avoided if possible. This has justifiable usage only in certain circumstances and should not be treated as a substitute of package dependency management.

In the next section, we will discuss PyPI mirroring.

PyPI mirroring

The problem of PyPI outages can be somehow mitigated by allowing the installation tools to download packages from one of its mirrors. In fact, the official Python Package Index is already served through **Content Delivery Network** (**CDN**), so it is intrinsically mirrored. This does not change the fact that it seems to have some bad days from time to time. Using unofficial mirrors is not a solution here because it might raise some security concerns.

The best solution is to have your own PyPI mirror that will have all of the packages you need. The only party that will use it is you, so it will be much easier to ensure proper availability. The other advantage is that whenever this service goes down, you don't need to rely on someone else to bring it up. The mirroring tool maintained and recommended by PyPA is **bandersnatch** (`https://pypi.python.org/pypi/bandersnatch`). It allows you to mirror the whole content of Python Package Index and it can be provided as the `index-url` option for the repository section in the `.pypirc` file (as explained in the previous chapter). This mirror does not accept uploads and does not have the web part of PyPI. Anyway, beware! A full mirror might require hundreds of gigabytes of storage and its size will continue to grow over time.

But why stop at a simple mirror while we have a much better alternative? There is a very low chance that you will require a mirror of the whole package index. Even with a project that has hundreds of dependencies, it will be only a minor fraction of all of the available packages. Also, not being able to upload your own private package is a huge limitation of such a simple mirror. It seems that the added value of using bandersnatch is very low for such a high price. And this is true in most situations. If the package mirror is to be maintained only for single or a few projects, a much better approach is to use **devpi** (`http://doc.devpi.net/`). It is a PyPI-compatible package index implementation that provides both of the following:

- A private index to upload nonpublic packages
- Index mirroring

The main advantage of devpi over bandersnatch is how it handles mirroring. It can, of course, do a full general mirror of other indexes like bandersnatch does, but it is not its default behavior. Instead of doing a rather expensive backup of the whole repository, it maintains mirrors for packages that were already requested by clients. So, whenever a package is requested by the installation tool (`pip`, `setuptools`, and `easy_install`), if it does not exist in the local mirror, the devpi server will attempt to download it from the mirrored index (usually PyPI) and serve. Once the package is downloaded, the devpi will periodically check for its updates to maintain a fresh state of its mirror.

The mirroring approach leaves a slight risk of failure when you request a new package that was not yet mirrored when the upstream package index has an outage. Anyway, this risk is reduced, thanks to the fact that in most deployments you will depend only on packages that were already mirrored in the index. The mirror state for packages that were already requested has eventual consistency guarantee and new versions will be downloaded automatically. This seems to be a very reasonable trade off.

Now let's see how to properly bundle and build additional non-Python resources in your Python application.

Bundling additional resources with your Python package

Modern web applications have a lot of dependencies and often require a lot of steps to properly install on the remote host. For instance, the typical bootstrapping process for a new version of the application on a remote host consists of the following steps:

1. Create a new virtual environment for isolation.
2. Move the project code to the execution environment.
3. Install the latest project requirements (usually from the `requirements.txt` file).
4. Synchronize or migrate the database schema.
5. Collect static files from project sources and external packages to the desired location.
6. Compile localization files for applications available in different languages.

For more complex sites, there might be lot of additional tasks mostly related to frontend code that is independent from previously defined tasks, as in the following example:

1. Generate CSS files using preprocessors such as SASS or LESS.
2. Perform minification, obfuscation, and/or concatenation of static files (JavaScript and CSS files).

3. Compile code written in JavaScript superset languages (CoffeeScript, TypeScript, and so on) to native JS.
4. Preprocess response template files (minification, style inlining, and so on).

Nowadays, for these kind of applications that require a lot of additional assets to be prepared, most developers would probably use Docker images. Dockerfiles allow you to easily define all of the steps that are necessary to bundle all assets with your application image. But if you don't use Docker, it means that all of these steps must be automated using other tools such as Make, Bash, Fabric, or Ansible. Still, it is not a good idea to do all of these steps directly on the remote hosts where the application is being installed. Here are the reasons:

- Some of the popular tools for processing static assets can be either CPU or memory intensive. Running them in production environments can destabilize your application execution.
- These tools very often will require additional system dependencies that may not be required for the normal operation of your projects. These are mostly additional runtime environments such as JVM, Node, or Ruby. This adds complexity to configuration management and increases the overall maintenance costs.
- If you are deploying your application to multiple servers (tens, hundreds, or thousands), you are simply repeating a lot of work that could be done once. If you have your own infrastructure, then you may not experience the huge increase of costs, especially if you perform deployments in periods of low traffic. But if you run cloud computing services in the pricing model that charges you extra for spikes in load or generally for execution time, then this additional cost may be substantial on a proper scale.
- Most of these steps just take a lot of time. You are installing your code on remote servers, so the last thing you want is to have your connection interrupted by some network issue. By keeping the deployment process quick, you are lowering the chance of deployment interruption.

Obviously, the results of these predeployment steps can't be included in your application code repository either. Simply, there are things that must be done with every release and you can't change that. It is obviously a place for proper automation but the clue is to do it in the right place and at the right time.

Most of the things, such as static collection and code/asset preprocessing, can be done locally or in a dedicated environment, so the actual code that is deployed to the remote server requires only a minimal amount of on-site processing. The following are the most notable of such deployment steps, either in the process of building distribution or installing a package:

1. Installation of Python dependencies and transferring of static assets (CSS files and JavaScript) to the desired location can be handled as a part of the `install` command of the `setup.py` script.

2. Preprocessing of code (processing JavaScript supersets, minification/obfuscation/concatenation of assets, and running SASS or LESS) and things such as localized text compilation (for example, `compilemessages` in Django) can be a part of the `sdist/bdist` command of the `setup.py` script.

Inclusion of preprocessed code other than Python can be easily handled with the proper `MANIFEST.in` file. Dependencies are, of course, best provided as an `install_requires` argument of the `setup()` function call from the `setuptools` package.

Packaging the whole application, of course, will require some additional work from you, such as providing your own custom `setuptools` commands or overriding the existing ones, but it gives you a lot of advantages and makes project deployment a lot faster and reliable.

Let's use a Django-based project (in Django 1.9 version) as an example. I have chosen this framework because it seems to be the most popular Python project of this type, so there is a high chance that you already know it a bit. A typical structure of files in such a project might look like the following:

```
$ tree . -I __pycache__ --dirsfirst
.
├── webxample
│   ├── conf
│   │   ├── __init__.py
│   │   ├── settings.py
│   │   ├── urls.py
│   │   └── wsgi.py
│   ├── locale
│   │   ├── de
│   │   │   └── LC_MESSAGES
│   │   │       └── django.po
│   │   ├── en
│   │   │   └── LC_MESSAGES
│   │   │       └── django.po
```

```
    │   │           └── pl
    │   │               └── LC_MESSAGES
    │   │                   └── django.po
    │   ├── myapp
    │   │   ├── migrations
    │   │   │   └── __init__.py
    │   │   ├── static
    │   │   │   ├── js
    │   │   │   │   └── myapp.js
    │   │   │   └── sass
    │   │   │       └── myapp.scss
    │   │   ├── templates
    │   │   │   ├── index.html
    │   │   │   └── some_view.html
    │   │   ├── __init__.py
    │   │   ├── admin.py
    │   │   ├── apps.py
    │   │   ├── models.py
    │   │   ├── tests.py
    │   │   └── views.py
    │   ├── __init__.py
    │   └── manage.py
    ├── MANIFEST.in
    ├── README.md
    └── setup.py
  15 directories, 23 files
```

Note that this slightly differs from the usual Django project template. By default, the name of the package that contains the WSGI application, the settings module, and the URL configuration has the same name as the project. Because we decided to take the packaging approach, this would be named as `webxample`. This can cause some confusion, so it is better to rename it to `conf`. Without digging into the possible implementation details, let's just make the following few simple assumptions:

- Our example application has some external dependencies. Here, it will be two popular Django packages: `djangorestframework` and `django-allauth`, plus one non-Django package: `gunicorn`.
- `djangorestframework` and `django-allauth` are provided as `INSTALLED_APPS` in the `webexample.webexample.settings` module.
- The application is localized in three languages (German, English, and Polish) but we don't want to store the compiled `gettext` messages in the repository.
- We are tired of vanilla CSS syntax, so we decided to use a more powerful SCSS language that we translate into CSS using SASS.

Knowing the structure of the project, we can write our `setup.py` script in a way that makes `setuptools` handle the following:

- Compilation of SCSS files under `webxample/myapp/static/scss`
- Compilation of `gettext` messages under `webexample/locale` from `.po` to `.mo` format
- Installation of the requirements
- A new script that provides an entry point to the package, so we will have the custom command instead of the `manage.py` script

We have a bit of luck here—Python binding for `libsass`, a C/C++ port of the SASS engine, provides some integration with `setuptools` and `distutils`. With only a little configuration, it provides a custom `setup.py` command for running the SASS compilation. This is shown in the following code:

```
from setuptools import setup

setup(
    name='webxample',
    setup_requires=['libsass == 0.6.0'],
    sass_manifests={
        'webxample.myapp': ('static/sass', 'static/css')
    },
)
```

So, instead of running the `sass` command manually or executing a subprocess in the `setup.py` script, we can type `python setup.py build_scss` and have our SCSS files compiled to CSS. This is still not enough. It makes our life a bit easier but we want the whole distribution fully automated so there is only one step for creating new releases. To achieve this goal, we are forced to override some of the existing `setuptools` distribution commands.

The example `setup.py` file that handles some of the project preparation steps through packaging might look like this:

```
import os

from setuptools import setup
from setuptools import find_packages
from distutils.cmd import Command
from distutils.command.build import build as _build

try:
    from django.core.management.commands.compilemessages \
```

```python
        import Command as CompileCommand
except ImportError:
    # note: during installation django may not be available
    CompileCommand = None

# this environment is requires
os.environ.setdefault(
    "DJANGO_SETTINGS_MODULE", "webxample.conf.settings"
)

class build_messages(Command):
    """ Custom command for building gettext messages in Django
    """
    description = """compile gettext messages"""
    user_options = []

    def initialize_options(self):
        pass

    def finalize_options(self):

        pass

    def run(self):
        if CompileCommand:
            CompileCommand().handle(
                verbosity=2, locales=[], exclude=[]
            )
        else:
            raise RuntimeError("could not build translations")

class build(_build):
    """ Overriden build command that adds additional build steps
    """
    sub_commands = [
        ('build_messages', None),
        ('build_sass', None),
    ] + _build.sub_commands

setup(
    name='webxample',
    setup_requires=[
        'libsass == 0.6.0',
        'django == 1.9.2',
    ],
    install_requires=[
        'django == 1.9.2',
        'gunicorn == 19.4.5',
```

```
            'djangorestframework == 3.3.2',
            'django-allauth == 0.24.1',
        ],
    packages=find_packages('.'),
    sass_manifests={
        'webxample.myapp': ('static/sass', 'static/css')
    },
    cmdclass={
        'build_messages': build_messages,
        'build': build,
    },
    entry_points={
        'console_scripts': {
            'webxample = webxample.manage:main',
        }
    }
)
```

With such an implementation, we can build all assets and create the source distribution of a package for the `webxample` project using the following single Terminal command:

```
$ python setup.py build sdist
```

If you already have your own package index (created with devpi), you can add the `install` subcommand or use `twine` so this package will be available for installation with `pip` in your organization. If we look into a structure of source distribution created with our `setup.py` script, we can see that it contains the following compiled `gettext` messages and CSS style sheets generated from SCSS files:

```
$ tar -xvzf dist/webxample-0.0.0.tar.gz 2> /dev/null
$ tree webxample-0.0.0/ -I __pycache__ --dirsfirst
webxample-0.0.0/
├── webxample
│   ├── conf
│   │   ├── __init__.py
│   │   ├── settings.py
│   │   ├── urls.py
│   │   └── wsgi.py
│   ├── locale
│   │   ├── de
│   │   │   └── LC_MESSAGES
│   │   │       ├── django.mo
│   │   │       └── django.po
│   │   ├── en
│   │   │   └── LC_MESSAGES
│   │   │       ├── django.mo
│   │   │       └── django.po
```

```
│   │       └── pl
│   │           └── LC_MESSAGES
│   │               ├── django.mo
│   │               └── django.po
│   ├── myapp
│   │   ├── migrations
│   │   │   └── __init__.py
│   │   ├── static
│   │   │   ├── css
│   │   │   │   └── myapp.scss.css
│   │   │   └── js
│   │   │       └── myapp.js
│   │   ├── templates
│   │   │   ├── index.html
│   │   │   └── some_view.html
│   │   ├── __init__.py
│   │   ├── admin.py
│   │   ├── apps.py
│   │   ├── models.py
│   │   ├── tests.py
│   │   └── views.py
│   ├── __init__.py
│   └── manage.py
├── webxample.egg-info
│   ├── PKG-INFO
│   ├── SOURCES.txt
│   ├── dependency_links.txt
│   ├── requires.txt
│   └── top_level.txt
├── MANIFEST.in
├── PKG-INFO
├── README.md
├── setup.cfg
└── setup.py

16 directories, 33 files
```

The additional benefit of using this approach is that we were able to provide our own entry point for the project in place of Django's default `manage.py` script. Now, we can run any Django management command using this entry point, for instance:

```
$ webxample migrate
$ webxample collectstatic
$ webxample runserver
```

This required a little change in the `manage.py` script for compatibility with
the `entry_points` argument in `setup()`, so the main part of its code is wrapped with
the `main()` function call. This is shown in the following code:

```python
#!/usr/bin/env python3
import os
import sys


def main():
    os.environ.setdefault(
        "DJANGO_SETTINGS_MODULE", "webxample.conf.settings"
    )

    from django.core.management import execute_from_command_line

    execute_from_command_line(sys.argv)


if __name__ == "__main__":
    main()
```

Unfortunately, a lot of frameworks (including Django) are not designed with the idea of
packaging your projects that way in mind. It means that, depending on the advancement of
your application, converting it to a package may require a lot of changes. In Django, this
often means rewriting many of the implicit imports and updating a lot of configuration
variables in your settings file.

The other problem here is consistency of releases created using Python packaging. If
different team members are authorized to create application distribution, it is crucial that
this process takes place in the same replicable environment. Especially when you do a lot of
asset preprocessing, it is possible that the package created in two different environments
will not look the same, even if it is created from the same code base. This may be due to
different versions of tools used during the build process. The best practice is to move the
distribution responsibility to some continuous integration/delivery system such as Jenkins,
Buildbot, Travis CI, or similar. The additional advantage is that you can assert that the
package passes all of the required tests before going to distribution. You can even make the
automated deployment as a part of such a continuous delivery system.

Mind that although distributing your code as Python packages using `setuptools` might seem elegant, it is actually not simple and effortless. It has potential to greatly simplify your deployments and so it is definitely worth trying but it comes with the cost of increased complexity. If your preprocessing pipeline for your application grows too complex, you should definitely consider building Docker images and deploying your application as containers.

Deployment with Docker requires some additional setup and orchestration but in the long term saves a lot of time and resources that are otherwise required to maintain repeatable build environments and complex preprocessing pipelines.

In the next section, we'll take a look at the common conventions and practices regarding deployment of Python applications.

Common conventions and practices

There are a set of common conventions and practices for deployment that not every developer may know but are obvious for anyone who did some operations in their life. As explained in this chapter's introduction, it is crucial to know at least a few of them, even if you are not responsible for code deployment and operations, because it will allow you to make better design decisions during the development.

Let's take a look at the filesystem hierarchy in the next section.

The filesystem hierarchy

The most obvious conventions that may come into your mind are probably about filesystem hierarchy and user naming. If you are looking for such suggestions here, then you will be disappointed. There is, of course, a **Filesystem Hierarchy Standard** (**FHS**) that defines the directory structure and directory contents in Unix and Unix-like operating systems, but it is really hard to find the actual OS distribution that is fully compliant with FHS. If system designers and programmers cannot obey such standards, it is very hard to expect the same from its administrators. During my experience, I've seen application code deployed almost everywhere it is possible, including nonstandard custom directories in the root filesystem level. Almost always the people behind such decisions had really strong arguments for doing so. The only suggestions in this matter that I can give you are as follows:

- Choose wisely and avoid surprises.
- Be consistent across all of the available infrastructure of your project.
- Try to be consistent across your organization (the company you work in).

What really helps is to document conventions for your project. Just remember to make sure that this documentation is accessible for every interested team member and that everyone knows that such a document exists.

In the next section, we will discuss isolation.

Isolation

Reasons for isolation as well as recommended tools were already discussed in `Chapter 2`, *Modern Python Development Environments*. These are: better environment reproducibility and solving the inevitable problems of dependency conflicts. For the purpose of deployments, there is only one important thing to add. You should always isolate project dependencies for each release of your application. In practice, it means that, whenever you deploy a new version of the application, you should create a new isolated environment for this release (using `virtualenv` or `venv`). Old environments should be left for some time on your hosts, so that, in case of issues, you can easily perform a rollback to one of the older versions of your application.

Creating fresh environments for each release helps in managing their clean state and compliance with a list of provided dependencies. By fresh environment we mean creating a new directory tree in the filesystem instead of updating already existing files. Unfortunately, it may make it a bit harder to perform things such as the graceful reload of services, which is much easier to achieve if the environment is updated in place.

The next section shows how to use process supervision tools.

Using process supervision tools

Applications on remote servers are never usually expected to quit. If it is a web application, its HTTP server process will indefinitely wait for new connections and requests and will exit only if some unrecoverable error occurs.

It is, of course, not possible to run it manually in shell and have a never-ending SSH connection. Using `nohup`, `screen`, or `tmux` to semi-daemonize the process is not an option. Doing so is like designing your service to fail.

What you need is to have some process supervision tool that can start and manage your application process. Before choosing the right one, you need to make sure it does the following things:

- Restarts the service if it quits
- Reliably tracks its state
- Captures its `stdout`/`stderr` streams for logging purposes
- Runs a process with specific user/group permissions
- Configures system environment variables

Most of the Unix and Linux distributions have some built-in tools/subsystems for process supervision such as `initd` scripts, `upstart`, and `runit`. Unfortunately, in most cases, they are not well suited for running user-level application code and are really hard to maintain. In particular, writing reliable `init.d` scripts is a real challenge because it requires a lot of Bash scripting that is hard to do right. Some Linux distributions such as Gentoo have a redesigned approach to `init.d` scripts, so writing them is a lot easier. Anyway, locking yourself to a specific OS distribution just for the purpose of a single process supervision tool is not a good idea.

Two popular tools in the Python community for managing application processes are Supervisor (`http://supervisord.org`) and Circus (`https://circus.readthedocs.org/en/latest/`). They are both very similar in configuration and usage. Circus is a bit younger than Supervisor because it was created to address some weaknesses of the latter. They both can be configured in simple INI-like configuration format. They are not limited to running Python processes and can be configured to manage any application. It is hard to say which one is better because they both provide very similar functionality. Anyway, Supervisor does not run on Python 3, so it does not get our approval. While it is not a problem to run Python 3 processes under Supervisor's control, I will take it as an excuse and feature only the example of the Circus configuration.

Let's assume that we want to run the `webxample` application (presented previously in this chapter) using `gunicorn` webserver under Circus control. In production, we would probably run Circus under an applicable system-level process supervision tool (`initd`, `upstart`, and `runit`), especially if it was installed from the system packages repository. For the sake of simplicity, we will run this locally inside of the virtual environment. The minimal configuration file (here named `circus.ini`) that allows us to run our application in Circus looks like this:

```
[watcher:webxample]
cmd = /path/to/venv/dir/bin/gunicorn webxample.conf.wsgi:application
numprocesses = 1
```

Now, the `circus` process can be run with this configuration file as the execution argument:

```
$ circusd circus.ini
2016-02-15 08:34:34 circus[1776] [INFO] Starting master on pid 1776
2016-02-15 08:34:34 circus[1776] [INFO] Arbiter now waiting for commands
2016-02-15 08:34:34 circus[1776] [INFO] webxample started
[2016-02-15 08:34:34 +0100] [1778] [INFO] Starting gunicorn 19.4.5
[2016-02-15 08:34:34 +0100] [1778] [INFO] Listening at:
http://127.0.0.1:8000 (1778)
[2016-02-15 08:34:34 +0100] [1778] [INFO] Using worker: sync
[2016-02-15 08:34:34 +0100] [1781] [INFO] Booting worker with pid: 1781
```

Now, you can use the `circusctl` command to run an interactive session and control all managed processes using simple commands. Here is an example of such a session:

```
$ circusctl
circusctl 0.13.0
webxample: active
(circusctl) stop webxample
ok
(circusctl) status
webxample: stopped
(circusctl) start webxample
ok
(circusctl) status
webxample: active
```

Of course, both of the mentioned tools have a lot more features available. All of them are explained in their documentation, so before making your choice, you should read them carefully.

The next section discusses the importance of running application code in user space.

Application code running in user space

Your application code should be always run in user space. This means it must not be executed under super-user privileges. If you design your application following the Twelve-Factor App, it is possible to run your application under a user that has almost no privileges. The conventional name for the user that owns no files and is in no privileged groups is `nobody`; anyway, the actual recommendation is to create a separate user for each application daemon. The reason for that is system security. It is to limit the damage that a malicious user can do if it gains control over your application process. In Linux, processes of the same user can interact with each other, so it is important to have different applications separated at the user level.

The next section shows how to use reverse HTTP proxies.

Using reverse HTTP proxies

Multiple Python WSGI-compliant web servers can easily serve HTTP traffic all by themselves without the need of any other web server on top of them. It is still very common to hide them behind a reverse proxy such as NGINX or Apache. A reverse proxy creates an additional HTTP server layer that proxies requests and responses between clients and your application and appears to your Python server as though it is the requesting client. Reverse proxies are useful for the following variety of reasons:

- TLS/SSL termination is usually better handled by top-level web servers such as NGINX and Apache. This allows the Python application to speak only simple HTTP protocol (instead of HTTPS), so complexity and configuration of secure communication channels are left for the reverse proxy.
- Unprivileged users cannot bind low ports (in the range of 0-1000), but the HTTP protocol should be served to the users on port 80, and HTTPS should be served on port 443. To do this, you must run the process with super-user privileges. Usually, it is safer to have your application serving on a high port or on a Unix domain socket and use that as an upstream for reverse proxy that is run under the more privileged user.
- Usually, NGINX can serve static assets (images, JS, CSS, and other media) more efficiently than Python code. If you configure it as a reverse proxy, then it is only a few more lines of configuration to serve static files through it.
- When a single host needs to serve multiple applications from different domains, Apache or NGINX are indispensable for creating virtual hosts for different domains served on the same port.
- Reverse proxies can improve performance by adding additional caching layers or can be configured as simple load balancers. Reverse proxies can also apply compression (for example, gzip) to responses in order to limit the amount of required network bandwidth.

Some of the web servers actually are recommended to be run behind a proxy such as NGINX. For example, `gunicorn` is a very robust WSGI-based server that can give exceptional performance results if its clients are fast as well. On the other hand, it does not handle slow clients well, so it is easily susceptible to the denial of service attacks based on a slow client connection. Using a proxy server that is able to buffer slow clients is the best way to solve this problem.

Mind that, with proper infrastructure, it is possible to almost completely get rid of reverse proxies in your architecture. Nowadays, things such as SSL termination and compression can be easily handled with load balancing services such as AWS Load Balancer. Static and media assets are also better served through **Content Delivery Networks** (**CDNs**) that can also be used to cache other responses of your service.

The mentioned requirement to serve HTTP/HTTPS traffic on low 80/433 ports (that cannot be bound by unprivileged users) is also no longer a problem if the only entry points that your clients communicate with are your load balancers and CDN. Still, even with that kind of architecture, it does not necessarily mean that your system does not facilitate reverse proxies at all. For instance, many load balancers support proxy protocol. It means that a load balancer may appear to your application as though it is the requesting client. In such scenarios, the load balancer acts as it were in fact a reverse proxy.

Process reloading is discussed in the next section.

Reloading processes gracefully

The ninth rule of Twelve-Factor App methodology deals with process disposability and says that you should maximize robustness with fast start up times and graceful shutdowns. While fast start up time is quite self-explanatory, the graceful shutdowns require some additional discussion.

In the scope of web applications, if you terminate the server process in a non-graceful way, it will quit immediately without the time to finish processing requests and reply with proper responses to connected clients. In the best scenario case, if you use some kind of reverse proxy, then the proxy might reply to the connected clients with some generic error response (for example, 502 Bad Gateway), even though it is not the right way to notify users that you have restarted your application and have deployed a new release.

According to the Twelve-Factor App, the web serving process should be able to quit gracefully upon receiving the Unix `SIGTERM` signal. This means the server should stop accepting new connections, finish processing all of the pending requests, and then quit with some exit code when there is nothing more to do.

Obviously, when all of the serving processes quit or start their shutdown procedure, you are not able to process new requests any longer. This means your service will still experience an outage. So there is an additional step you need to perform—start new workers that will be able to accept new connections while the old ones are gracefully quitting. Various Python WSGI-compliant web server implementations allow you to reload the service gracefully without any downtime.

The most popular Python web servers are Gunicorn and uWSGI, which provide the following functionality:

- Gunicorn's master process upon receiving the `SIGHUP` signal (`kill -HUP <process-pid>`) will start new workers (with new code and configuration) and attempt a graceful shutdown on the old ones.
- uWSGI has at least three independent schemes for doing graceful reloads. Each of them is too complex to explain briefly, but its official documentation provides full information on all of the possible options.

Today, graceful reloads are a standard in deploying web applications. Gunicorn seems to have an approach that is the easiest to use but also leaves you with the least flexibility. Graceful reloads in uWSGI on the other hand allow much better control on reloads but require more effort to automate and set up. Also, how you handle graceful reloads in your automated deploys is also affected by what supervision tools you use and how they are configured. For instance, in Gunicorn, graceful reloads are as simple as the following:

```
kill -HUP <gunicorn-master-process-pid>
```

But, if you want to properly isolate project distributions by separating virtual environments for each release and configure process supervision using symbolic links (as presented in the `fabfile` example earlier), you will shortly notice that this feature of Gunicorn may not work as expected. For more complex deployments, there is still no system-level solution available that will work for you out-of-the-box. You will always have to do a bit of hacking and sometimes this will require a substantial level of knowledge about low-level system implementation details.

In such complex scenarios, it is usually better to solve the problem on a higher level of abstraction. If you finally decide to run your applications as containers and distribute new releases as new container images (it is strongly advised), then you can leave the responsibility of graceful reloads to your container orchestration system of choice (for example, Kubernetes) that can usually handle various reloading strategies out-of-the-box.

Even without advanced container orchestration systems, you can do graceful reloading on the infrastructure level. For instance, AWS Elastic Load Balancer is able to gracefully switch traffic from your old application instances (for example, EC2 hosts) to new ones. Once old application instances receive no new traffic and are done handling their requests, they can be simply terminated without any observable outage to your service. Other cloud providers, of course, usually provide analogous features in their service portfolio.

The next section discusses code instrumentation and monitoring.

Code instrumentation and monitoring

Our work does not end on writing an application and deploying it to target the execution environment. It is possible to write an application, which after deployment will not require any further maintenance, although it is very unlikely. In reality, we need to ensure that it is properly observed for errors and performance.

To be sure that your product works as expected, you need to properly handle application logs and monitor the necessary application metrics. This often includes the following:

- Monitoring web application access logs for various HTTP status codes
- A collection of process logs that may contain information about runtime errors and various warnings
- Monitoring usage of system resources (CPU load, memory, network traffic, I/O performance, disk usage, and so on) on the remote hosts where the application is run
- Monitoring application-level performance and metrics that are business performance indicators (customer acquisition, revenue, conversion rates, and so on)

Luckily, there are a lot of free tools available for instrumenting your code and monitoring its performance. Most of them are very easy to integrate.

Logging errors with Sentry/Raven is explained in the next section.

Logging errors – Sentry/Raven

The truth is painful. No matter how precisely your application is tested, your code will eventually fail at some point. This can be anything—unexpected exception, resource exhaustion, crash of some backing service, network outage, or simply an issue in the external library. Some of the possible issues (such as resource exhaustion) can be predicted and prevented in advance with proper monitoring. Unfortunately, there will always be something that passes your defenses, no matter how much you try.

What you can do instead is to prepare for such scenarios and make sure that no error passes unnoticed. In most cases, any unexpected failure scenario results in an exception raised by the application and logged through the logging system. This can be `stdout`, `stderr`, log file, or whatever output you have configured for logging. Depending on your implementation, this may or may not result in the application quitting with some system exit code.

You could, of course, depend solely on the log files stored in the filesystem for finding and monitoring your application errors. Unfortunately, observing errors in plain textual form is quite painful and does not scale well beyond anything more complex than running code in development. You will eventually be forced to use some services designed for log collection and analysis. Proper log processing is very important for other reasons (that will be explained a bit later) but does not work well for tracking and debugging errors. The reason is simple. The most common form of error logs is just Python stack trace. If you stop only on that, you will shortly realize that it is not enough in finding the root cause of your issues. This is especially true when errors occur in unknown patterns or in certain load conditions.

What you really need is as much context information about the error occurrence as possible. It is also very useful to have a full history of the errors that occurred in the production environment that you can browse and search in some convenient way.

One of the most common tools that gives such capabilities is Sentry (`https://getsentry.com`). 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 web developers. Now, it outgrew its initial ambitions and has support for many more languages, including PHP, Ruby, and JavaScript but still stays the most popular tool of choice for many Python web developers.

> **Exception stack tracebacks in web applications**
> It is common that 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 is available as a paid software-as-a-service model, but it is open source, 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 but have no access to your own Sentry server, then you can easily sign up for a free trial on Sentry's on-premise service site. Once you have access to a Sentry server and have created a new project, you will obtain a string called **Data Source Name** (**DSN**). This DSN string is the minimal configuration setting needed to integrate your application with sentry. It contains protocol, credentials, server location, and your organization/project identifier in the following form:

```
'{PROTOCOL}://{PUBLIC_KEY}:{SECRET_KEY}@{HOST}/{PATH}{PROJECT_ID}'
```

Once you have 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>'
)

try:
    1 / 0
except Exception as e:
    sentry_sdk.capture_exception(e)
```

> **Sentry and Raven**
> The old library for Sentry integration is Raven. It is still maintained and available on PyPI but is being phased out, so it is best to start your Sentry integration using the newer `python-sdk` package. It is possible though that some framework integrations or Raven extensions haven't been ported to new SDK, so in such situations, integration using Raven is still a feasible integration path.

Sentry SDK has numerous integrations with most popular Python frameworks such as Django, Flask, Celery, or Pyramid to make integration easier. These integrations will automatically provide additional context that is specific to the given framework. If your web framework of choice does not have a dedicated support, the `sentry-sdk` package provides generic WSGI middleware that makes it compatible with any WSGI-based web servers, 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>'
)

# ...

# note: application is some WSGI application object defined earlier
application = SentryWsgiMiddleware(application)
```

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

sentry_logging = LoggingIntegration(
    level=logging.INFO,
    event_level=logging.ERROR,
)

sentry_sdk.init(
    dsn='https://<key>:<secret>@app.getsentry.com/<project>',
    integrations=[sentry_logging],
)
```

Capturing of `logging` messages may have caveats, so make sure to read the official documentation on that topic if you are interested in such a feature. This should save you from unpleasant surprises.

The last note is about running your own Sentry as a way to save some money. *There ain't no such thing as a free lunch.* You will eventually pay additional infrastructure costs and Sentry will be just another service to maintain. *Maintenance = additional work = costs*! As your application grows, the number of exceptions grow, so you will be forced to scale Sentry as you scale your product. Fortunately, this is a very robust project, but will not give you any value if overwhelmed with too much load. Also, keeping Sentry prepared for a catastrophic failure scenario where thousands of crash reports per second can be sent 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 such dilemma if security policies in your organization deny sending any data to third parties. If so, just host it on your own infrastructure. There are costs, of course, but ones that are definitely worth paying.

Next, we will discuss the monitoring system and application metrics.

Monitoring system and application metrics

When it comes to monitoring performance, the amount of tools to choose from may be overwhelming. If you have high expectations, then it is possible that you will need to use a few of them at the same time.

Munin (`http://munin-monitoring.org`) is one of the popular choices used by many organizations regardless of the technology stack they use. It is a great tool for analyzing resource trends and provides a lot of useful information, even with a default installation without additional configuration. Its installation consists of the following two main components:

- The Munin master that collects metrics from other nodes and serves metrics graphs
- The Munin node that is installed on a monitored host, which gathers local metrics and sends it to the Munin master

The master node and most of the plugins are written in Perl. There are also node implementations in other languages: `munin-node-c` is written in C (`https://github.com/munin-monitoring/munin-c`) and `munin-node-python` is written in Python (`https://github.com/agroszer/munin-node-python`). Munin comes with a huge number of plugins available in its `contrib` repository. This means it provides out-of-the-box support for most of the popular databases and system services. There are even plugins for monitoring popular Python web servers, such as uWSGI or Gunicorn.

The main drawback of Munin is the fact that it serves graphs as static images and actual plotting configuration is included in specific plugin configurations. This does not help in creating flexible monitoring dashboards and comparing metric values from different sources at the same graph. But this is the price we need to pay for simple installation and versatility. Writing your own plugins is quite simple. There is the `munin-python` package (`http://python-munin.readthedocs.org/en/latest/`) that helps to write Munin plugins in Python.

Unfortunately, the architecture of Munin that assumes that there is always a separate monitoring daemon process on every host that is responsible for collection of metrics may not be the best solution for monitoring custom application performance metrics. It is indeed very easy to write your own Munin plugins, but under the assumption that the monitoring process can already report its performance statistics in some way.

If you want to collect some custom application-level metrics, it might be necessary to aggregate and store them in some temporary storage until reporting to a custom Munin plugin. It makes creation of custom metrics more complicated, so you might want to consider other solutions for such purposes.

The other popular solution that makes it especially easy to collect custom metrics is StatsD (`https://github.com/etsy/statsd`). It's a network daemon written in Node.js that listens to various statistics such as counters, timers, and gauges. It is very easy to integrate, thanks to the simple protocol based on UDP. It is also easy to use the Python package named `statsd` for sending metrics to the StatsD daemon, as follows:

```
>>> import statsd
>>> c = statsd.StatsClient('localhost', 8125)
>>> c.incr('foo')  # Increment the 'foo' counter.
>>> c.timing('stats.timed', 320)  # Record a 320ms 'stats.timed'.
```

Because UDP is a connectionless protocol, it has a very low performance overhead on the application code, so it is very suitable for tracking and measuring custom events inside the application code.

Unfortunately, StatsD is the only metrics collection daemon, so it does not provide any reporting features. You need other processes that are able to process data from StatsD in order to see the actual metrics graphs. The most popular choice is Graphite (`http://graphite.readthedocs.org`). It does mainly the following two things:

- Stores numeric time-series data
- Renders graphs of this data on demand

Graphite provides you with the ability to save graph presets that are highly customizable. You can also group many graphs into thematic dashboards. Graphs are, similar to Munin, rendered as static images, but there is also the JSON API that allows other frontends to read graph data and render it by other means.

One of the great dashboard plugins integrated with Graphite is Grafana (`http://grafana.org`). It is really worth trying because it has way better usability than plain Graphite dashboards. Graphs provided in Grafana are fully interactive and easier to manage.

Graphite is unfortunately a bit of a complex project. It is not a monolithic service and consists of the following three separate components:

- **Carbon**: This is a daemon written using Twisted that listens for time-series data.
- **whisper**: This is a simple database library for storing time-series data.
- **graphite webapp**: This is a Django web application that renders graphs on-demand as static images (using Cairo library) or as JSON data.

When used with the StatsD project, the `statsd` daemon sends its data to the `carbon` daemon. This makes the full solution a rather complex stack of various applications, where each of them is written using completely different technology. Also, there are no preconfigured graphs, plugins, and dashboards available, so you will need to configure everything by yourself. This is a lot of work at the beginning and it is very easy to miss something important. This is the reason why it might be a good idea to use Munin as a monitoring backup, even if you decide to have Graphite as your core monitoring service.

Another good monitoring solution for arbitrary metric collection is Prometheus. It has a completely different architecture than Munin and StatsD. Instead of relying on monitored applications or daemons to push metrics in configured intervals, Prometheus actively pulls metrics directly from the source using the HTTP protocol. This requires monitored services to store (and sometimes preprocess) metrics internally and expose them on HTTP endpoints.

Fortunately, Prometheus comes with a handful of libraries for various languages and frameworks to make this kind of integration as easy as possible. There are also various exporters that act as bridges between Prometheus and other monitoring systems. So, if you already use other monitoring solutions, it is usually very easy to migrate gradually to a Prometheus architecture. Prometheus also wonderfully integrates with Grafana.

In the next section, we will see how to deal with application logs.

Dealing with application logs

While solutions such as Sentry are usually way more powerful than ordinary textual output stored in files, logs will never die. Writing some information to a standard output or file is one of the simplest things that an application can do and this should never be underestimated. There is a risk that messages sent to Sentry by Raven will not get delivered. The network can fail. Sentry's storage can get exhausted or may not be able to handle the incoming load. Your application might crash before any message is sent (with a segmentation fault, for example). These are only a few of the possible scenarios.

What is less likely is that your application won't be able to log messages that are going to be written to the filesystem. It is still possible, but let's be honest, if you face such a condition where logging fails, probably you have a lot more burning issues than some missing log messages.

Remember that logs are not only about errors. Many developers used to think about logs only as a source of data that is useful when debugging issues and/or that can be used to perform some kind of forensics.

Definitely, less of them try to use it as a source for generating application metrics or to do some statistical analysis. But logs may be a lot more useful than that. They can even be a core of the product implementation. A great example of building a product with logs is Amazon's article presenting example architecture for the real-time bidding service, where everything is centered around access log collection and processing.
See `https://aws.amazon.com/blogs/aws/real-time-ad-impression-bids-using-dynamodb/`

Let's discuss the basic low-level log practices.

Basic low-level log practices

The Twelve-Factor App manifesto says that logs should be treated as event streams. So, the log file is not a log by itself, but only an output format. The fact that they are streams means they represent time ordered events. In raw, they are typically in a plaintext format with one line per event, although in some cases they may span across multiple lines (this is typical for any back traces related to runtime errors).

According to the Twelve-Factor App methodology, the application should never be aware of the format in which logs are stored. This means that writing to the file, or log rotation and retention should never be maintained by the application code.

These are the responsibilities of the environment in which the applications is run. This may be confusing because a lot of frameworks provide functions and classes for managing log files as well as rotation, compression, and retention utilities. It is tempting to use them because everything can be contained in your application code base, but actually it is an anti-pattern that should be avoided.

The best practices for dealing with logs are as follows:

- The application should always write logs unbuffered to the standard output (`stdout`).
- The execution environment should be responsible for collection and routing of logs to the final destination.

The main part of the mentioned execution environment is usually some kind of process supervision tool. The popular Python solutions, such as Supervisor or Circus, are the first ones responsible for dealing with log collection and routing. If logs are to be stored in the local filesystem, then only they should write to actual log files.

Both Supervisor and Circus are also capable of handling log rotation and retention for managed processes but you should really consider whether this is a path that you want to take. Successful operations are mostly about simplicity and consistency. Logs of your own application are probably not the only ones that you want to process and archive. If you use Apache or NGINX as a reverse proxy, you might want to collect their access logs.

You might also want to store and process logs for caches and databases. If you are running some popular Linux distribution, then the chances are very high that each of these services have their own log files processed (rotated, compressed, and so on) by the popular utility named `logrotate`. My strong recommendation is to forget about Supervisor's and Circus' log rotation capabilities for the sake of consistency with other system services. `logrotate` is way more configurable and also supports compression.

> **logrotate and Supervisor/Circus**
> There is an important thing to know when using `logrotate` with Supervisor or Circus. Rotation of logs will always happen while process Supervisor still has open descriptor to rotated logs. If you don't take proper countermeasures, then new events will be still written to the file descriptor that was already deleted by `logrotate`. As a result, nothing more will be stored in a filesystem. Solutions to this problem are quite simple. Configure `logrotate` for log files of processes managed by Supervisor or Circus with the `copytruncate` option. Instead of moving the log file after rotation, it will copy it and truncate the original file to zero size in place. This approach does not invalidate any of the existing file descriptors and processes that are already running can write to log files uninterrupted. Supervisor can also accept the `SIGUSR2` signal that will make it reopen all of the file descriptors. It may be included as the `postrotate` script in the `logrotate` configuration. This second approach is more economical in the terms of I/O operations, but is also less reliable and harder to maintain.

Different tools for log processing are explained in the next section.

Tools for log processing

If you have no experience in working with big amounts of logs, you will eventually gain it when working with a product that has some substantial load. You will shortly notice that a simple approach based on storing them in files and backing them up in some persistent storage for later retrieval is not enough. Without proper tools, this will become crude and expensive. Simple utilities such as `logrotate` help you only to ensure that the hard disk is not overloaded by the ever-increasing amount of new events, although splitting and compressing log files only helps in the data archival process but does not make data retrieval or analysis simpler.

When working with distributed systems that span across multiple nodes, it is nice to have a single central point from which all logs can be retrieved and analyzed. This requires a log processing flow that goes way beyond simple compression and backing up. Fortunately, this is a well-known problem so there are many tools available that aim to solve it.

One of the popular choices among many developers is **Logstash**. This is the log collection daemon that can observe active log files, parse log entries, and send them to the backing service in a structured form. The choice of backing stays almost always the same—**Elasticsearch**. Elasticsearch is the search engine built on top of Lucene. Among text search capabilities, it has a unique data aggregation framework that fits extremely well into the purpose of log analysis. The other addition to this pair of tools is **Kibana**. It is a very versatile monitoring, analysis, and visualization platform for Elasticsearch. The way that these three tools complement each other is the reason why almost always they are used together as a single stack for log processing.

The integration of existing services with Logstash is very simple because it can listen on existing log file changes for the new events with only minimal changes in your logging configuration. It parses logs in textual form and has preconfigured support for some of the popular log formats, such as Apache/NGINX access logs. Logstash can be complemented with Beats. Beats are log shippers compatible with Logstash input protocols that can collect not only raw log data from files (Filebeat) but also various system metrics (Metricbeat) and even audit user activities on hosts (Auditbeat).

The other solution that seems to fill some of Logstash gaps is Fluentd. It is an alternative log collection daemon that can be used interchangeably with Logstash in the mentioned log monitoring stack. It also has an option to listen and parse log events directly in log files, so integration requires only a little effort. In contrast to Logstash, it handles reloads very well and even does not need to be signaled if log files were rotated. Anyway, the most advantage comes from using one of its alternative log collection options that will require some substantial changes to logging configuration in your application.

Fluentd really treats logs as event streams (as recommended by the Twelve-Factor App). The file-based integration is still possible but it is only kind of backward compatible for legacy applications that treat logs mainly as files. Every log entry is an event and it should be structured. Fluentd can parse textual logs and has multiple plugin options to handle, including the following:

- Common formats (Apache, NGINX, and syslog)
- Arbitrary formats specified using regular expressions or handled with custom parsing plugins
- Generic formats for structured messages such as JSON

The best event format for Fluentd is JSON because it adds the least amount of overhead. Messages in JSON can also be passed almost without any change to the backing service such as Elasticsearch or the database.

The other very useful feature of Fluentd is the ability to pass event streams using transports other than a log file written to the disk. The following are the most notable built-in input plugins:

- `in_udp`: With this plugin, every log event is sent as UDP packets.
- `in_tcp`: With this plugin, events are sent through TCP connection.
- `in_unix`: With this plugin, events are sent through a Unix domain socket (named socket).
- `in_http`: With this plugin, events are sent as HTTP POST requests.
- `in_exec`: With this plugin, Fluentd process executes an external command periodically to pull events in the JSON or MessagePack format.
- `in_tail`: With this plugin, Fluentd process listens for an event in a textual file.

Alternative transports for log events may be especially useful in situations where you need to deal with poor I/O performance of machine storage. It is very often on cloud computing services that the default disk storage has a very low number of I**nput Output Operations Per Second** (**IOPS**) and you need to pay a lot of money for better disk performance.

If your application outputs a large amount of log messages, you can easily saturate your I/O capabilities, even if the data size is not very high. With alternate transports, you can use your hardware more efficiently because you leave the responsibility of data buffering only to a single process-log collector. When configured to buffer messages in memory instead of disk, you can even completely get rid of disk writes for logs, although this may greatly reduce the consistency guarantees of collected logs.

Using different transports seems to be slightly against the 11th rule of the Twelve-Factor App methodology. Treating logs as event streams when explained in detail suggests that the application should always log only through a single standard output stream (`stdout`). It is still possible to use alternate transports without breaking this rule. Writing to `stdout` does not necessarily mean that this stream must be written to file.

You can leave your application logging that way and wrap it with an external process that will capture this stream and pass it directly to Logstash or Fluentd without engaging the filesystem. This is an advanced pattern that may not be suitable for every project. It has the obvious disadvantage of higher complexity, so you need to consider for yourself whether it is really worth doing.

Summary

Code deployment is not a simple topic and you should already know that after reading this chapter. Extensive discussion of this problem could easily take a few books. Even though we limited our scope exclusively to web application, we have barely scratched the surface. We used the Twelve-Factor App methodology as the basis for showing possible solutions for various problems related to code deployment. We discussed in detail only a few of them: log treatment, managing dependencies, and separating build/run stages.

After reading this chapter, you should know how to start automating your deployment process, taking into consideration best practices, and be able to add proper instrumentation and monitoring for code that is run on your remote hosts.

In the next chapter, we will be learning why writing extensions in C and C++ for Python can sometimes be a good solution and show that it is not as hard as it seems to be as long as the proper tools are used.

9
Python Extensions in Other Languages

At the time of writing Python-based applications, you are not limited to the Python language alone. There are tools such as **Hy** (mentioned briefly in `Chapter 5`, *Elements of Metaprogramming*) that allow you to write modules, packages, or even whole applications with some other language (a dialect of Lisp) that will run in a Python virtual machine. Although it gives you the ability to express program logic with completely different syntax, it is still the same language as it compiles to the same bytecode, which means that it has the same limitations as ordinary Python code. Let me list some of the following limitations for you:

- Threading usability is greatly reduced due to the existence of **Global Interpreter Lock** (**GIL**) in CPython and dependent on the Python implementation of choice.
- Python is not a compiled language so lacks compile-time optimizations.
- Python does not provide static typing and the possible optimizations that come with it.

The solution that helps in overcoming such core limitations is Python extensions that are entirely written in a different language and expose their interface through Python extension APIs.

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:

- Differentiating between the C and C++ languages
- Writing a simple extension in C using Python/C API

- Writing a simple extension in C using Cython
- Understanding the main challenges and problems introduced by extensions
- Interfacing with compiled dynamic libraries without creating dedicated extensions and using only Python code

In the next section, we will look into the difference between the C and C++ languages.

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 App Store).

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

- `Cython`
- `cffi`

You can install these packages using following command:

```
python3 -m pip install <package-name>
```

Code files for this chapter can be found at `https://github.com/PacktPublishing/Expert-Python-Programming-Third-Edition/tree/master/chapter9`.

Differentiating between the C and C++ languages

When we talk about different languages integrated with Python, we think almost exclusively about C and C++. Even tools such as Cython or Pyrex, which define Python language supersets only for the purpose of creating Python extensions, are in fact source-to-source compilers that generate the C code from extended Python-like syntax.

In fact, you can use Python 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++. It's because 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 name such a library as a Python extension if it does not use Python/C API.

Unfortunately, writing your own extensions only in C or C++ using 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 understand better 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.

The next section explains loading extensions in C or C++.

Loading extensions in C or C++

The Python interpreter is able to load extensions from dynamic/shared libraries such as Python modules if they provide an applicable interface using Python/C API. This API must be incorporated in a source code of extension using 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 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.

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 version of Python. Also, different operating systems have incompatible ABIs, so this makes it practically impossible to create a 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 Python/C API has been defined to have stable ABIs. 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 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. So this is a trade-off and the price of the stable ABI seems to be a bit high for a very low gain.

It is important to know that 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 because there is no native way to import C/C++ code in Python directly from sources. Fortunately, `distutils` and `setuptools` provide 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. The following is an example of the `setup.py` script from the official documentation that handles the preparation of simple package distribution that has some extension written in C:

```python
from distutils.core import setup, Extension


module1 = Extension(
    'demo',
    sources=['demo.c']
)


setup(
    name='PackageName',
    version='1.0',
```

```
        description='This is a demo package',
        ext_modules=[module1]
    )
```

Once prepared this way, the following additional step is required in your distribution flow:

```
python 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 *Challenges with using extensions* section.

In the next section, we will discuss why we 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 third-party dynamic libraries
- Integrating source code written in different languages
- Creating 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 concurrencies, such as green threads or multiprocessing instead of a threading model. Third-party libraries can be integrated with the `ctypes` module. Every datatype can be implemented in Python. Still, the native Python approach may not always be optimal. Python-only integration of an external library may be clumsy and hard to maintain. 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 and so on. 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 help us to improve the performance in critical code sections.

Improving the 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. And 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 (place for 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 low 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*

Integrating existing code written in different languages 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, many 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 possible 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 how we can integrate 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 with extensions using Python/C API.

This, of course, is not the only solution and there are tools such as `ctypes` or CFFI that allow you to interact with dynamic libraries using pure Python without the need for writing extensions in C. Very often, the Python/C API may still be a better choice because it provides a better separation between the integration layer (written in C) and the rest of your application.

The next section shows us how to create custom datatypes.

Creating 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 our possible needs.

You can, of course, create many custom data structures in Python, either by basing them completely on some built-in types or by building them from scratch as completely new classes. Unfortunately, for some applications that may heavily rely on such custom data structures, 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?

In the next section, we will discuss how to write extensions.

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 or Pyrex. These projects will increase your productivity and also 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 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 *nth* number of the Fibonacci sequence. It is very unlikely that you would like to create the compiled extensions solely for this problem, but it is very simple so it will serve as a very good example of wiring any C function to 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 < 2:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Note that this is one of the most simple implementations of the `fibonnaci()` function and a lot of improvements could be applied to it even in Python. We refuse to improve our implementation (using 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.

Let's look into pure C extensions in the next section.

Pure C extensions

Before we fully dive into the code examples of Python extensions written in C, here is a huge warning. If you want to extend Python with C, you need to already know both of these languages pretty well. This is especially true for C. Lack of proficiency with it can lead to real disasters due to how easily it can be mishandled.

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. Nothing other than Python/C API details will be explained here. This book is about Python and not any other language. If you don't know C at all, you should definitely not try to write your own Python extensions in C until you gain enough experience and skills. Leave it to others and stick with Cython or Pyrex because they are a lot safer from the beginner's perspective. It's because Python/C API, despite being crafted with great care, is definitely not a good introduction to C.

As proposed 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 rough as follows:

```c
long long fibonacci(unsigned int n) {
    if (n < 2) {
        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
#include <Python.h>


long long fibonacci(unsigned int n) {
    if (n < 2) {
        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);
    }
```

The preceding example might be a bit overwhelming at first glance because we had to add four times more code just to make the `fibinacci()` 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. Same source files you can find in the directory `chapter9/fibonacci_c` of this book's code package. The following transcript presents the result of the installation in development mode and a simple interactive session where our compiled `fibonacci()` function is inspected and executed:

```
$ ls -1ap
fibonacci.c
setup.py

$ python3 -m pip install -e .
Obtaining file:///Users/swistakm/dev/Expert-Python-Programming-
Third_edition/chapter9
Installing collected packages: fibonacci
  Running setup.py develop for fibonacci
Successfully installed Fibonacci

$ ls -1ap
build/
fibonacci.c
fibonacci.cpython-35m-darwin.so
fibonacci.egg-info/
setup.py
$ python3
Python 3.7.2 (default, Feb 12 2019, 00:16:38)
[Clang 10.0.0 (clang-1000.11.45.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import fibonacci
>>> help(fibonacci.fibonacci)

Help on built-in function fibonacci in fibonacci:
```

```
    fibonacci.fibonacci = fibonacci(...)
        fibonacci(n): Return nth Fibonacci sequence number computed recursively

>>> [fibonacci.fibonacci(n) for n in range(10)]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Python/C API is explained in the next section.

A closer look at Python/C API

Since we know how to properly package, compile, and install custom C extensions and we are sure that it works as expected, now it is the right time to discuss our code in detail.

The `extensions` module starts with the following single C preprocessor directive that 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 < 2) {
        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 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 only `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 leave it as it is for the sake of brevity. We will try to fix it later when discussing details of dealing with errors in the *Exception handling* section.

The `"l"` string in the `PyArg_ParseTuple(args, "l", &n)` call means that we expect `args` to contain only a single `long` value. In case of failure, it will return `NULL` and store information about the exception in the per thread interpreter state. The details of exception handling will be described a bit later in the *Exception handling* section.

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 that for some simple examples, such as this one, can take more place 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. This structure contains exactly the four following 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) as follows:

```
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 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 `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 pick 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.

In the next section, we will see how we can call and bind conventions.

Calling and binding conventions

As explained in the *A closer look at 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 (`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 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 keyword arguments that are printed on standard output:

```
#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 Python/C API is very elastic and is extensively described in the official documentation at `https://docs.python.org/3.7/c-api/arg.html`. The format argument in `PyArg_ParseTuple()` and `PyArg_ParseTupleAndKeywords()` allows fine-grained control over 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 variable numbers of arguments

The **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 `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 own classes and types using 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 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 an `int` type, it returns –1.

The only exceptions from the preceding rules in 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;
}
```

Lines that somehow take part in our error handling are highlighted. 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 case of an exception and return `0`. This is part of the `if` statement and in that case, we don't do anything more and 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 case of failure, it gives `NULL`. We can simply store it as our result variable and pass further 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. 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 error cause. The full signature of this function is as follows:

```
void PyErr_SetString(PyObject* type, const char* message)
```

We have already said that the implementation of our `fibonacci_py()` function has a serious bug. 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 (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()`.

In the next section, we will discuss releasing GIL.

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. While multiprocessing is the suggested approach to circumvent this problem (see `Chapter 15`, *Concurrency*), it may not be a good 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 Python/C API, it is possible (even advisable) to release GIL in some application sections while still doing some data processing. Thanks to this, you can still benefit from having multiple CPU cores and multithreaded application design. 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 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:

```
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;
}
```

The preceding technique is fair but requires caution.

Reference counting is discussed in the next section.

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 and they are always shared. The actual creation of objects is managed by Python's memory manager. It is the component of 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 called. Python/C API provides two macros for increasing and decreasing reference counts—`Py_INCREF()` and `Py_DECREF()`. 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 for some circumstances to additionally protect such references with separate `Py_INCREF()` before using it as an argument to other functions and then calling `Py_DECREF()` when it is no longer needed.

- **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 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 0, 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 become deallocated. It is possible that it was some user-defined class with custom implementation of the `__del__()` method. A serious issue will occur if in the result of such `__del__()` execution the `item[0]` will be 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 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);
}
```

In the next section, we will learn how to write extensions using Cython instead of bare Python/C API.

Writing extensions with Cython

Cython is both an optimizing static compiler and the name of a programming language that is a superset of Python. As a compiler, it can perform *source-to-source* compilation of native Python code and its Cython dialect to Python C extensions using Python/C API. It allows you to combine the power of Python and C without the need to manually deal with Python/C API.

Let's discuss Cython as a source-to-source compiler in the next section.

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 the 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 at a very low development cost.

Cython provides a simple `cythonize` utility function that allows you to easily integrate the compilation process with `distutils` or `setuptools`. 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` module, the minimal `setup.py` script could be as follows:

```
from setuptools import setup
from Cython.Build import cythonize

setup(
    name='fibonacci',
    ext_modules=cythonize(['fibonacci.py'])
)
```

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. 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 he wants to use Cython with a specific environment variable, for example:

```python
import os

from distutils.core import setup
from distutils.extension import Extension

try:
    # cython source to source compilation available
    # only when Cython is available
    import Cython
    # and specific environment variable says
    # explicitly that Cython should be used
    # to generate C sources
    USE_CYTHON = bool(os.environ.get("USE_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 intalled
        # with '[with-cython]' extra feature
        'with-cython': ['cython==0.23.4']
    }
)
```

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 variables guarantee 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.

We will take a look at Cython as a 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 and it can be further updated 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. 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 saying that our reference `fibonacci()` function is also code for valid extensions in this superset of Python, we will try to improve it a bit. This won't be any real optimization to our function design but 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`. We still want to implement our Fibonacci sequence recursively.

The content of `fibonacci.pyx` might look like this:

```
"""Cython module that provides fibonacci sequence function."""


def fibonacci(unsigned int n):
    """Return nth Fibonacci sequence number computed recursively."""
    if n < 2:
        return n
    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 as follows:

```
>>> from fibonacci import fibonacci
>>> fibonacci(5)
5
>>> fibonacci(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "fibonacci.pyx", line 21, in fibonacci.fibonacci (fibonacci.c:704)
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 21, in fibonacci.fibonacci (fibonacci.c:704)
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 `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 < 2:
        return n
    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)
```

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 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:

```
def fibonacci(unsigned int n):
    """ Return nth Fibonacci sequence number computed recursively
    """
    with nogil:
        result = fibonacci_cc(n)

    return fibonacci_cc(n)
```

You can also mark the whole C-style function as safe to call without GIL as follows:

```
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.

The next section discusses challenges with using extensions.

Challenges with 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 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 as other languages are. We could go on with that list forever.

One of the solutions to fix that problem is to write extensions, so we can 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.

Additional complexities are explained in the next section.

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 debugging of C code requires completely different workflow and tools, but also because you will need to switch context between two different languages very often.

We are all humans and 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 *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.

Debugging is explained in the next section.

Debugging

When it comes to failures, the extensions may break very badly. 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. On the other hand, all memory management 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. 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 next section discusses interfacing 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), 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, so this is an interesting alternative to writing own extensions in C.

This does not mean you don't need to know anything about C. Both solutions 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 take a look at `ctypes` in the next section.

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. It is a **Foreign Function Interface** (**FFI**) library and provides APIs for creating C-compatible datatypes.

In the next section, we will take a look at loading libraries.

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 last two are available only on Windows, so we won't discuss them here in detail. The differences between `CDLL` and `PyDLL` 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 `int`. GIL is released during the call.

- `ctypes.PyDLL`: This class works like `CDLL`, but GIL is not released during the call. After execution, the Python error flag is checked and an exception is raised if it is set. It is only useful when the loaded library is directly calling functions from Python/C API or uses callback functions that may be a Python code.

To load the library, you can either instantiate one of the preceding classes with proper arguments or call the `LoadLibrary()` function from 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 (refer to `https://docs.python.org/3.5/library/ctypes.html`).

Both library loading conventions (the `LoadLibrary()` function 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 X, 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 '/usr/lib/libc.dylib', handle 7fff69b97c98 at 0x101b73ac8>
>>> ctypes.cdll.LoadLibrary(find_library('bz2'))
<CDLL '/usr/lib/libbz2.dylib', handle 10042d170 at 0x101b6ee80>
>>> ctypes.cdll.LoadLibrary(find_library('AGL'))
<CDLL '/System/Library/Frameworks/AGL.framework/AGL', handle 101811610 at
0x101b73a58>
```

So, if you are writing a `ctypes` package that is supposed to work both under macOS and Linux, always use `ctypes.util.find_library()`.

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 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 object
`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 unsigned 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 * (NUL terminated)	bytes object or None
c_wchar_p	wchar_t * (NUL 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
```

As 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 CPython interpreter that 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:

```
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:

```
from random import shuffle

import ctypes
from ctypes.util 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),
)


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 lenght
        # same as the lenght 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:

```
$ python 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 next section talks about CFFI.

CFFI

CFFI is an FFI for Python that is an interesting alternative to `ctypes`. It is not a part of the standard library but it is easily available as a 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 C compiler. So 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 more 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 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 advanced topics in the book. We discussed the reasons and tools for building Python extensions. We started by writing pure C extensions that depend only on 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 or Pyrex (not featured here) 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 next chapter, we will take a short rest from advanced programming techniques and delve into topics that are no less important—code management and version control systems.

Section 3: Quality over Quantity **3**

This part deals with various development processes aimed to increase software quality and streamline the overall development processes. The reader will learn how to properly manage source code in version-control systems, how to document their code, and how to ensure that it will be always thoroughly tested.

The following chapters are included in this section:

- Chapter 10, *Managing Code*
- Chapter 11, *Documenting Your Project*
- Chapter 12, *Test-Driven Development*

10
Managing Code

Working on a software project that involves more than one person is tough. Everything seems to slow down and get harder as you add more people to the team. This happens for many reasons. In this chapter, we will explore a few of these reasons and also try to provide some ways of working that aim to improve the collaborative development of code.

First of all, every code base evolves over time, and it is important to track all the changes that are made, even more so when many developers work on it. That is the role of a **version control system**.

It's very common that multiple people expand the same code base simultaneously and in parallel. It's definitely easier if all these people have different roles and work on different aspects. But that's rarely true. Therefore, a lack of global visibility generates a lot of confusion about what is going on, and what is being done by others. This is unavoidable, and some tools have to be used to provide continuous visibility and mitigate the problem. This is done by setting up a series of tools for continuous development processes, such as **continuous integration** or **continuous delivery**.

In this chapter, we will cover the following topics:

- Working with a version control system
- Setting up continuous development processes

Technical requirements

You can download the latest version of Git from `https://git-scm.com` for this chapter.

Working with a version control system

Version control systems (**VCSes**) provide a way to share, synchronize, and back up any kind of file, but often concentrate on text files containing source code. They are categorized into the following two families:

- Centralized systems
- Distributed systems

Let's take a look at the preceding families in the following sections.

Centralized systems

A centralized version control system is based on a single server that holds the files and lets people check in and check out the changes that are made to those files. The principle is quite simple—everyone can get a copy of the files on his/her system and work on them. From there, every user can *commit* his/her changes to the server. They will be applied and the *revision* number will be raised. The other users will then be able to get those changes by synchronizing their *repository* copy through an *update*.

As the following diagram shows, a repository evolves through all the commits, and the system archives all revisions into a database in order to be able to undo any change, or provide information on what has been done and by whom:
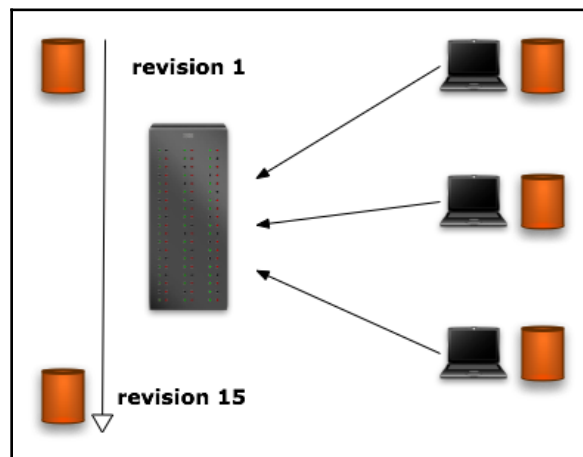
Figure 1

Every user in this centralized configuration is responsible for synchronizing his/her local repository with the main one, in order to get the other users' changes. This means that some conflicts can occur when a locally modified file has been changed and checked in by someone else. A conflict resolution mechanism is carried out, in this case on the user system, as shown in the following diagram:
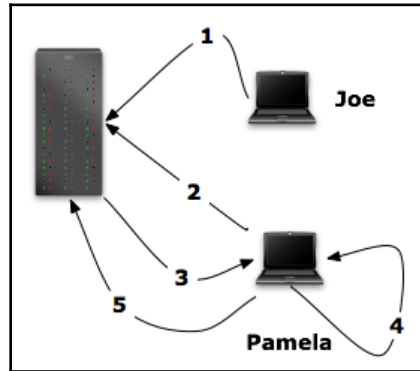
Figure 2

The following steps will help you to understand this process better:

1. Joe checks in a change.
2. Pamela attempts to check in a change on the same file.
3. The server complains that her copy of the file is out of date.
4. Pamela updates her local copy. The version control software may or may not be able to merge the two versions seamlessly (that is, without a conflict).
5. Pamela commits a new version that contains the latest changes made by Joe and her own.

This process is perfectly fine on small-sized projects that involve a few developers and a small number of files, but it becomes problematic for bigger projects. For instance, a complex change involves a lot of files, which is time-consuming, and keeping everything local before the whole work is done is unfeasible. The following are some problems of such an approach:

- The user may keep his/her changes in private for a long time without a proper backup
- It is hard to share work with others until it is checked in, and sharing it before it is fully done would leave the repository in an unstable state, and so the other users would not want to share

A centralized VCS can resolve this problem by providing *branches* and *merges*. It is possible to fork from the main stream of revisions to work on a separated line, and then to get back to the main stream.

In the following diagram, Joe starts a new branch from revision 2 to work on a new feature. The revisions are incremented in the main stream and in his branch, every time a change is checked in. At revision 7, Joe has finished his work and committed his changes into the trunk (the main branch). This process often requires some conflict resolution.

But in spite of their advantages, a centralized VCS has the following pitfalls:

- Branching and merging is quite hard to deal with. It can become a nightmare.
- Since the system is centralized, it is impossible to commit changes offline. This can lead to a huge and single commit to the server when the user gets back online.
- Lastly, it doesn't work very well for projects such as Linux, where many companies permanently maintain their own branch of the software and there is no central repository that everyone has an account on.

For the latter, some tools are making it possible to work offline, such as SVK, but a more fundamental problem is how the centralized VCS works:
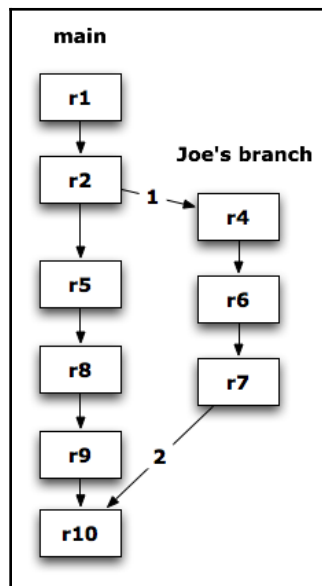
Figure 3

Despite these pitfalls, centralized VCSes are still quite popular among many companies, mainly due to the inertia of corporate environments. The main examples of centralized VCSes used by many organizations are **Subversion** (**SVN**) and **Concurrent Version System** (**CVS**). The obvious issues with a centralized architecture for version control systems is the reason why most of the open source communities have switched already to the more reliable architecture of **Distributed VCS** (**DVCS**).

Distributed systems

Distributed VCS is the answer to the centralized VCS deficiencies. It does not rely on a main server that people work with, but on peer-to-peer principles. Everyone can hold and manage his/her own independent repository for a project and synchronize it with other repositories, as shown in the following diagram:
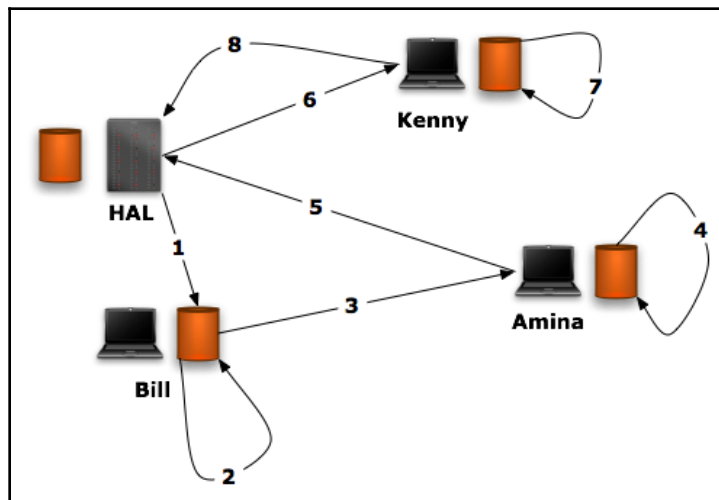
Figure 4

In preceding diagram, we can see an example of such a system in use:

1. Bill *pulls* the files from HAL's repository.
2. Bill makes some changes to the files.
3. Amina *pulls* the files from Bill's repository.
4. Amina changes the files too.
5. Amina *pushes* the changes to HAL.

6. Kenny *pulls* the files from HAL.
7. Kenny makes changes.
8. Kenny regularly *pushes* his changes to HAL.

The key concept is that people *push* and *pull* the files to or from other repositories, and this behavior changes according to the way people work and the way the project is managed. Since there is no main repository anymore, the maintainer of the project needs to define a strategy for people to *push* and *pull* the changes.

Furthermore, people have to be a bit smarter when they work with several repositories. In most distributed version control systems, revision numbers are local to each repository; there are no global revision numbers anyone can refer to. Therefore, *tags* have to be used to make things clearer. Tags are textual labels that can be attached to a revision. Lastly, users are responsible for backing up their own repositories, which is not the case in a centralized infrastructure where the administrator usually sets proper backup strategies.

In the next section, distributed strategies are explained.

Distributed strategies

A central server is, of course, still desirable with a DVCS if you're working with other people. But, the purpose of that server is completely different than in centralized VCSes. It is simply a hub that allows all developers to share their changes in a single place, rather than pull and push between each other's repositories. Such a single central repository (often called *upstream* also) serves as a backup for all the changes tracked in the individual repositories of all the following team members.

Different approaches can be applied to sharing code with the central repository in a DVCS. The simplest one is to set up a server that acts like a regular centralized server, where every member of the project can push his/her changes into a common stream. But this approach is a bit simplistic. It does not take full advantage of the distributed system, since people will use push and pull commands in the same way as they would with a centralized system.

Another approach consists of providing several repositories on a server with the following different levels of access:

- **Unstable repository**: This is where everyone can push changes.
- **Stable repository**: This is read-only for all members, except the release managers. They are allowed to pull changes from the unstable repository and decide what should be merged.
- **Release repositories**: These correspond to the releases and are read-only.

This allows people to contribute and managers to review the changes before they make it to the stable repository. But depending on the tools used, this may be too much of an overhead. In many distributed version control systems, this can also be handled with a proper branching strategy.

Let' take a look at centralized and distributed version control systems.

Centralized or distributed?

Just forget about the centralized version control systems. Let's be honest, centralized version control systems are relics of the past. In the times when most of us have the opportunities to work remotely full-time, it is unreasonable to be constrained by all the deficiencies of a centralized VCS. For instance, with CVS or SVN, you can't track the changes when offline. And that's silly.

What should you do when the internet connection at your workplace is temporarily broken or the central repository goes down? Should you forget about all your workflow and just allow changes to pile up until the situation changes, and then just commit them as one huge blob of unstructured updates? No!

Also, most of the centralized version control systems do not handle branching schemes efficiently. And branching is a very useful technique that allows you to limit the number of merge conflicts in the projects where many people work on multiple features. Branching in SVN is so ridiculous that most of the developers try to avoid it at all costs. Instead, most of the centralized VCSes provide some file locking primitives that should be considered the anti-pattern for any version system.

The sad truth about every version control tool (and software in general) is that, if it contains a dangerous option, someone in your team will start using it on a daily basis eventually. And locking is one such feature that, in return for less merge conflicts, will drastically reduce the productivity of your whole team. By choosing a version control system that does not allow for such bad workflows, you are creating an environment that makes it more likely that your developers will use it effectively.

In the next section, the Git distributed version control system is explained.

Use Git if you can

Git is currently the most popular distributed version control system. It was created by Linus Torvalds for the purpose of maintaining versions of the Linux kernel when its core developers needed to resign from proprietary BitKeeper software that they used previously.

If you have not used any of the version control systems, then you should start with Git from the beginning. If you already use some other tools for version control, learn Git anyway. You should definitely do that, even if your organization is unwilling to switch to Git in the near future. Otherwise, you risk becoming a living fossil.

I'm not saying that Git is the ultimate and best DVCS. It surely has some disadvantages. Most of all, it is not an easy to use tool and is very challenging for newcomers. Git's steep learning curve is already a source of many jokes online. There may be some version control systems that may perform better for a lot of projects, and the full list of open source Git contenders would be quite long. Anyway, Git is currently the most popular DVCS, so the *network effect* really works in its favor.

Briefly speaking, the network effect means that the overall benefit of using popular tools is greater than others, even if definitely better alternatives exist, precisely due to its high popularity (this is how VHS killed Betamax). It is very probable that people in your organization, as well as new hires, are somewhat proficient with Git, so the cost of integrating exactly this DVCS will be lower than trying something less popular.

Anyway, it is still always good to know something more, and familiarizing yourself with other DVCSes won't hurt you. The most popular open source rivals of Git are Mercurial, Bazaar, and Fossil. The first one is especially neat because it is written in Python and was the official version control system for CPython sources. There are some signs that it may change in the near future, so CPython developers may already use Git at the time you read this book. But it really does not matter. Both systems are great. If there was no Git or it was less popular, I would definitely recommend Mercurial. There is evident beauty in its design. It's definitely not as powerful as Git, but is a lot easier to master for beginners.

Let's take a look at GitFlow and GitHub Flow in the next section.

GitFlow and GitHub Flow

The very popular and standardized methodology for working with Git is simply called **GitFlow**. Here is a brief description of the main rules of that flow:

- There is a main working branch, usually called `develop`, where all the development for the latest version of the application occurs.

- New project features are implemented in separate branches called *feature branches* that always start from the develop branch. When work on a feature is finished and the code is properly tested, this branch is merged back to `develop`.

- When the code in `develop` is stabilized (without known bugs) and there is a need for a new application release, a new *release branch* is created. This release branch usually requires additional tests (extensive QA tests, integration tests, and so on), so new bugs will definitely be found. If additional changes (such as bug fixes) are included in a release branch, they need to eventually be merged back to the `develop` branch.

- When code on a *release branch* is ready to be deployed/released, it is merged to the `master` branch, and the latest commit on the `master` is labeled with an appropriate version tag. No other branches but feature branches can be merged to the `master`. The only exceptions are hot fixes that need to be immediately deployed or released.

- Hot fixes that require urgent release are always implemented on separate branches that start from the `master`. When the fix is done, it is merged to both the `develop` and `master` branches. Merging of the hot fix branch is done as if it were an ordinary release branch, so it must be properly tagged and the application version identifier should be modified accordingly.

The visual example of GitFlow in action is presented in the following. For those that have never worked in such a way and also have never used a distributed version control system, this may be a bit overwhelming. Anyway, it is really worth trying in your organization if you don't have any formalized workflow. It has multiple benefits and also solves real problems. It is especially useful for teams of multiple programmers that are working on many separate features, and when continuous support for multiple releases needs to be provided.

This methodology is also handy if you want to implement continuous delivery using continuous deployment processes because it makes it clear which version of code represents a deliverable release of your application or service. It is also a great tool for open source projects because it provides great transparency to both the users and the active contributors:
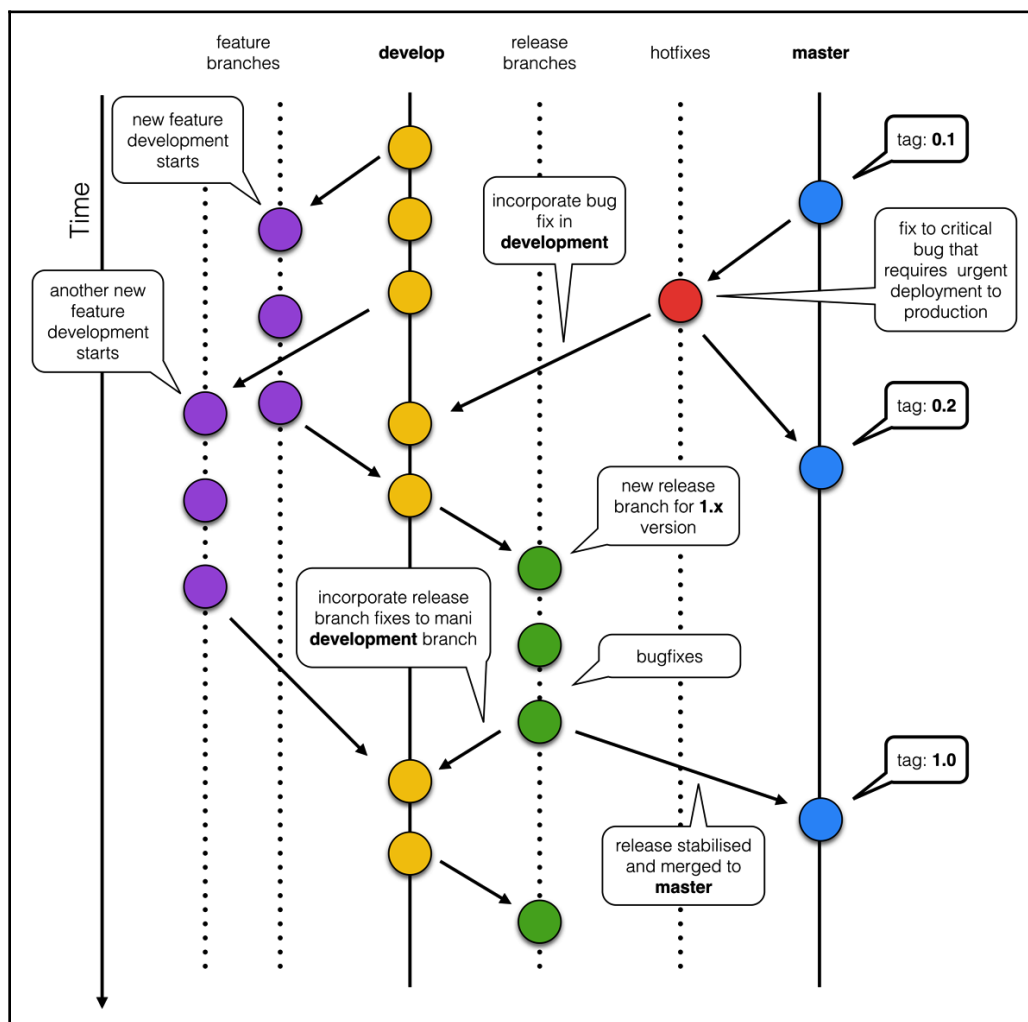
Figure 5

So, if you think that this short summary of GitFlow makes a bit of sense and it did not scare you yet, then you should dig deeper into online resources on that topic. It is really hard to say who is the original author of the preceding workflow, but most online sources point to Vincent Driessen. Thus, the best starting material to learn about GitFlow is his online article titled *A successful Git branching model* (refer to `http://nvie.com/posts/a-successful-git-branching-model/`).

Like every other popular methodology, GitFlow gained a lot of criticism over the internet from programmers that do not like it. The most commented thing about Vincent Driessen's article is the rule (strictly technical) saying that every merge should create a new artificial commit representing that merge. Git has an option to do *fast forward* merges, and Vincent discourages that option. This is, of course, an unsolvable problem because the best way to perform merges is a completely subjective matter. Anyway, the real issue of GitFlow is that it is noticeably complicated. The full set of rules is really long, so it is easy to make some mistakes. It is very probable that you would like to choose something simpler.

One such simpler flow is used at GitHub and described by Scott Chacon on his blog (refer to `http://scottchacon.com/2011/08/31/github-flow.html`). It is referred to as **GitHub Flow**, and is very similar to GitFlow in the following two main aspects:

- Anything in the master branch is deployable
- The new features are implemented on separate branches

The main difference from GitFlow is simplicity. There is only one main development branch, `master`, and it is always stable (in contrast to the `develop` branch in GitFlow). There are also no release branches and such big emphasis on tagging the code. There is no such need in GitHub Flow because, as they say, when something is merged into the master, it is usually deployed to production immediately. A diagram presenting an example of GitHub Flow in action is shown in here.

GitHub Flow seems like a good and lightweight workflow for teams that want to set up a continuous deployment process for their project. Such a workflow is, of course, not viable for any project that has a strong notion of release (with strict version numbers), at least without any modifications. It is important to know that the main assumption of the *always deployable* `master` branch cannot be ensured without a proper automated testing and building procedure. This is what continuous integration systems take care of, and we will discuss that a bit later.

The following is a diagram presenting an example of GitHub Flow in action:
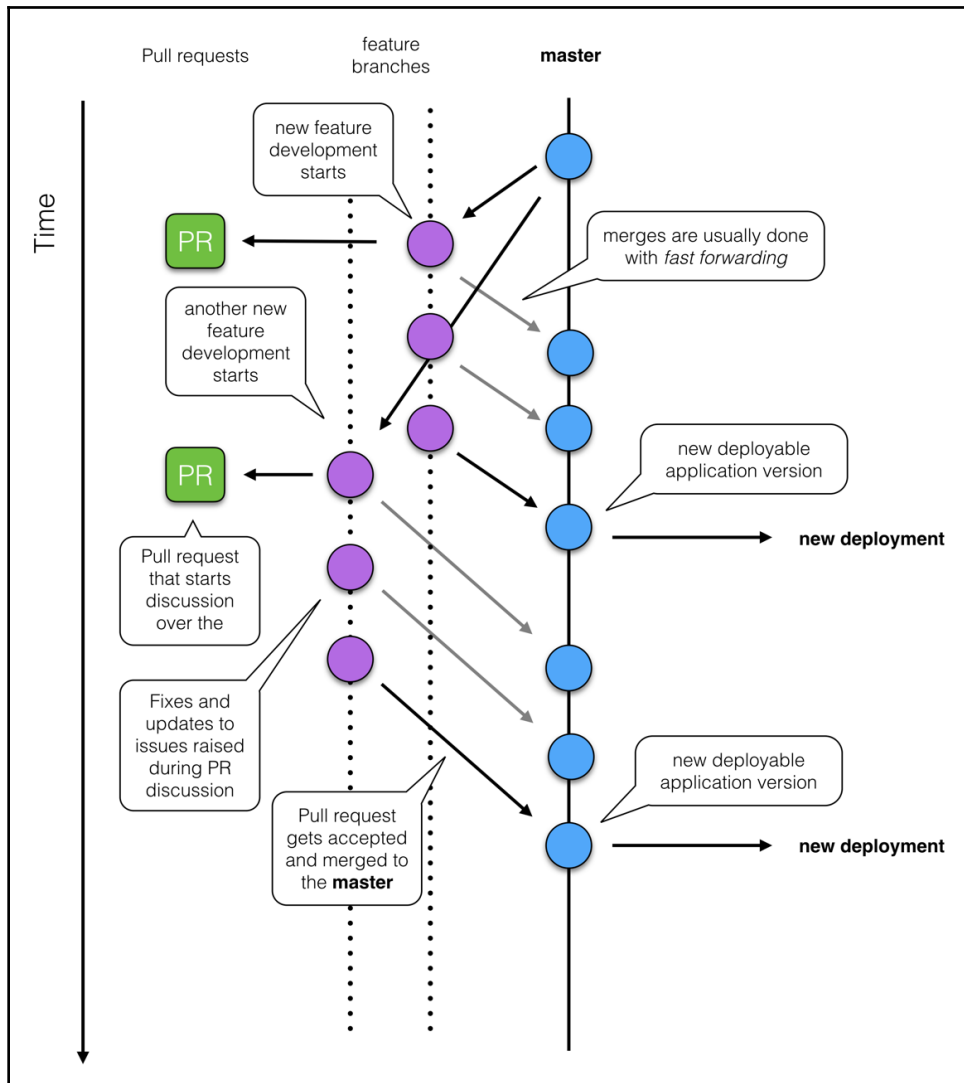
Figure 6: Visual presentation of GitHub flow in action

Note that both GitFlow and GitHub Flow are only branching strategies, so despite having *Git* in their names, these strategies are not limited to that single DVCS solution. It's true that the official article describing GitFlow mentions even specific `git` command parameter that should be used when performing a merge, but the general idea can be easily applied to almost any other distributed version control system.

In fact, due to the way it is suggested to handle merges, Mercurial seems like a better tool to use for this specific branching strategy! The same applies to GitHub Flow. This is the only branching strategy sprinkled with a bit of specific development culture, so it can be used in any version control system that allows you to easily create and merge branches of code.

As a last comment, remember that no methodology is carved in stone and no one forces you to use it. Methodologies are created to solve some existing problems and keep you from common mistakes or pitfalls. You can take all of their rules or modify some of them to your own needs. They are great tools for beginners that may easily get into common pitfalls.

If you are not familiar with any version control systems, you should then start with a lightweight methodology like GitHub Flow without any custom modifications. You should start thinking about more complex workflows only when you get enough experience with Git or any other tool of your choice. Anyway, as you gain more and more proficiency, you will eventually realize that there is no perfect workflow that suits every project. What works well in one organization doesn't necessarily work well in others.

Setting up continuous development processes

There are some processes that can greatly streamline your development and reduce the time in getting the application ready to be released or deployed to the production environment. They often have *continuous* in their name. We will discuss the most important and popular ones in this section. It is important to highlight that they are strictly technical processes, so it is almost not related to project management methodologies, although they can highly dovetail with the latter.

The following are the most important processes that we will mention:

- Continuous integration
- Continuous delivery
- Continuous deployment

The order of listing is important because each one of them is an extension of the previous one. Continuous deployment could be simply considered as a variation of the same process. We will discuss them separately anyway, because what is only a minor difference for one organization may be critical for others.

The fact that these are technical processes means that their implementation strictly depends on the usage of proper tools. The general idea behind each of them is rather simple, so you could build your own continuous integration/delivery/deployment tools, but the best approach is to choose something that is already built and tested. This way, you can focus more on building your own product instead of developing the tool chain for continuous development.

Let's take a look at continuous integration in the next section.

Continuous integration

Continuous integration, often abbreviated as **CI**, is a process that takes benefit from automated testing and version control systems to provide a fully automatic integration environment. It can be used with centralized version control systems, but in practice, it spreads its wings only when a good DVCS tool is being used to manage the code.

Setting up a repository is the first step toward continuous integration, which is a set of software practices that have emerged from **eXtremeProgramming** (**XP**).

The first and most important requirement to implement continuous integration is to have a fully automated workflow that can test the whole application in the given revision in order to decide if it is technically correct. And technically correct means that it is free of known bugs and that all the features work as expected.

The general idea behind CI is that tests should be run always before merging to the mainstream development branch. This could be handled only through formal arrangements in the development team, but practice shows that this is not a reliable approach. The problem is that, as programmers, we tend to be overconfident and unable to look critically at our code. If continuous integration is built only on team arrangements, it will inevitably fail because some of the developers will eventually skip their testing phase and commit possibly faulty code to the mainstream development branch that should always remain stable. And, in reality, even simple changes can introduce critical issues.

The obvious solution is to utilize a dedicated build server that automatically runs all the required application tests whenever the code base changes. There are many tools that streamline this process, and they can be easily integrated with version control hosting services such as GitHub or Bitbucket, and self-hosted services such as GitLab. The benefit of using such tools is that the developer may locally run only the selected subset of tests (that, according to them, are related to their current work) and leave the potentially time-consuming suite of integration tests for the build server. This really speeds up the development, but still reduces the risk that new features will break the existing stable code found in the mainstream code branch.

Another plus of using a dedicated build server is that tests can be run in the environment that is closer to the production. Despite the fact that developers should also use environments that match the production as much as possible, and there are great tools for that (such as Vagrantand Docker), it is hard to enforce this in any organization. You can easily do that on one dedicated build server or even on a cluster of build servers. Many CI tools make that even less problematic by utilizing various virtualization and/or containerization tools that help to ensure that tests are run always in the same and completely fresh testing environment.

Having a build server is also a must if you create desktop or mobile applications that must be delivered to users in binary form. The obvious thing to do is to always perform such a building procedure in the same environment. Almost every CI system takes into account the fact that applications often need to be downloaded in binary form after testing/building is done. Such building results are commonly referred to as **build artefacts**.

Because CI tools originated in times where most of the applications were written in compiled languages, they mostly use the term *building* to describe their main activity. For languages such as C or C++, this is obvious because applications cannot be run and tested if they are not built (compiled). For Python, this makes a bit less sense because most of the programs are distributed in a source form and can be run without any additional building step. So, in the scope of our language, the *building* and *testing* terms are often used interchangeably when talking about continuous integration.

Testing every commit

The best approach to continuous integration is to perform the whole test suite on every change being pushed to the central repository. Even if one programmer pushes a series of multiple commits in a single branch, it often makes sense to test each change separately. If you decide to test only the latest change set in a single repository push, then it will be harder to find sources of possible regression problems that were introduced somewhere in the middle.

Of course, many DVCS, such as Git or Mercurial, allow you to limit time spent on searching regression sources by providing commands to *bisect* the history of changes, but in practice, it is much more convenient to do that automatically as part of your continuous integration process.

There is an issue of projects that have very long running test suites that may require tens of minutes or even hours to complete. One server may not be enough to perform all the builds on every commit made in the given time frame. This will make waiting for results even longer. In fact, long running tests are a problem on their own that will be described later in the *Problem 2 – Too long building time* section.

For now, you should know that you should always strive to test every commit pushed to the repository. If you have no power to do that on a single server, then set up the whole building cluster. If you are using a paid service, then pay for a higher pricing plan with more parallel builds. Hardware is cheap. Your developers' time is not. Eventually, you will save more money by having faster parallel builds and a more expensive CI setup than you would save on skipping tests for selected changes.

Merge testing through CI

Reality is complicated. If the code on a feature branch passes all the tests, it does not mean that the build will not fail when it is merged to a stable mainstream branch. Both of the popular branching strategies mentioned in the *GitFlow and GitHub Flow* section assume that code merged to the `master` branch is always tested and deployable. But how can you be sure that this assumption is met if you did not perform the merge yet? This is a lesser problem for GitFlow (if implemented well and used precisely), due to its emphasis on release branches. But it is a real problem for simple GitHub Flow, where merging to `master` is often related with conflicts and is very likely to introduce regressions in tests. Even for GitFlow, this is a serious concern. This is a complex branching model, so for sure people will make mistakes when using it. So, you can never be sure that the code on master will pass the tests after a merge unless you take some special precautions.

One of the solutions to this problem is to delegate the duty of merging feature branches into a stable mainstream branch to your CI system. In many CI tools, you can easily set up an on-demand building job that will locally merge a specific feature branch to the stable branch and push it to the central repository if it passed all the tests. If the build fails, then such a merge will be reverted, leaving the stable branch untouched.

Of course, this approach gets more complex in fast paced projects where many feature branches are developed simultaneously, because there is high risk of conflicts that can't be resolved automatically by any CI system. There are, of course, solutions to that problem, like *rebasing* in Git.

Such an approach to merging anything into the stable branch in a version control system is practically a must if you about going further and implementing continuous delivery processes. It is also required if you have a strict rule in your workflow stating that everything in a stable branch is releasable.

Matrix testing

Matrix testing is a very useful tool if your code needs to be tested in different environments. Depending on your project's needs, the direct support of such a feature in your CI solution may be more or less required.

The easiest way to explain the idea of matrix testing is to take the example of some open source Python packages. Django, for instance, is the project that has a strictly specified set of supported Python language versions. The 1.9.3 version lists Python 2.7, Python 3.4, and Python 3.5 versions as required in order to run Django code. This means that every time Django core developers make a change to the project, the full test suite must be executed on these three Python versions in order to back this claim. If even a single test fails on one environment, the whole build must be marked as failed because the backwards compatibility constraint was possibly broken. For such a simple case, you do not need any support from CI. There is a great Tox tool (refer to `https://tox.readthedocs.org/`) that, among other features, allows you to easily run test suites in different Python versions in isolated virtual environments. This utility can also be easily used in local development.

But this was only the simplest example. It is not uncommon that the application must be tested in multiple environments where completely different parameters must be tested. The following are a few of these:

- Different operating systems
- Different databases
- Different versions of backing services
- Different types of filesystems

The full set of combinations forms a multi-dimensional environment parameter matrix, and this is why such a setup is called matrix testing. When you need such a deep testing workflow, it is very possible that you require some integrated support for matrix testing in your CI solution. With a large number of possible combinations, you will also require a highly parallelizable building process, because every run over the matrix will require a large amount of work from your building server. In some cases, you will be forced to decide on some trade-off if your test matrix has too many dimensions.

In the next section, continuous delivery is explained.

Continuous delivery

Continuous delivery is a simple extension of the continuous integration idea. This approach to software engineering aims to ensure that the application can be released reliably at any time. The goal of continuous delivery is to release software in short circles. It generally reduces both costs and the risk of releasing software by allowing the incremental delivery of changes to the application in production.

The following are the main prerequisites for building successful continuous delivery processes:

- A reliable continuous integration process
- An automated process of deployment to the production environment (if the project has a notion of the production environment)
- A well-defined version control system workflow or branching strategy that allows you to easily define what version of software represents releasable code

In many projects, the automated tests are not enough to reliably tell if the given version of the software is really ready to be released. In such cases, additional manual user acceptance tests are usually performed by skilled QA staff. Depending on your project management methodology, this may also require some approval from the client. This does not mean that you can't use GitFlow, GitHub Flow, or a similar branching strategy, if some of your acceptance tests must be performed manually by humans. This only changes the semantics of your stable and release branches from *ready to be deployed* to *ready for user acceptance tests and approval*.

Also, the previous paragraph does not change the fact that code deployment should always be automated. We already discussed some of the tools and benefits of automation in `Chapter 8`, *Deploying the Code*. As stated there, it will always reduce the cost and risk of a new release. Also, most of the available CI tools allow you to set up special build targets that instead of testing will perform automated deployment for you. In most continuous delivery processes, this is usually triggered manually (on demand) by authorized staff members when they are sure that there is the required approval and all acceptance tests ended with success.

Let's take a look at continuous deployment in the next section.

Continuous deployment

Continuous deployment is a process that takes continuous delivery to the next level. It is a perfect approach for projects where all acceptance tests are automated and there is no need for manual approval from anyone. In short, once code is merged to the stable branch (usually `master`), it is automatically deployed to the production environment.

This approach seems to be very nice and robust, but is not often used because it is very hard to find a project that does not need manual QA testing and someone's approval before a new version is released. Anyway, it is definitely doable and some companies claim to be working that way.

In order to implement continuous deployment, you need the same basic prerequisites as the continuous delivery process. Also, a more careful approach to merging into a stable branch is often required. What gets merged into the `master` in continuous integration usually goes instantly to the production environment. Because of that, it is reasonable to hand off the merging task to your CI system, as explained in the *Merge testing through CI* section.

In the next section, popular tools for continuous integration are described.

Popular tools for continuous integration

There is a tremendous variety of choices for CI tools nowadays. They greatly vary on ease of use and also available features, and almost each one of them has some unique features that others will lack. Therefore, it is hard to give a good general recommendation because each project has completely different needs and also a different development workflow. There are, of course, some great free and open source projects, but paid hosted services are also worth researching. It's because open source software, such as Jenkins or Buildbot, are freely available to install without any fee, but it is false thinking that they are free to run. Both hardware and maintenance are added costs of having your own CI system. In some circumstances, it may be less expensive to pay for such a service instead of paying for additional infrastructure and spend time on resolving any issues in open source CI software. Still, you need to make sure if sending your code to any third-party service is in line with security policies at your company.

Here, we will review some of the most popular, free open source tools, as well as paid hosted services. I really don't want to advertise any vendor, so we will discuss only those that are available without any fees for open source projects to justify this rather subjective selection. No best recommendation will be given, but we will point out both the good and bad sides of any solution. If you are still in doubt, the next section, which describes common continuous integration pitfalls, should help you in making good decisions.

Let's take a look at Jenkins in the next section.

Jenkins

Jenkins (`https://jenkins-ci.org`) seems to be the most popular tool for continuous integration. It is also one of the oldest open source projects in this field in pair with Hudson (the development of these two projects split and Jenkins is a fork of Hudson):
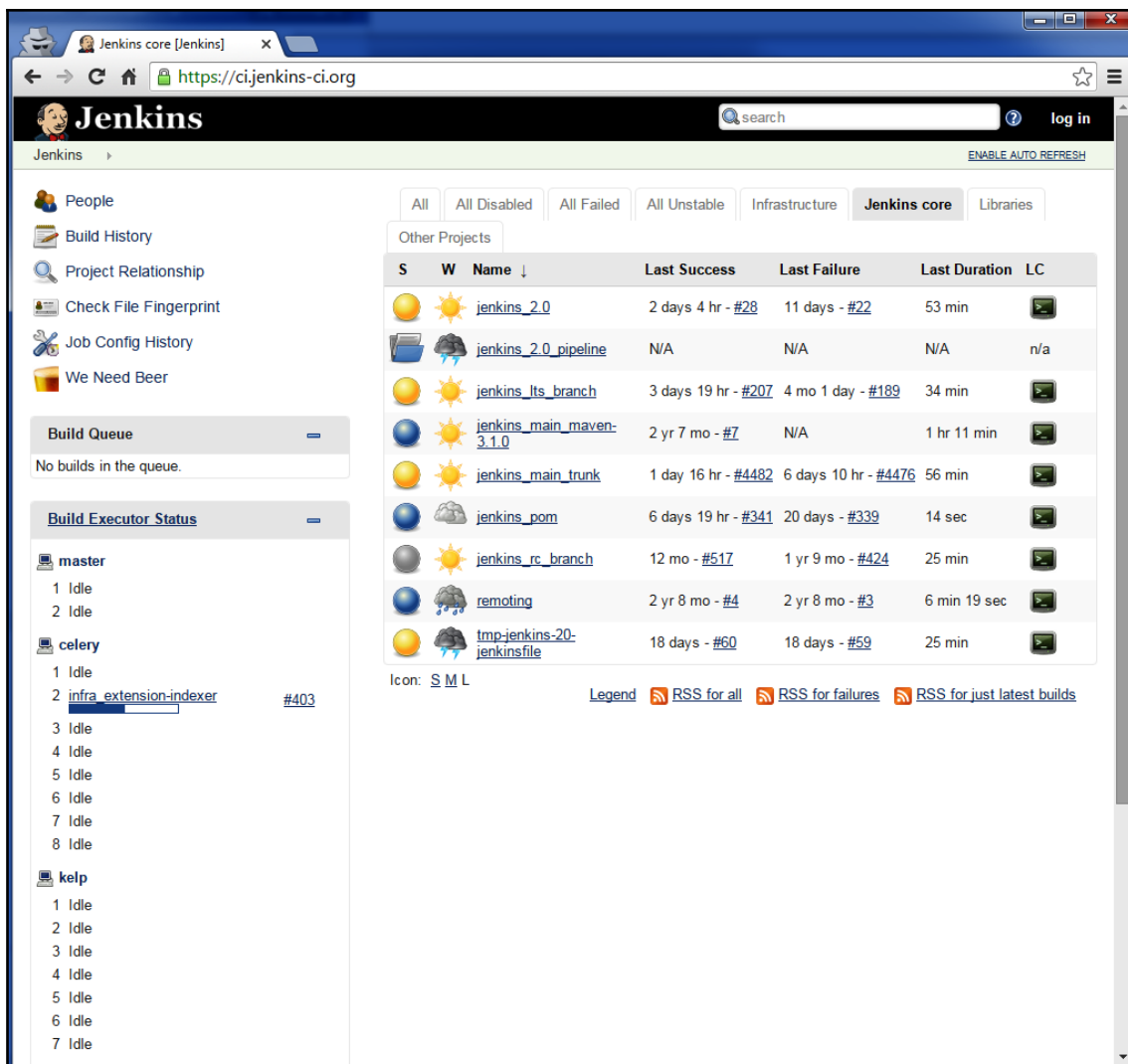
Figure 7: Preview of the Jenkins main interface

Jenkins is written in Java and was initially designed mainly for building projects written in the Java language. This means that for Java developers, it is a perfect CI system, but you may struggle a bit if you want to use it with another technology stack.

One big advantage of Jenkins is a very extensive list of features that Jenkins have implemented straight out-of-the-box. The most important one, from the Python programmer's point of view, is the ability to understand test results. Instead of giving only plain binary information about build success, Jenkins is able to present results of all tests that were executed during a run in the form of tables and graphs. This will, of course, not work automatically as you need to provide those results in a specific format at first (by default, Jenkins understands JUnit files) during your build. Fortunately, a lot of Python testing frameworks are able to export results in a machine-readable format.

The following is an example presentation of unit test results in Jenkins in its web UI:
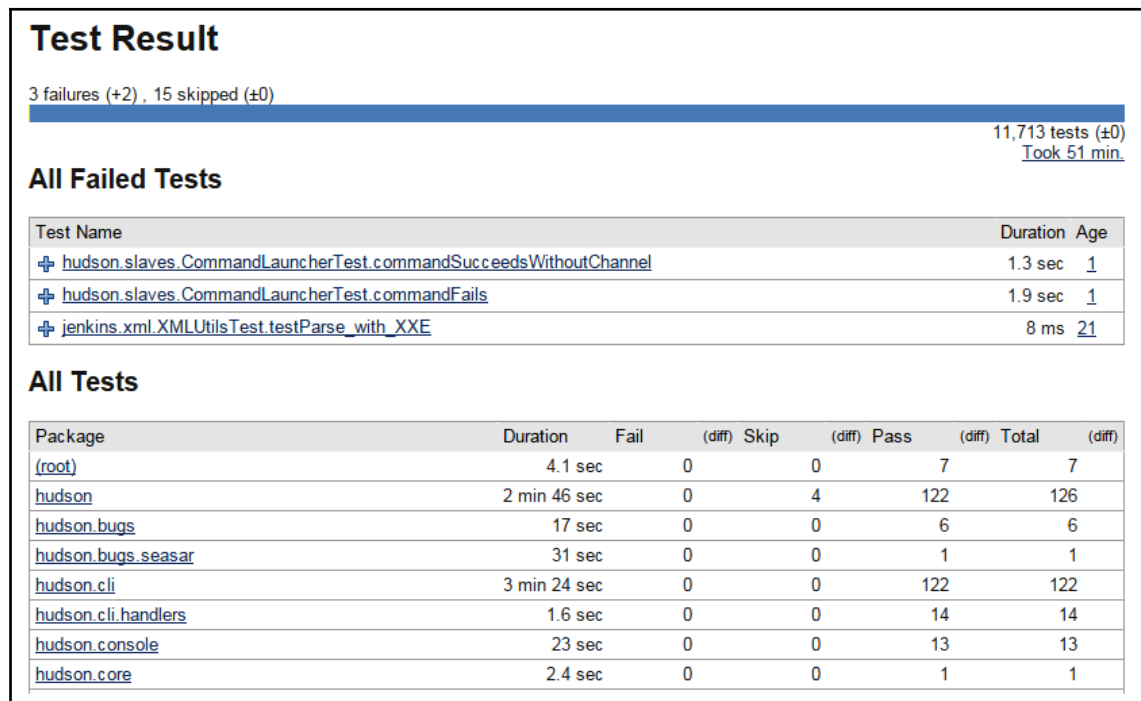
Figure 8: Presentation of unit test results in Jenkins

The following screenshot illustrates how Jenkins presents additional build information, such as trends or downloadable artefacts:
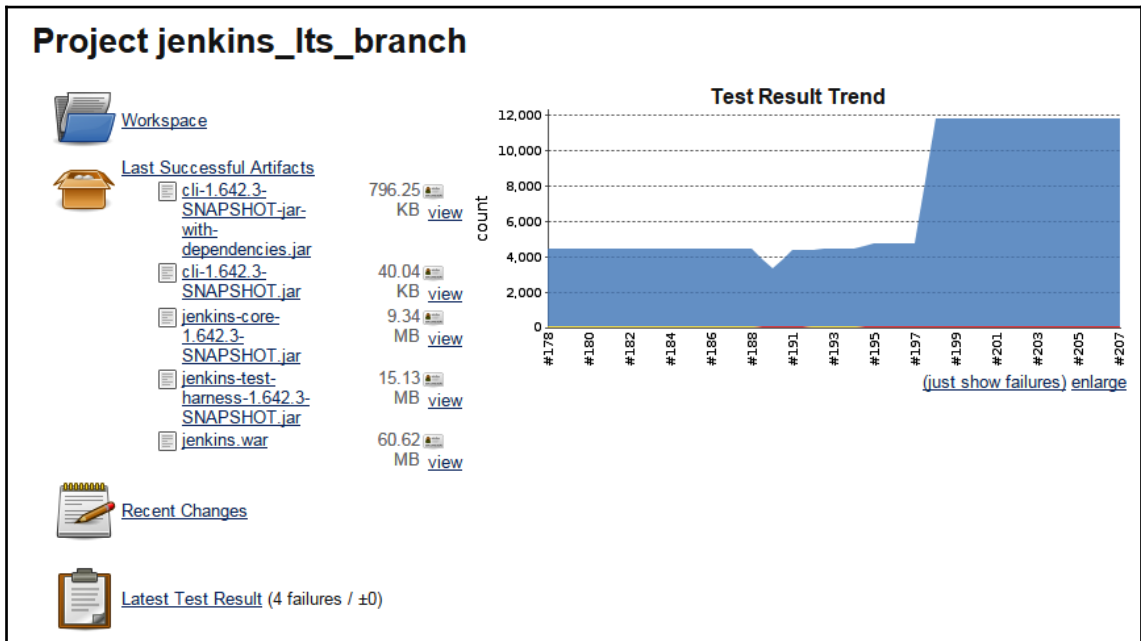
Figure 9: Test result trends graph of an example Jenkins project

Surprisingly, most of Jenkins' power does not come from its built-in features, but from a huge repository of free plugins. What is available from clean installation may be great for Java developers, but programmers using different technologies will need to spend a lot of time to make it suitable for their project. Even support for Git is provided by some plugin.

It is great that Jenkins is so easily extendable, but this also has some serious downsides. You will eventually depend on installed plugins to drive your continuous integration process, and these are developed independently from the Jenkins core. Most authors of popular plugins try to keep them up to date and compatible with the latest releases of Jenkins. Nevertheless, the extensions with smaller communities will be updated less frequently, and some day you may be either forced to resign from them or postpone the update of the core system. This may be a real problem when there is an urgent need for an update (a security fix, for instance), but some of the plugins that are critical for your CI process will not work with the new version.

The basic Jenkins installation that provides you with a master CI server is also capable of performing builds. This is different from other CI systems that put more emphasis on distribution and make strict separation from master and slave build servers. This is both good and bad. On the one hand, it allows you to set up a wholly working CI server in a few minutes. Jenkins, of course, supports deferring work to build slaves, so you can scale out in the future whenever it is needed. On the other hand, it is very common that Jenkins is underperforming because it is deployed in single server settings, and its users complain about performance because it is not providing enough resources. It is not hard to add new building nodes to the Jenkins cluster. It seems that this is a mental challenge rather than a technical problem for those that have got used to the single server setup.

In the next section, Buildbot is explained.

Buildbot

Buildbot (`http://buildbot.net/`) is software written in Python that automates the compile and test cycles for any kind of software projects. It is configurable in a way that every change made on a source code repository generates some builds and launches some tests, and then provides some feedback:
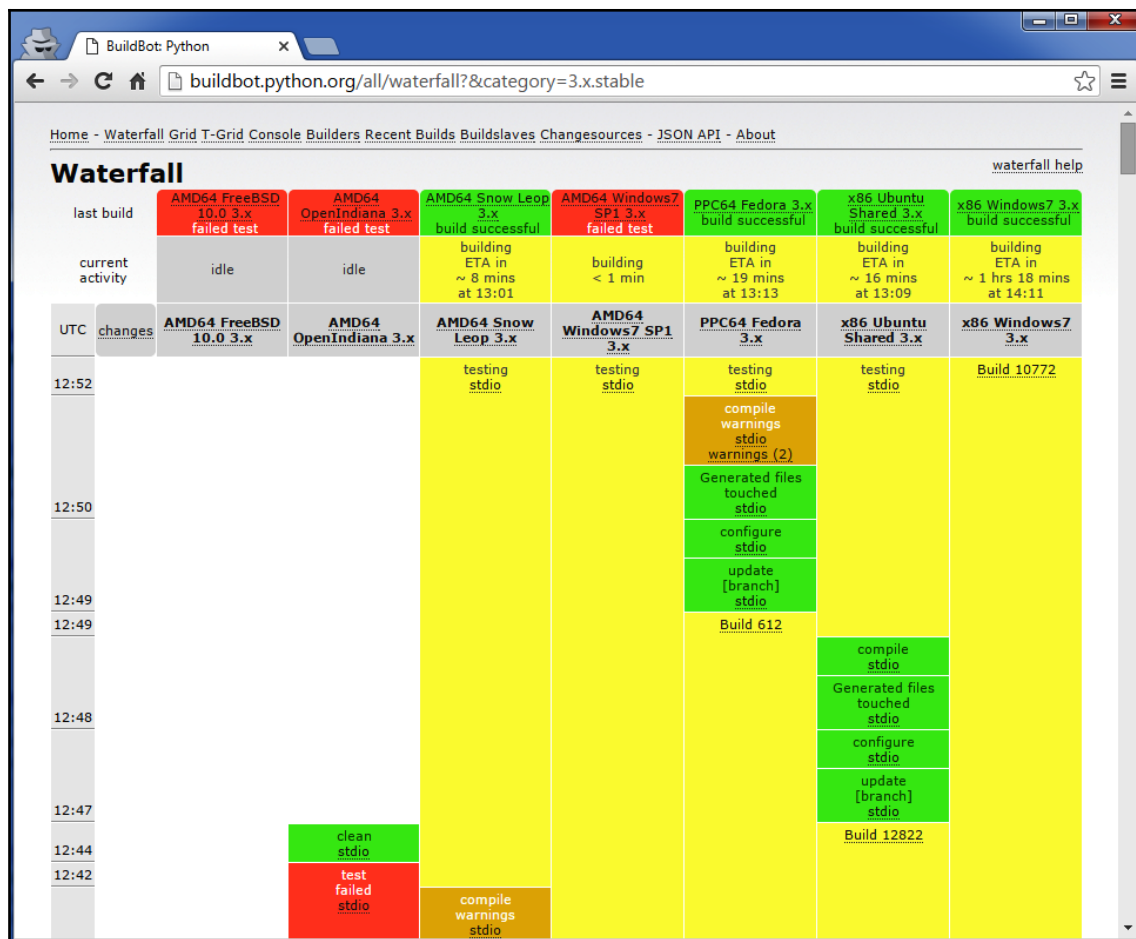
Figure 10: Buildbot's Waterfall view for the CPython 3.x branch

This tool is used, for instance, by CPython core, and can be found at `http://buildbot.python.org/all/waterfall?&category=3.x.stable`.

The default Buildbot's representation of build results is a Waterfall view, as shown in the preceding diagram. Each column corresponds to a **build** composed of **steps** and is associated with some **build slaves**. The whole system is driven by the build master as follows:

- The build master centralizes and drives everything
- A build is a sequence of steps used to build an application and run tests over it

- A **step** is an atomic command that does the following:
  - Checks out the files of a project
  - Builds the application
  - Runs tests

A build slave is a machine that is in charge of running a build. It can be located anywhere as long as it can reach the build master. Thanks to this architecture, Buildbot scales very well. All of the heavy lifting is done on build slaves, and you can have as many of them as you want.

Its very simple and clear design makes Buildbot very flexible. Each build step is just a single command. Buildbot is written in Python, but it is completely language agnostic. So, the build step can be absolutely anything. The process exit code is used to decide if the step ended as a success, and all standard output of the step command is captured by default. Most of the testing tools and compilers follow good design practices, and they indicate failures with proper exit codes and return readable error and warning messages on the `stdout` or `stderr` output streams. If that's not true, you can usually easily wrap them with simple Bash script. In most cases, this is a simple task. Thanks to this, a lot of projects can be integrated with Buildbot with only minimal effort.

The next advantage of Buildbot is that it supports the following version control systems out of the box, without the need to install any additional plugins:

- CVS
- Subversion
- Perforce
- Bzr
- Darcs
- Git
- Mercurial
- Monotone

The main disadvantage of Buildbot is its lack of higher-level presentation tools for presenting build results. For instance, other projects such as Jenkins can take the notion of unit tests being run during the build. If you feed them with the test results data presented in the proper format (usually XML), they can present all the tests in readable form, like tables and graphs. Buildbot does not have such a built-in feature, and this is the price it pays for its flexibility and simplicity. If you need some extra bells and whistles, you need to build them by yourself or search for some custom extensions. On the other hand, thanks to such simplicity, it is easier to reason about Buildbot's behavior and to maintain it. So, there is always a trade-off.

Let's take a look at Travis CI in the next section.

Travis CI

Travis CI (`https://travis-ci.org/`) is a continuous integration system sold in Software as a Service form. It is a paid service for enterprises, but can be used for free in open source projects hosted on GitHub:

Figure 11: Travis CI page for the django-userena project, showing failed builds in its build matrix

Naturally, it is the free part of its pricing plan that made it very popular. Currently, it is one of the most popular CI solutions for projects hosted on GitHub. But the biggest advantage over older projects, such as Buildbot or Jenkins, is how the build configuration is stored. All build definition is provided in a single `.travis.yml` file in the root of the project repository. Travis works only with GitHub, so if you have enabled such integration, your project will be tested on every commit if there is only a `.travis.yml` file.

Having the whole CI configuration for a project in its code repository is really a great approach. This makes the whole process a lot clearer for the developers and also allows for more flexibility. In systems where build configuration must be provided to build a server separately (using a web interface or through server configuration), there is always some additional friction when something new needs to be added to the testing rig. In some organizations where only selected staff are authorized to maintain the CI system, this really slows the process of adding new build steps down. Also, sometimes, there is a need to test different branches of the code with completely different procedures. When build configuration is available in project sources, it is a lot easier to do so.

The other great feature of Travis is the emphasis it puts on running builds in clean environments. Every build is executed in a completely fresh virtual machine, so there is no risk of some persisted state that would affect build results. Travis uses a rather big virtual machine image, so you have a lot of open source software and programming environments available without the need for additional installs. In this isolated environment, you have full administrative rights, so you can download and install anything you need to perform your build, and the syntax of the `.travis.yml` file makes this very easy. Unfortunately, you do not have a lot of choice of the operating system available as the base of your testing environment. Travis does not allow you to provide your own virtual machine images, so you must rely on the very limited options provided. Usually, there is no choice at all and all the builds must be done in some version of Ubuntu, macOS, or Windows (still experimental at the time of writing this book). Sometimes, there is an option to select some legacy version of one system or the preview of the new testing environment, but such a possibility is always temporary. There is always a way to bypass this. You can run another virtual machine or container inside of the one provided by Travis. This should be something that allows you to easily encode VM configuration in your project sources, such as Vagrant or Docker. But this will add more time to your builds, so it is not the best approach you could take. Stacking virtual machines this way may not be the best and most efficient approach if you need to perform tests under different operating systems. If this is an important feature for you, then this is a sign that Travis is not a service for you.

The biggest downside of Travis is that it is completely locked to GitHub. If you would like to use it in your open source project, then this is not a big deal. For enterprises and closed source projects, this is mostly an unsolvable issue.

In the next section, GitLab CI is explained.

GitLab CI

GitLab CI is a part of a larger GitLab project. It is available as both a paid service (Enterprise Edition) and an open source project that you may host on your own infrastructure (Community Edition). The open source edition lacks some of the paid service features but, in most cases, is everything that any company needs from the software that manages version control repositories and continuous integration.

GitLab CI is very similar in feature sets to Travis. It is even configured with very similar YAML syntax stored in the `.gitlab-ci.yml` file. The biggest difference is that the GitLab Enterprise Edition pricing model does not provide you with free accounts for open source projects. The Community Edition is open source by itself, but you need to have some own infrastructure in order to run it.

Compared with Travis, GitLab has the obvious advantage of having more control over the execution environment. Unfortunately, in the area of environment isolation, the default build runner in GitLab is a bit inferior. The process called GitLab Runner executes all the build steps in the same environment it is run in, so it works more like Jenkins' or Buildbot's slave servers. Fortunately, it plays well with Docker, so you can easily add more isolation with container-based virtualization, but this will require some effort and additional setup.

Choosing the right tool and common pitfalls

As stated previously, there is no perfect CI tool that will suit every project or, most importantly, every organization and workflow it uses. I can give only a single suggestion for open source projects hosted on GitHub. For small open source code bases with platform independent code, **Travis CI** seems like the best choice. It is easy to start with and will give you almost instant gratification with minimal amount of work.

For projects with closed sources, the situation is completely different. It is possible that you will need to evaluate a few CI systems in various setups until you are able decide which one is best for you. We discussed only four of the popular tools, but it should be seen as a rather representative group. To make your decision a bit easier, we will discuss some of the common problems related to continuous integration systems. In some of the available CI systems, it is more possible to make certain kinds of mistakes than in others. On the other hand, some of the problems may not be important to every application. I hope that by combining the knowledge of your needs with this short summary, it will be easier to make your first decision the right one.

Let's discuss the problems in the next sections.

Problem 1 – Complex build strategies

Some organizations like to formalize and structure things beyond the reasonable levels. In companies that create computer software, this is especially true in two areas: project management tools and build strategies on CI servers.

Excessive configuration of project management tools usually ends with issues processing workflows on JIRA (or any other management software) so becoming complicated that it will never fit a single wall when expressed as graphs, no matter how large this wall is. If your manager has such configuration mania, you can either talk to him or start looking for a new team. Unfortunately, this does not reliably ensure any improvement in that matter.

But when it comes to CI, we can do more. Continuous integration tools are usually maintained and configured by us: developers. These are OUR tools that are supposed to improve OUR work. If someone has an irresistible temptation to toggle every switch and turn every knob possible, then they should be kept away from the configuration of CI systems, especially if their main job is to talk the whole day and make decisions.

There is really no need for making complex strategies to decide which commit or branch should be tested. No need to limit testing to specific tags. No need to queue commits in order to perform larger builds. No need to disable building via custom commit messages. Your continuous integration process should be simple to reason about. Test everything! Test always! That's all! If there are not enough hardware resources to test every commit, then add more hardware. Remember that the programmer's time is more expensive than silicon.

Problem 2 – Long building time

Long building times is a thing that kills the performance of any developer. If you need to wait hours to know if your work was done properly, then there is no way you can be productive. Of course, having something else to do when your feature is being tested helps a lot. Anyway, as humans, we are really terrible at multitasking. Switching between different problems takes time and in the end reduces our programming performance to zero. It's simply hard to keep focus when working on multiple problems at once.

The solution is obvious: keep your builds fast at any price. At first, try to find bottlenecks and optimize them. If the performance of build servers is the problem, then try to scale out. If this does not help, then split each build into smaller parts and parallelize.

There are plenty of solutions to speed up a slow build, but sometimes nothing can be done about that problem. For instance, if you have automated browser tests or need to perform long-running calls to external services, then it is very hard to improve performance beyond some hard limit. For instance, when the speed of automated acceptance tests in your CI becomes a problem, then you can loosen the *test everything, test always* rule a bit. What matters the most for programmers are usually unit tests and static analysis. So, depending on your workflow, the slow browser tests may be sometimes deferred in time to the moment when the release is being prepared.

The other solution to slow build runs is rethinking the overall architecture design of your application. If testing the application takes a lot of time, it is often a sign that it should be split into a few independent components that can be developed and tested separately. Writing software as huge monoliths is one of the shortest paths to failure. Usually, any software engineering process breaks on software that is not modularized properly.

Problem 3 – External job definitions

Some continuous integration systems, especially Jenkins, allow you to set up most of the build configurations and testing processes completely through a web UI, without the need to touch the code repository. However, you should really avoid putting anything more than simple entry points to the build steps/commands into externals systems. This is the kind of CI anti-pattern that can cause nothing more than trouble.

Your building and testing process is usually tightly tied to your code base. If you store its whole definition in an external system, such as Jenkins or Buildbot, then it will be really hard to introduce changes to that process.

As an example of a problem introduced by a global external build definition, let's assume that we have some open source project. The initial development was hectic and we did not care for any style guidelines. Our project was successful, so the development required another major release. After some time, we moved from `0.x` version to `1.0` and decided to reformat all of our code to conform to PEP 8 guidelines. It is a good approach to have a static analysis check as part of CI builds, so we decided to add the execution of the `pep8` tool to our build definition. If we had only a global external build configuration, then there would be a problem if some improvement needs to be done to the code in older versions. Let's say that there is a critical security issue that needs to be fixed in both branches of the application: `0.x` and `1.y`. We know that anything below version 1.0 was compliant with the style guide and the newly introduced check against PEP 8 will mark the build as failed.

The solution to this problem is to keep the definition of your build process as close to the source as possible. With some CI systems (Travis CI and GitLab CI), you get that workflow by default. With other solutions (Jenkins and Buildbot), you need to take additional care in order to ensure that most of the build process definition is included in your code instead of some external tool configuration. Fortunately, you have the following choices that allow that kind of automation:

- Bash scripts
- Makefiles
- Python code

Problem 4 – Lack of isolation

We have discussed the importance of isolation when programming in Python many times already. We know that the best approach to isolate the Python execution environment on the application level is to use virtual environments with `virtualenv` or `python -m venv`. Unfortunately, when testing code for the purpose of continuous integration processes, it is usually not enough. The testing environment should be as close as possible to the production environment, and it is really hard to achieve that without additional system-level virtualization.

The following are the main issues you may experience when not ensuring proper system-level isolation when building your application:

- Some state persisted between builds either on the filesystem or in backing services (caches, databases, and so on)
- Multiple builds or tests interfacing with each other through the environment, filesystem, or backing services
- Problems that would occur due to specific characteristics of the production operating system not caught on the build server

These issues are particularly troublesome if you need to perform concurrent builds of the same application or even parallelize single builds.

Some Python frameworks (mostly Django) provide some additional level of isolation for databases that try to ensure that the storage will be cleaned before running tests. There is also quite a useful extension for `py.test` called `pytest-dbfixtures` (refer to `https://github.com/ClearcodeHQ/pytest-dbfixtures`) that allows you to achieve that even more reliably. Anyway, such solutions add even more complexity to your builds instead of reducing it. Using the *always fresh* virtual machines on every build (in the style of Travis CI) seems like a more elegant and simpler approach.

Summary

In this chapter, we took a look at the difference between centralized and distributed versions of control systems and why we should prefer a distributed version control system over the other. We saw why Git should be our first choice for DVCS. We also observed the common workflows and branching strategies for Git. Finally, we saw what is continuous integration/delivery/deployment is and what the popular tools that allow us to implement these processes are.

The next chapter will explain how to clearly document your code.

11
Documenting Your Project

Documentation is the work that is often neglected by developers and their managers. This is often due to lack of time toward the end of development cycles, and the fact that people think they are bad at writing. Even though some developers may not be very good at writing, the majority of them should be able to produce fine documentation.

The common result of neglecting the documentation efforts is a disorganized documentation landscape that is made up of documents written in a rush. Developers often hate doing this kind of work. Things get even worse when the existing documents need to be updated. Many projects out there are just providing poor, out of date documentation because no one in their team knows how to properly deal with it.

But setting up a documentation process at the beginning of the project and treating documents as if they were modules of code makes documenting easier. Writing can even be fun if you start following a few simple rules.

This chapter provides the following few tips to help document your projects:

- The seven rules of technical writing that summarize the best practices
- Writing documentation as a code
- Using documentation generators
- Writing self-documenting APIs

Technical requirements

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

- `Sphinx`
- `mkdocs`

You can install these packages using the following command:

```
python3 -m pip install <package-name>
```

The seven rules of technical writing

Writing good documentation is easier in many aspects than writing code, but many developers think otherwise. It will become easy once you start following a simple set of rules regarding technical writing.

We are not talking here about writing a novel or poems, but a comprehensive piece of text that can be used to understand software design, an API, or anything that makes up the code base.

Every developer can produce such material, and this section provides the following seven rules that can be applied in all cases:

- **Write in two steps**: Focus on ideas, and then on reviewing and shaping your text.
- **Target the readership**: Who is going to read it?
- **Use a simple style**: Keep it straight and simple. Use good grammar.
- **Limit the scope of the information**: Introduce one concept at a time.
- **Use realistic code examples**: *Foos* and *bars* should be avoided.
- **Use a light but sufficient approach**: You are not writing a book!
- **Use templates**: Help the readers get used to the common structure of your documents.

These rules are mostly inspired and adapted from *Agile Documentation: A Pattern Guide to Producing Lightweight Documents for Software Projects*, *Wiley*, a book by Andreas Rüping that focuses on producing the best documentation in software projects.

Write in two steps

Peter Elbow, in *Writing With Power: Techniques for Mastering the Writing Process*, *Oxford University Press*, explains that it is almost impossible for any human being to produce a perfect text in one shot. The problem is that many developers write documentation and try to directly come up with some perfect text. The only way they succeed in this exercise is by stopping the writing after every two sentences to read them back, and do some corrections. This means that they are focusing both on the content and the style of the text.

This is too hard for the brain, and the result is often not as good as it could be. A lot of time and energy is spent on polishing the style and shape of the text before its meaning is completely thought through.

Another approach is to drop the style and organization of the text and at first focus on its content. All ideas are laid down on paper, no matter how they are written. You start to write a continuous stream of thoughts and do not pause, even if you know that you are making obvious grammatical mistakes, or know that what you just wrote may read silly. At this stage, it does not matter if the sentences are barely understandable, as long as the ideas are written down. You just write down what you want to say, and apply only minimal structuring to your text.

By doing this, you focus only on what you want to say and will probably get more content out of your mind than you would initially expect.

Another side effect of doing this *free writing* is that other ideas that are not directly related to the topic will easily go through your mind. A good practice is to write them down on the side as soon as they appear, so they are not lost, and then get back to the main writing.

The second step obviously consists of reading back the draft of your document and polishing it so that it is comprehensible to everyone. Polishing a text means enhancing its style, correcting mistakes, reorganizing it a bit, and removing any redundant information it has.

A rule of thumb is that both steps should take an equal amount of time. If your time for writing documentation is strictly limited, then plan it accordingly.

Write in two steps
Focus on the content first, and then on style and cleanliness.

Target the readership

When writing content, there is a simple, but important, question the writer should consider: *Who is going to read it?*

This is not always obvious, as documentation is often written for every person that might get and use the code. The reader can be anyone from a researcher who is looking for an appropriate technical solution to their problem, or a developer who needs to implement a new feature in the documented software.

Good documentation should follow a simple rule—each text should target one kind of reader. This philosophy makes the writing easier, as you will precisely know what kind of reader you're dealing with.

A good practice is to provide a small introductory document that explains in one sentence what the documentation is about, and guides different readers to the appropriate parts of documentation, for example:

Atomisator is a product that fetches RSS feeds and saves them in a database, with a filtering process.

If you are a developer, you might want to look at the API description (api.txt).

If you are a manager, you can read the features list and the FAQ (features.txt).

If you are a designer, you can read the architecture and infrastructure notes (arch.txt).

Know your readership before you start to write.

Use a simple style

Simple things are easier to understand. That's a fact.

By keeping sentences short and simple, your writing will require less cognitive effort for their content to be extracted, processed, and then understood. Writing technical documentation aims to provide a software guide to readers. It is not a fiction book, and should be closer to your microwave operation manual than to a Dickens novel.

The following are a few tips to keep in mind:

- Use short sentences. They should be no longer than 100–120 characters (including spaces). This is the length of two lines in a typical paperback.
- Each paragraph should be composed of three to four sentences at most, which express one main idea. Let your text breathe.
- Don't repeat yourself too much. Avoid journalistic styles where ideas are repeated again and again to make sure they are understood.
- Don't use several tenses. The present tense is enough most of the time.

- Do not make jokes in the text if you are not a really fine writer. Being funny in a technical book is really hard, and few writers master it. If you really want to distill some humor, keep it in code examples and you will be fine.

> You are not writing fiction; keep the style as simple as possible.

Limit the scope of information

There's a simple sign of bad documentation in software—you cannot find specific information in it, even if you're sure that it is there. After spending some time reading the table of contents, you are starting to search through text files using `grep` with several word combinations and still cannot find what you are looking for. But you're sure the information is there because you saw it once.

This often happens when writers do not organize their texts well with meaningful titles and headings. They might provide tons of information, but it won't be useful if the reader is not able to scan through all the documentation for a specific topic.

In a good document, paragraphs should be gathered under a meaningful heading for a given section, and the document title should synthesize the content in a short phrase. A table of contents could be made of all the sections' titles, in order to help the reader scan through the document.

> A simple yet effective practice to compose your titles and headings is to ask yourself, "*What phrase would I type in Google to find this section?*"

Use realistic code examples

Unrealistic code examples simply make your documentation harder to understand.

For instance, if you have to provide some string literals, the *Foos* and *bars* are really bad choices. If you have to show your reader how to use your code, why not to use a real-world example? A common practice is to make sure that each code example can be cut and pasted into a real program.

To show an example of bad usage, let's assume we want to show how to use the `parse()` function from the `atomisator` project, which aims to parse RSS feeds. Here is the usage example using an unrealistic imaginary source:

```
>>> from atomisator.parser import parse
>>> # Let's use it:
>>> stuff = parse('some-feed.xml')
>>> next(stuff)
{'title': 'foo', 'content': 'blabla'}
```

A better example, such as the following, would be using a data source that looks like a valid URL to RSS feed and shows output that resembles the real article:

```
>>> from atomisator.parser import parse
>>> # Let's use it:
>>> my_feed = parse('http://tarekziade.wordpress.com/feed')
>>> next(my_feed)
{'title': 'eight tips to start with python', 'content': 'The first tip
is..., ...'}
```

This slight difference might sound like overkill but, in fact, makes your documentation a lot more useful. A reader can copy those lines into a shell, understand that `parse()` expects a URL as a parameter, and that it returns an iterator that contains web articles.

Of course, giving a realistic example is not always possible or viable. This is especially true for very generic code. Even this book has a few occurrences of vague `"foo"` and `"bar"` strings. Anyway, you should always strive to reduce the amount of such unrealistic examples to a minimum.

> Code examples should be directly reusable in real programs.

Use a light but sufficient approach

In most agile methodologies, documentation is not the first citizen. Making software that just works is more important than the detailed documentation. So, a good practice, as Scott Ambler explains in his book *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*, *John Wiley & Sons*, is to define the real documentation needs, rather than try to document everything possible.

For instance, let's look at some example documentation of a simple project that is available on GitHub. `ianitor` (available at `https://github.com/ClearcodeHQ/ianitor`) is a tool that helps to register processes in the Consul service discovery cluster, and it is mostly aimed at system administrators. If you take a look at its documentation, you will realize that this is just a single document (the `README.md` file). It explains only how it works and how to use it. From the administrator's perspective, this is sufficient. They only need to know how to configure and run the tool, and there is no other group of people expected to use `ianitor`. This document limits its scope by answering one question, "*How do I use ianitor on my server?*"

Use templates

Many pages on Wikipedia look similar. There are boxes on the right-hand side that are used to summarize some information for documents belonging to the same area. The first section of the article usually contains a table of contents with links that refer to anchors in the same text. There is always a reference section at the end.

Users get used to it. For instance, they know they can have a quick look at the table of contents, and if they do not find the information they are looking for, they will go directly to the reference section to see if they can find another website on the topic. This works for any page on Wikipedia. Once you learn the format of *Wikipedia* articles, you become more efficient in finding useful information.

So, using templates forces a common pattern for documents, and therefore enables more efficient searching for information. Users get used to the common structure of information and know how to read it quickly.

Providing a template for each kind of document also provides a quick start for writers.

Documentation as code

The best way to keep the documentation of your project up to date is to treat it as code and store it in the same repository as the source code it documents. Keeping documentation sources with the source code has the following benefits:

- With a proper version control system, you can track all changes that were made to the documentation. If you ever wonder if a particular surprising code behavior is really a bug or just an old and forgotten feature, you can dive into the history of the documentation to trace how the documentation for the specific feature evolved over time.

- It is easier to develop different versions of the documentation if the project has to be maintained on several parallel branches (for example, for different clients). If the source code of the project diverges from the main development branch, so does the documentation for it.
- There are many tools that allow you to generate the reference documentation of software APIs straight from the comments included in the source code. This is one of the best ways to generate documentation for projects that provide APIs for other components (for example, in the form of reusable libraries and remote services).

The Python language has some unique qualities that make documenting software extremely easy and fun. The Python community also provides a huge selection of tools that allow you to create beautiful and usable API reference documentation straight from Python sources. The foundation for these tools are so-called docstrings.

Using Python docstrings

Docstrings are special Python string literals that are intended for documenting Python functions, methods, classes, and modules. If the first statement of the function, method, class, or module is a string literal, it will automatically become a docstring and be included as a value of the __doc__ attribute of that related function, method, class, or module.

Many of the code examples in this book already feature docstrings, but for the sake of consistency, let's look at a general example of a module that contains all possible types of docstrings, as follows:

```
"""Example module with doctrings.

This is a module that shows all four types of docstrings:
- module docstring
- function docstring
- method docstring
- class docstring
"""


def show_module_documentation():
    """Prints module documentation.

    Module documentation is available as global __doc__ attribute.
    This attribute can be accessed and modified at any time.
    """
    print(__doc__)
```

```
class DocumentedClass:
    """Class that showcases method documentation.
    """

    def __init__(self):
        """Initialize class instance.
        Interesting note: docstrings are valid statements.
        It means that if function or method doesn't have to
        do nothing and has docstring it doesn't have to
        feature any other statements.

        Such no-op functions are useful for defining abstract
        methods or providing implementation stubs that have
        to be implemented later.
        """
```

Python also provides a `help()` function, which is an entry point for the built-in help system. It is intended for interactive use within the interactive interpreter session in a similar way as viewing system manual pages using the UNIX man `command`. If you provide a module instance as an input argument to the `help()` function, it will format all docstrings of that module's objects in a tree-like structure. The following is an example of `help()` output for the module we presented in the previous code snippet:

```
Help on module docexample:

NAME
    docexample - Example module with doctrings.

FILE
    /Users/swistakm/docexample.py

DESCRIPTION
    This is a module that shows all four types of docstrings:
    - module docstring
    - function docstring
    - method docstring
    - class docstring

CLASSES
    DocumentedClass

    class DocumentedClass
     |  Class that showcases method documentation.
     |
     |  Methods defined here:
     |
     |  __init__(self)
```

```
    |         Initialize class instance.
    |
    |         Interesting note: docstrings are valid statements.
    |         It means that if function or method doesn't have to
    |         do nothing and has docstring it doesn't have to
    |         feature any other statements.
    |
    |         Such no-op functions are useful for defining abstract
    |         methods or providing implementation stubs that have
    |         to be implemented later.

FUNCTIONS
    show_module_documentation()
        Prints module documentation.

        Module documentation is available as global __doc__ attribute.
        This attribute can be accessed and modified at any time.
```

Popular markup languages and styles for documentation

Inside docstring, you can put whatever you like in any form you like. There is, of course, the official *PEP 257 (Docstring Conventions)* document, which is a general guideline for docstring conventions, but it concentrates mainly on normalized formatting of multiline string literals for documentation purposes and does not enforce any markup language.

 Anyway, if you want to have nice and usable documentation, it is a good thing to decide on some formalized markup language to use in your docstrings, especially if you plan to use some kind of documentation generation tool. Proper markup allows documentation generators to provide code highlighting, do advanced text formatting, include hyperlinks to other documents and functions, or even include non-textual assets like images of automatically generated class diagrams.

The best markup language is easy to write and is also readable in raw textual form outside of the autogenerated reference documentation. It is best if it can be easily used to provide longer documentation sources for documents living outside of Python docstrings. One of the most common markup languages designed specifically for Python with these goals in mind is reStructuredText. It is used by the Sphinx documentation system and is a markup language used to create official Python language documentation. The basic syntax elements of this markup are described in `Appendix A`, *reStructuredText Primer.*

Other popular choices for lightweight text markup languages for docstrings are Markdown and AsciiDoc. The former is particularly popular within the community of GitHub users and is the most common documentation markup language in general. It is also often supported out of the box by various tools for self-documenting web APIs.

Popular documentation generators for Python libraries

As stated previously, software documentation may have varied readership. Accessing documentation directly from project source code is often natural to users that are programmers developing a given project. But this way of accessing project documentation may not be the most convenient for others. Also, some companies may have requirements to deliver documentation to their clients in a printable form.

This is why documentation generation tools are so important. They allow you to benefit from documentation being treated as code while still maintaining the ability to have a deliverable document that can be browsed, searched, and read without access to the original source code. The Python ecosystem comes with a variety of amazing open source tools that allow you to generate project documentation directly from your source code. The two most popular tools in the Python community for generating user-friendly documentations are Sphinx and MkDocs. We will discuss them briefly in the following sections.

Sphinx

Sphinx (`http://sphinx.pocoo.org`) is a set of scripts and `docutils` extensions that can be used to generate an HTML structure from the tree of plain text documents that are created using the reStructuredText syntax language (you can find more details on that markup language in `Appendix A`, *reStructuredText Primer)*. Sphinx also supports multiple other documentation output formats, like man pages, PDF, or even LaTex. This tool is used (for instance) to build official Python documentation and is very popular among many open source Python projects. It provides a really nice browsing system, together with a light but sufficient client-side JavaScript search engine. It also uses `pygments` for rendering code examples, which produces really nice syntax highlights.

Sphinx can be easily configured to stick with the document landscape we defined in the previous section. It can be easily installed with `pip` as a `Sphinx` package.

The easiest way to start working with Sphinx is to use the `sphinx-quickstart` script. This utility will generate a script together with `Makefile`, which can be used to generate the web documentation every time it is needed. It will interactively ask you some questions and then bootstrap the whole initial documentation source tree and configuration file. Once it is done, you can easily tweak it whenever you want. Let's assume we have already bootstrapped the whole Sphinx environment and we want to see its HTML representation. This can be easily done using the `make html` command, as follows:

```
project/docs$ make html
sphinx-build -b html -d _build/doctrees   . _build/html
Running Sphinx v1.3.6
making output directory...
loading pickled environment... not yet created
building [mo]: targets for 0 po files that are out of date
building [html]: targets for 1 source files that are out of date
updating environment: 1 added, 0 changed, 0 removed
reading sources... [100%] index
looking for now-outdated files... none found
pickling environment... done
checking consistency... done
preparing documents... done
writing output... [100%] index
generating indices... genindex
writing additional pages... search
copying static files... done
copying extra files... done
dumping search index in English (code: en) ... done
dumping object inventory... done
build succeeded.
Build finished. The HTML pages are in _build/html.
```

The following screenshot shows an example HTML version of documentation built with Sphinx:
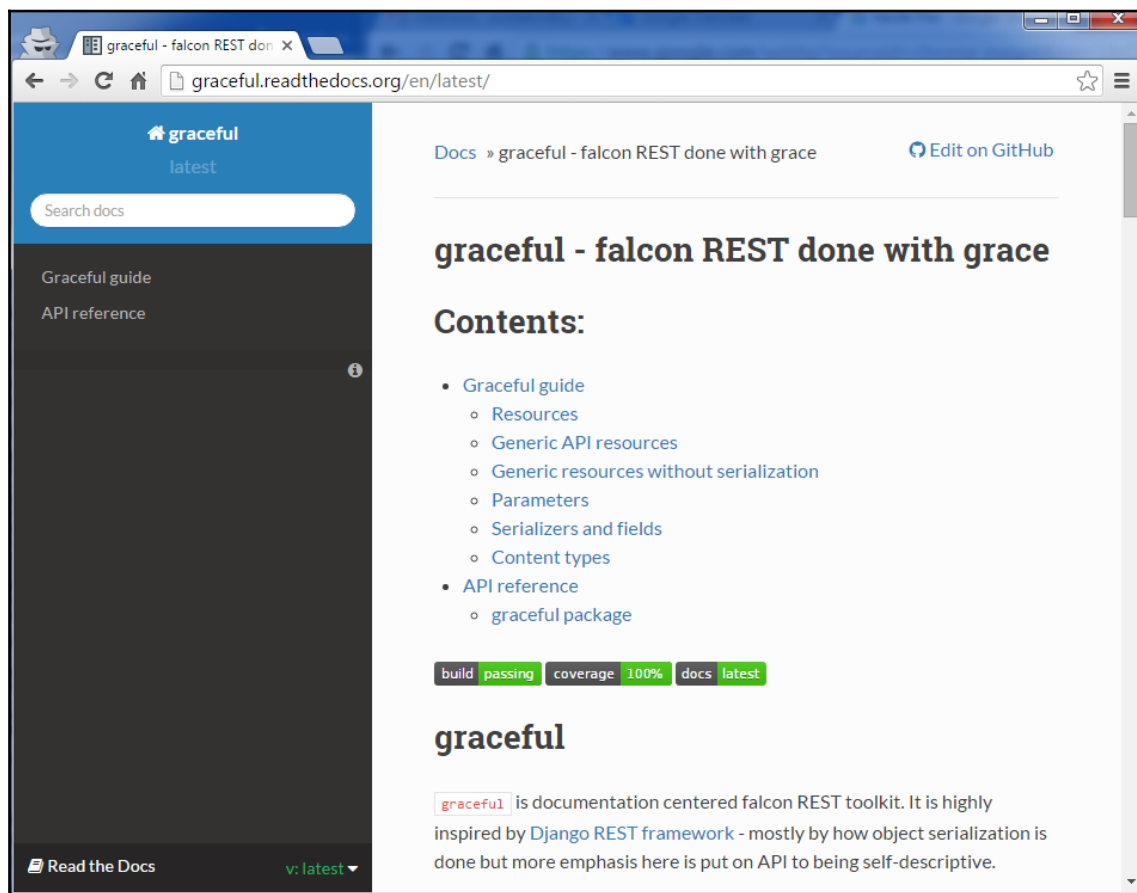
Besides the HTML versions of the documents, the tool also builds automatic pages, such as a module list and an index. Sphinx provides a few `docutils` extensions to drive these features. These are the main ones:

- A directive that builds a table of contents
- A marker that can be used to register a document as a module helper
- A marker to add an element in the index

Working with the index pages

Sphinx provides a `toctree` directive that can be used to inject a table of contents in a document, with links to other documents. Each line must be a file with its relative path, starting from the current document. Glob-style names can also be provided to add several files that match the expression.

For example, the index file in the `cookbook` folder, which we previously defined in the producer's landscape, can look like this:

```
========
Cookbook
========

Welcome to the Cookbook.

Available recipes:

.. toctree::
   :glob:
   *
```

With this syntax, the HTML page will display a list of all the reStructuredText documents available in the `cookbook` folder. This directive can be used in all the index files to build browsable documentation.

Registering module helpers

For module helpers, a marker can be added so that it is automatically listed and available in the module's index page, as follows:

```
=======
session
=======

.. module:: db.session

The module session...
```

Notice that the `db` prefix here can be used to avoid module collision. Sphinx will use it as a module category and will group all modules that start with `db.` in this category.

Adding index markers

Another option can be used to fill the index page by linking the document to an entry, as follows:

```
=======
session
=======

.. module:: db.session

.. index::
   Database Access
   Session

The module session...
```

Two new entries, `Database Access` and `Session`, will be added in the index page.

Cross-references

Finally, Sphinx provides an inline markup to set cross-references. For instance, a link to a module can be done like this:

```
:mod:`db.session`
```

Here, `:mod:` is the module marker's prefix and `` `db.session` `` is the name of the module to be linked to (as registered previously). Keep in mind that `:mod:`, as well as the previous elements, are the specific directives that were introduced in reStructuredText by Sphinx.

> Sphinx provides a lot more features that you can discover on its website. For instance, the *autodoc* feature is a great option to automatically extract your doctests to build the documentation. For more information, refer to `http://sphinx.pocoo.org`.

MkDocs

MkDocs (`https://www.mkdocs.org/`) is a very minimalistic static page generator that can be used to document your projects. It lacks built-in *autodoc* features, similar to those in Sphinx, but uses the lot simpler and readable Markdown markup language. It is also really extensible. It is definitely easier to write a MkDocs plugin than a docutils extension that could be used by Sphinx. So, if you have very specific documentation needs that cannot be satisfied by existing tools and their extensions are available at the moment, then MkDocs provides a very good foundation for building something custom-tailored.

Documentation building and continuous integration

Sphinx and similar documentation generation tools really improve the readability and experience of reading the documentation from the consumer's point of view. As we stated previously, it is especially helpful when some of the documentation parts are tightly coupled to the code, as in the form of docstrings. While this approach really makes it easier to ensure that the source version of the documentation matches with the code it documents, it does not guarantee that the documentation readership will have access to the latest and most up-to-date compiled version.

Having only bare source representation is also not enough if the target readers of the documentation are not proficient enough with command-line tools and will not know how to build it into a browsable and readable form. This is why it is important to build your documentation into a consumer-friendly form automatically whenever any change to the code repository is committed/pushed.

The best way to host the documentation built with Sphinx is to generate an HTML build and serve it as a static resource with your web server of choice. Sphinx provides a proper `Makefile` to build HTML files with the `make html` command. Because make is a very common utility, it should be very easy to integrate this process with any of the continuous integration systems we discussed in `Chapter 10`, *Managing Code*.

If you are documenting an open source project with Sphinx, then you will make your life a lot easier by using **Read the Docs** (`https://readthedocs.org/`). It is a free service for hosting the documentation of open source Python projects with Sphinx. The configuration is completely hassle-free, and it integrates very easily with two popular code hosting services: GitHub and Bitbucket. In practice, if you have your accounts properly connected and code repository properly set up, enabling documentation hosting on Read the Docs is a matter of just a few clicks.

Documenting web APIs

The principles for documenting web APIs are almost the same as for other kinds of software. You want to properly target your readership, provide documentation in a way and form that is native for the usage environment (here, as a web page), and, most of all, make sure that readers have access to the up to date and relevant version of your documentation.

Because of this, it is extremely important to have your documentation of web APIs generated from the sources of the code that provides these APIs. Unfortunately, due to the complex architecture of most web frameworks, classical documentation tools like Sphinx are rarely useful for documenting typical HTTP endpoints of web APIs. In this context, it is very common that auto-documentation capabilities are built into your web framework of choice. These kind of frameworks either serve user-readable documentation by themselves or serve a standardized API description in a machine-readable format that can be later processed with a specialized documentation browser.

There is also another completely different philosophy for documenting web APIs, and it is based on the idea of API prototyping. Tools for API prototyping allow you to use documentation as a software contract that can be used as an API stub, even before service development starts. Often, this kind of tool allows you to automatically verify if the API structure matches the one actually implemented in the service. In this approach, documentation may serve the additional function of an API testing tool.

Documentation as API prototype with API Blueprint

API Blueprint is a web API description language that is both human-readable and well-defined. You can think of it like a Markdown for web service description language. It allows documenting anything from the structure of URL paths, through body structures of HTTP request/responses and headers, to complex request-response exchanges. The following is an example of an imaginary Cat API described using API Blueprint:

```
FORMAT: 1A
HOST: https://cats-api.example.com

# Cat API
This API Blueprint demonstrates example documentation of some imaginary
Cat API.

# Group Posts
```

```
This section groups Cat resources.

## Cat [/cats/{cat_id}]

A Cat is central and only resource utilized by Cat API.

+ Parameters
    + cat_id: `1` (string) - The id of the Cat.

+ Model (application/json)

    ```js
    {
        "data": {
            "id": "1", // note this is a string
            "breed": "Maine Coon",
            "name": "Smokey"
    ```

Retrieve a Cat [GET]

Returns a specific Cat.

+ Response 200

    [Cat][]

Create a Cat [POST]

Create a new Post object. Mentions and hashtags will be parsed out of the
post
text, as will bare URLs...

+ Request

    [Cat][]

+ Response 201

    [Cat][]
```

API Blueprint alone is nothing more than a language. Its strength really comes from the fact that it can be easily written by hand and from the huge selection of tools supporting that language. At the time of writing this book, the official API Blueprint page lists over 70 tools that support this language. Some of these tools can even generate functional API mock servers that are meant to shorten development cycles, as mock servers can be used, for instance, by frontend code, even before programmers start the development of backend API services.

Self-documenting APIs with Swagger/OpenAPI

While self-documenting APIs is a more traditional approach for documenting web APIs (compared to documenting through API prototypes), we can clearly see some interesting trends that appeared during the past few years. In the past, when API frameworks had to support auto-documentation capabilities, it almost always meant that the framework had a built-in API metadata structure with a custom documentation rendering engine. If someone wanted to have multiple services auto-documented, they had to use the same framework for every service, or decide to have very a inconsistent documentation landscape.

With the advent of microservice architectures, this approach becomes extremely inconvenient and inefficient. Nowadays, it's very common that services within the same projects are written using different frameworks, libraries, and even using completely different programming languages. Having different documentation libraries for every framework and language would produce very inconsistent documentation, as every tool would have different strengths and weaknesses.

One approach that solves this problem requires splitting the documentation display (rendering and browsing) from the actual documentation definition. This approach is analogous to API prototyping because it requires a standardized API definition language. But here, the developer rarely uses this language explicitly. It is the framework's responsibility to create a machine-readable API definition from the structure of the code written with this framework.

One such machine-readable web API description languages is OpenAPI. The specification of OpenAPI is the result of the development of the popular Swagger documentation tool. At first, it was an internal metadata format of the Swagger tool, but once it became standardized, many tools around that specification appeared. With OpenAPI, many web frameworks can describe their API structure using the same metadata format, so their documentation can be rendered in the same consistent form by a single documentation browser.

Building a well-organized documentation system

An easier way to guide your documentation readers and your writers is to provide each one of them with helpers and guidelines, as we have learned in the previous section of this chapter.

From a writer's point of view, this is done by having a set of reusable templates, together with a guide that describes how and when to use them in a project. This is called a **documentation portfolio**.

From a reader's point of view, it is important to be able to browse the documentation with no pain, and get used to finding the information efficiently. This is done by building a **document landscape**.

Obviously, we need to start from guiding documentation writers, because without them, the readers would not have anything to read. Let's see how such a portfolio looks and how to build a one.

Building documentation portfolio

There are many kinds of documents a software project can have, from low-level documents that refer directly to the code, to design papers that provide a high-level overview of the application.

For instance, Scott Ambler defines an extensive list of document types in his book *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*, *John Wiley & Sons*. He builds a portfolio from early specifications to operations documents. Even the project management documents are covered, so the whole documenting needs are built with a standardized set of templates.

Since a complete portfolio is tightly related to the methodologies used to build the software, this chapter will only focus on a common subset that you can complete with your specific needs. Building an efficient portfolio takes a long time, as it captures your working habits.

A common set of documents in software projects can be classified into the following three categories:

- **Design**: This includes all the documents that provide architectural information and low-level design information, such as class diagrams or database diagrams

- **Usage**: This includes all the documents on how to use the software; this can be in the shape of a cookbook and tutorials, or a module-level help
- **Operations**: This provides guidelines on how to deploy, upgrade, or operate the software

Let's discuss the preceding categories.

Design

The important point when creating such documents is to make sure the target readership is perfectly known, and that the content scope is limited. So, a generic template for design documents can provide a light structure with a little advice for the writer.

Such a structure might include the following:

- Title
- Author
- Tags (keywords)
- Description (abstract)
- Target (who should read this?)
- Content (with diagrams)
- References to other documents

The content should be three or four pages at most when printed, so be sure to limit the scope. If it gets bigger, it should be split into several documents or summarized.

The template also provides the author's name and a list of tags to manage its evolutions and ease its classification. This will be covered later in this chapter.

The example design document template written using reStructuredText markup could be as follows:

```
========================================
Design document title
========================================

:Author: Document Author
:Tags: document tags separated with spaces

:abstract:

    Write here a small abstract about your design document.
```

```
.. contents ::


Audience
========

Explain here who is the target readership.


Content
=======

Write your document here. Do not hesitate to split it in several
sections.


References
==========

Put here references, and links to other documents.
```

Usage

The usage documentation describes how a particular part of the software works. This documentation can describe low-level parts, such as how a function works, but also high-level parts, such as command-line arguments for calling the program. This is the most important part of documentation in framework applications, since the target readership is mainly the developers that are going to reuse the code.

The three main kinds of documents are as follows:

- **Recipe**: This is a short document that explains how to do something. This kind of document targets one readership and focuses on one specific topic.
- **Tutorial**: This is a step-by-step document that explains how to use a feature of the software. This document can refer to recipes, and each instance is intended for one readership.
- **Module helper**: This is a low-level document that explains what a module contains. This document can be shown (for instance) by calling the `help` built into over a module.

Recipe

A recipe answers a very specific problem and provides a solution to resolve it. For example, ActiveState provides a huge repository of Python recipes online, where developers can describe how to do something in Python (refer to `http://code.activestate.com/recipes/langs/python/`). Such a set of recipes related to a single area/project is often called a *cookbook*.

These recipes must be short and are structured, like this:

- Title
- Submitter
- Last updated
- Version
- Category
- Description
- Source (the source code)
- Discussion (the text explaining the code)
- Comments (from the web)

Often, they are one screen long and do not go into great detail. This structure perfectly fits a software's needs and can be adapted in a generic structure, where the target readership is added and the category is replaced by tags:

- Title (short sentence)
- Author
- Tags (keywords)
- Who should read this?
- Prerequisites (other documents to read, for example)
- Problem (a short description)
- Solution (the main text, one or two screens)
- References (links to other documents)

The date and version are not useful here, since project documentation should be managed like source code in the project. This means that the best way to handle the documentation is to manage it through the version control system. In most cases, this is exactly the same code repository as the one that's used for the project's code.

A simple reusable template for the recipes could be as follows:

```
===========
Recipe name
===========

:Author: Recipe Author
:Tags: document tags separated with spaces

:abstract:

    Write here a small abstract about your design document.

.. contents ::


Audience
========

Explain here who is the target readership.


Prerequisites
=============

Write the list of prerequisites for implementing this recipe. This can be
additional documents, software, specific libraries, environment settings or
just anything that is required beyond the obvious language interpreter.


Problem
=======

Explain the problem that this recipe is trying to solve.


Solution
========

Give solution to problem explained earlier. This is the core of a recipe.


References
==========

Put here references, and links to other documents.
```

Tutorial

A tutorial differs from a recipe in its purpose. It is not intended to resolve an isolated problem, but rather describes how to use a feature of the application, step by step. This can be longer than a recipe and can concern many parts of the application. For example, Django provides a list of tutorials on its website. *Writing your first Django App, part 1* (refer to `https://docs.djangoproject.com/en/1.9/intro/tutorial01/`) explains in a few screens how to build an application with Django.

A structure for such a document will be as follows:

- Title (short sentence)
- Author
- Tags (words)
- Description (abstract)
- Who should read this?
- Prerequisites (other documents to read, for example)
- Tutorial (the main text)
- References (links to other documents)

Module helper

The last template that can be added in our collection is the module helper template. A module helper refers to a single module and provides a description of its contents, together with usage examples.

Some tools can automatically build such documents by extracting the docstrings and computing module help using `pydoc`, such as Epydoc (refer to `http://epydoc.sourceforge.net`). So, it is possible to generate extensive documentation based on API introspection. This kind of documentation is often provided in Python frameworks. For instance, Plone provides a server that keeps an up-to-date collection of module helpers. You can read more about it at `http://api.plone.org`.

The following are the main problems with this approach:

- There is no smart selection performed over the modules that are really interesting to the document
- The code can be obfuscated by the documentation

Furthermore, a module documentation provides examples that sometimes refer to several parts of the module, and are hard to split between the functions' and classes' docstrings. The module docstring could be used for that purpose by writing text at the top of the module. But this ends in having a hybrid file composed of a block of text, then a block of code. This is rather obfuscating when the code represents less than 50% of the total length. If you are the author, this is perfectly fine. But when people try to read the code (not the documentation), they will have to skip the docstrings part.

Another approach is to separate the text in its own file. A manual selection can then be operated to decide which Python module will have its module helper file. The documents can then be separated from the code base and allowed to live their own life, as we will see in the next section. This is how Python is documented.

Many developers will disagree on the fact that doc and code separation is better than docstrings. This approach means that the documentation process is fully integrated in the development cycle; otherwise, it will quickly become obsolete. The docstrings approach solves this problem by providing proximity between the code and its usage example, but doesn't bring it to a higher level—a document that can be used as part of plain documentation.

The following template for Module Helper is really simple as it contains just a little metadata before the content is written. The target is not defined since it is the developers who wish to use the module:

- Title (module name)
- Author
- Tags (words)
- Content

> The next chapter will cover **test-driven development** (**TDD**) using doctests and module helpers.

Operations

Operation documents are used to describe how the software can be operated. Consider the following points:

- Installation and deployment documents
- Administration documents

- Frequently Asked Questions (FAQ) documents
- Documents that explain how people can contribute, ask for help, or provide feedback

These documents are very specific, but they can probably use the tutorial template we defined in the earlier section.

Your very own documentation portfolio

The templates that we discussed earlier are just a basis that you can use to document your software. With time, you will eventually develop your own templates and style for making documentation. But always keep in mind the light but sufficient approach for project documentation: each document that's added should have a clearly defined target readership and should fill a real need. Documents that don't add a real value should not be written.

Each project is unique and has different documentation needs. For example, small terminal tools with simple usage can definitely live with only a single README file as its document landscape. Having such a minimal single-document approach is completely fine if the target readers are precisely defined and consistently grouped (system administrators, for instance).

Also, do not take the provided templates too rigorously. Some additional metadata provided as an example is really useful in either big projects or in strictly formalized teams. Tags, for instance, are intended to improve textual searches in big documentations, but will not provide any value in a documentation landscape consisting only of a few documents.

Also, including a document author is not always a good idea. Such an approach may be especially questionable in open source projects. In such projects, you will want the community to also contribute to the documentation. In most cases, such documents are continuously updated whenever there is such a need by whoever makes the contribution. People tend to treat the document *author* as the document *owner*. This may discourage people to update the documentation if every document has its author always specified. Usually, the version control software provides clearer and more transparent information about real document authors than explicitly provided metadata annotations. The situations where explicit authors are really recommended are various design documents, especially in projects where the design process is strictly formalized. The best example of this is the series of PEP documents provided with the Python language enhancement proposals.

Building a documentation landscape

The document portfolio we built in the previous section provides a structure at the document level, but does not provide a way to group and organize it to build the documentation the readers will have. This is what Andreas Rüping calls a document landscape, referring to the mental map the readers use when they browse the documentation. He came up with the conclusion that the best way to organize documents is to build a logical tree.

In other words, the different kinds of documents composing the portfolio need to find a place to live within a tree of directories. This place must be obvious to the writers when they create the document and to the readers when they are looking for it.

A great helper in browsing documentation is the index pages at each level that can drive writers and readers.

Building a document landscape is done in the following two steps:

- Building a tree for the producers (the writers)
- Building a tree for the consumers (the readers), on top of the producers' tree

This distinction between producers and consumers is important since they access the documents in different places and different formats.

Producer's layout

From a producer's point of view, each document is processed exactly like a Python module. It should be stored in the version control system and work like code. Writers do not care about the final appearance of their prose and where it is available; they just want to make sure that they are writing a document that is the single source of truth on the topic covered. reStructuredText files stored in a folder tree are available in the version control system, together with the software code, and are a convenient solution to build the documentation landscape for producers.

By convention, the `docs` folder is used as a root of documentation tree, as follows:

```
$ cd my-project
$ find docs
docs
docs/source
docs/source/design
docs/source/operations
docs/source/usage
docs/source/usage/cookbook
docs/source/usage/modules
docs/source/usage/tutorial
```

Notice that the tree is located in a `source` folder because the `docs` folder will be used as a root folder to set up a special tool in the next section.

From there, an `index.txt` file can be added at each level (besides the root), explaining what kind of documents the folder contains, or summarizing what each subfolder contains. These index files can define a listing of the documents they contain. For instance, the `operations` folder can contain a list of operations documents that are available, as follows:

```
==========
Operations
==========


This section contains operations documents:

- How to install and run the project
- How to install and manage a database for the project
```

It is important to know that people tend to forget to update such lists of documents and tables of contents. So, it is better to have them updated automatically.

In the next section, we will discuss one tool that, among many other features, can also handle this use case.

Consumer's layout

From a consumer's point of view, it is important to work out the index files and to present the whole documentation in a format that is easy to read and looks good. Web pages are the best pick and are easy to generate from reStructuredText files.

Summary

In this chapter, we have discussed an approach that should make documentation management an organized, lightweight, efficient, and (hopefully) fun process. We started from the seven rules of technical writing that apply to any kind of technical writing—not only documentation. From there, we have introduced tools and techniques that convert these simple rules into a clear and organized engineering process.

One of the hardest things to do when documenting a project is to keep it accurate and up-to-date. The only way to make this happen is by treating it is as a first-class engineering artefact. Good documentation is always close to the code it documents. Making the documentation part of the code repository makes this a lot easier. From there, every time a developer makes any change in the code, he or she should change the corresponding documentation as well.

A complementary approach to make sure the documentation is always accurate is to combine the documentation with tests through doctests. This is covered in the next chapter, which presents test-driven development principles, and then document-driven development.

12
Test-Driven Development

Test-driven development (**TDD**) is a methodology that aims to produce high-quality software by concentrating on automated tests. It is widely used in the Python community, but it is also very popular in other communities.

Testing is especially important in Python due to its dynamic nature. Python lacks static typing, so many, even minute, errors won't be noticed until the code is run and each of its lines is executed. But the problem is not only how types in Python work. The most troublesome bugs are not related to bad syntax or wrong type usage, but rather to logical errors and subtle mistakes that may lead to major failures.

In order to efficiently test your software, you often need to use a wide variety of testing strategies that will be executed on different abstraction levels of your application. Unfortunately, not every testing strategy is suitable for every project, and not every testing framework will be suitable for every kind of testing strategy. This is why it's not uncommon for larger projects to use multiple testing libraries and frameworks that fulfill different (often overlapping) testing needs. So, in order to better guide you through the sophisticated world of software testing and test-driven development, this chapter is split into the following two parts:

- *I don't test*, which advocates TDD and quickly describes how to do it with the standard library
- *I do test*, which is intended for developers who practice tests and wish to get more out of them

Technical requirements

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

- `pytest`
- `nose`
- `coverage`
- `tox`

You can install these packages using the following command:

```
python3 -m pip install <package-name>
```

The code files for this chapter can be found at `https://github.com/PacktPublishing/Expert-Python-Programming-Third-Edition/tree/master/chapter12`.

I don't test

If you are already convinced by TDD, you should move on to the next section. It will focus on advanced techniques and tools for making your life easier when working with tests. This part is mainly intended for those who are not using this approach, and tries to advocate its usage.

Three simple steps of test-driven development

The test-driven development process, in its simplest form, consists of the following three steps:

1. Writing automated tests for a new functionality or improvement that was not implemented yet.
2. Providing minimal code that just passes all the defined tests.
3. Refactoring code to meet the desired quality standards.

The most important fact to remember about this development cycle is that tests should be written before actual implementation. It is not an easy task for inexperienced developers, but it is the only approach that reliably guarantees that the code you are going to write will be testable.

For example, a developer who is asked to write a function that checks whether the given number is a prime number may write a few examples on how to use it, and the expected results, as follows:

```
assert is_prime(5)
assert is_prime(7)
assert not is_prime(8)
```

The developer that implements this feature does not need to be the only one responsible for providing tests. The examples can be provided by another person as well. For instance, often, official specifications of network protocols or cryptography algorithms provide test vectors that are intended to verify the correctness of the implementation. These are a perfect basis for test cases of code that aims to implement such protocols and algorithms.

From there, the function can be iteratively implemented until the preceding example works, as follows:

```
def is_prime(number):
    for element in range(2, number):
        if number % element == 0:
            return False
    return True
```

It is possible that upon usage, code users will find bugs or unexpected results. Such special cases are new examples of usage that the implemented function should be able to deal with, as follows:

```
>>> assert not is_prime(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

As new problematic usage examples are discovered, the function is gradually improved, as follows:

```
def is_prime(number):
    if number in (0, 1):
        return False

    for element in range(2, number):
        if number % element == 0:
            return False

    return True
```

This process may be repeated multiple times, as it is often hard to predict all meaningful and problematic usage examples, for example:

```
>>> assert not is_prime(-3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

The point of this approach is to find an implementation through a series of gradual improvements. This process guarantees that the function properly handles all possible edge cases within the defined usage constraints, as follows:

```
def is_prime(number):
    if number < 0 or number in (0, 1):
        return False

    for element in range(2, number):
        if number % element == 0:
            return False

    return True
```

And the collection of discovered or planned usage examples serves as a definition of such usage constraints. Common usage examples become a test for an implemented function that verifies that the implementation meets all known requirements. In practice, common usage examples are gathered in their own names function so, they can be executed every time the code evolves, as follows:

```
def test_is_prime():
    assert is_prime(5)
    assert is_prime(7)

    assert not is_prime(8)
    assert not is_prime(0)
    assert not is_prime(1)

    assert not is_prime(-1)
    assert not is_prime(-3)
    assert not is_prime(-6)
```

In the previous example, every time we come up with a new requirement, the `test_is_prime()` function should be updated first to define the expected behavior of the `is_prime()` function. Then, a test is run to check if the implementation delivers the desired results. Only if the tests are known to be failing is there a need to update the code for the tested function.

Test-driven development provides a lot of benefits, including the following:

- It helps to prevent software regression
- It improves software quality
- It provides a kind of low-level documentation of code behavior
- It allows you to produce robust code faster and to work in shorter development cycles

The best convention to deal with multiple tests in Python is to gather all of them in a single module or package (usually named `tests`) and have an easy way to run their whole suite using a single shell command. Fortunately, there is no need to build whole test toolchains all by yourself. Both the Python standard library and Python Package Index come with plenty of test frameworks and utilities that allow you to build, discover, and run tests in a convenient way. We will discuss the most notable examples of such packages and modules later in this chapter.

Let's discuss the benefits of test-driven development in the following sections.

Preventing software regression

We all face software regression issues in our developer lives. Software regression is a new bug introduced by a change. It manifests when features or functionalities that were known to be working in the previous versions of the software get broken and stop working at some point of project development.

The main reason for regressions is high complexity of the software. At some point, it is impossible to guess what a single change in the code base might lead to. Changing some code might break some other features, and sometimes lead to vicious side effects, such as silently corrupting data. And high complexity is not only a problem of huge code bases. There is, of course, obvious correlation between the amount of code and its complexity, but even small projects (few hundreds/thousands lines of code) may have such convoluted architecture that it is hard to predict all possible consequences of a relatively small changes.

To avoid regression, the whole set of features the software provides should be tested every time a change occurs. Without this, you are not able to reliably tell the difference between bugs that always existed in your software and the new ones that were introduced with new code changes.

Opening a code base to several developers amplifies the problem, since each person may not be fully aware of all the development activities. While having a version control system prevents unexpected conflicts, it does not prevent all unwanted interactions.

TDD helps reduce software regression. The whole software can be automatically tested after each change. This will work as long as each feature has the proper set of tests. When TDD is done properly, the collection of tests grows together with the main code base.

Since, execution of a full test campaign may take quite a long time, it is good practice to delegate it to some continuous integration system, which can do the work in the background. We discussed such solutions already in `Chapter 10`, *Managing Code*. Nevertheless, every developer should be able to launch their tests manually, at least for the concerned modules. Relying solely on continuous integration will have a negative impact on the developers' productivity. Programmers should be able to run selections of tests easily in their environments. This is the reason why you should carefully choose the testing tools for the project. You should prefer a test frameworks that allows you to easily select and group tests for execution.

Improving code quality

When writing code, we often focus on algorithms, data structures, and performance, but lose the code user's point of view—how and when will our function, class, or module be used? Are the arguments easy and logical to use? Are the names in this new API right? Will it be easy to extend the code in future?

You can ensure such qualities by applying the tips described in the previous chapters, such as in `Chapter 6`, *Choosing Good Names*. But the only way to do this efficiently is to write usage examples. This is the moment when you'll realize if the code you wrote is logical and easy to use. Often, the first refactoring occurs right after the module, class, or function is finished.

Writing tests, which are use cases for the code, helps in maintaining the user's point of view. By starting from usage examples defined in tests, you will often produce better code. It is difficult to test gigantic functions and huge monolithic classes. Code that is written with testing in mind tends needs to be architected more cleanly and modularly.

Providing the best developer documentation

Tests are the best place for a developer to learn how software works. They are the use cases the code was primarily created for. Reading them provides a quick and deep insight into how the code works. Sometimes, an example is worth more than a thousand words.

The fact that these tests are always up to date with the code base makes them the best developer documentation that a piece of software can have. Tests don't go stale in the same way textual documentation does, otherwise they would fail.

Producing robust code faster

Writing without testing leads to long debugging sessions. A consequence of a bug in one module might manifest itself in a completely different part of the software. Since you don't know who to blame, you spend an inordinate amount of time debugging. It's better to fight small bugs one at a time when a test fails, because you'll have a better clue as to where the real problem is. And testing is often more fun than debugging because it is still a kind of coding.

If you measure the time taken to fix the code together with the time taken to write it, it will usually be longer than the time a TDD approach would take. This is not obvious when you start a new piece of code. This is because the time taken to set up a test environment and write the first few tests is extremely long compared to the time taken just to write the first pieces of code.

What kind of tests?

There are several kinds of tests that can be made on any software. The main ones are **unit tests, acceptance tests,** and **functional tests**. These are the ones that most people think of when discussing the topic of software testing. But there are a few other kinds of tests that you can use in your project, such as **integration tests**, **load and performance testing**, and **code quality testing**. We will discuss some of them shortly in the following sections.

Unit tests

Unit tests are low-level tests that perfectly fit test-driven development. As the name says, they focus on testing software units. A software unit can be understood as the smallest testable piece of the application code. Depending on the application, the size may vary from whole modules to a single method or function, but usually unit tests are written for the smallest fragments of code possible. Unit tests usually isolate the tested unit (module, class, function, and so on) from the rest of the application and other units. When external dependencies are required, such as web APIs or databases, they are often replaced by fake objects or mocks.

Acceptance tests

An acceptance test focuses on a feature and deals with the software like a black box. It just makes sure that the software really does what it is supposed to do, using the same media as that of the users, and observes application output. These tests are sometimes written out of the ordinary development cycle to validate that the application meets defined requirements. They are usually run as a checklist over the software. Often, these tests are not done through TDD, and are built by managers, QA staff, or even customers. In that case, they are often called **user acceptance tests**.

Still, they can and they should be done with TDD principles in mind. Acceptance tests can be provided before the features are written. Developers get a pile of acceptance tests, usually made out of the functional specifications, and their job is to make sure the code will pass all of them.

The tools that are used to write these tests depend on the user interface the software provides. Some popular tools used by Python developers are shown in the following table:

Application type	Tool
Web application	Selenium (for Web UI with JavaScript)
Web application	`zope.testbrowser` (doesn't test JS)
WSGI application	`paste.test.fixture` (doesn't test JS)
Gnome desktop application	dogtail
Win32 desktop application	`pywinauto`

> For an extensive list of functional testing tools,
> see the *PythonTestingToolsTaxonomy* page at the Python
> Wiki: `https://wiki.python.org/moin/PythonTestingToolsTaxonomy`.

Functional tests

Functional tests focus on whole features and functionalities instead of small code units. They are similar in their purpose to acceptance tests. The main difference is that functional tests do not necessarily need to use the same interface as the user. For instance, when testing web applications, some of the user interactions (or its consequences) can be simulated by synthetic HTTP requests or direct database access, instead of simulating real page loading and mouse clicks.

This approach is often easier and faster than testing with tools that are used in *user acceptance tests*. The downside of limited functional tests is that they tend to not cover enough parts of the application where different abstraction layers and components meet. Tests that focus on such *meeting points* are often called *integration tests*.

Integration tests

Integration tests represent a higher level of testing than unit tests. They test bigger parts of the code and focus on situations where many application layers or components meet and interact with each other. The form and scope of integration tests varies depending on the project's architecture and complexity. For example, in small and monolithic projects, this may be as simple as running more complex functional tests and allowing them to interact with real backing services (databases, caches, and so on), instead of mocking or faking them. For complex scenarios or products that are built from multiple services, the real integration tests may be very extensive and even require running the whole project in a big distributed environment that mirrors the production environment.

Integration tests are often very similar to functional tests, and the border between them is very blurry. It is very common that integration tests are also logically testing separate functionalities and features.

Load and performance testing

Load tests and performance tests provide objective information about code efficiency rather than its correctness. The terms load testing and performance testing are used by some interchangeably, but the first one refers to a limited aspect of performance. Load testing focuses on measuring how code behaves under some artificial demand (load). This is a very popular way of testing web applications, where load is understood as web traffic from real users or programmatic clients. It is important to note that load tests in web applications tend to cover HTTP transactions. It makes them very similar in behavior to integration and functional tests. So, it is very important to make sure that the tested application components are fully verified to be working correctly before attempting load testing. Performance tests are generally those tests that aim to measure code performance and can target even small units of code. So, load tests are only a specific subtype of performance tests.

Both load and performance tests are special kinds of tests because they do not provide binary results (failure/success), but only some performance quality measurement. This means that single results need to be interpreted and/or compared with results of different test runs. In some cases, the project requirements may define some hard constraint on load or performance on the code, but this does not change the fact that there is always some arbitrary interpretation involved in these kinds of testing approaches.

Load and performance tests are great tools during the development of any software that needs to fulfill some **Service Level Agreements** (for example, uptime percentage, number of simultaneous connections), because it helps to reduce the risk of compromising the performance of critical code paths.

Code quality testing

There is no arbitrary scale that would say definitely if code 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 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. The good and popular approach is to test at least basic metrics (static code analysis and code style violations) and not allow merging of any code to the main stream that makes these metrics lower.

In, next section, we will discuss some basic testing tools from the Python standard library that allow you to implement many different types of software tests.

Python standard test tools

Python provides the following two simple modules in the standard library that allow you to write automated tests:

- `unittest` (`https://docs.python.org/3/library/unittest.html`): This is the standard and most common Python unit testing framework based on Java's JUnit and originally written by Steve Purcell (formerly `PyUnit`)

- doctest (`https://docs.python.org/3/library/doctest.html`): This is a literate programming testing tool with interactive usage examples

Let's take a look at these two modules in the following sections.

unittest

`unittest` basically provides what JUnit does for Java. It offers a base class called `TestCase`, which has an extensive set of methods to verify the output of function calls and statements. It is the most basic and common Python testing library and often serves as a basis for more complex and elaborate testing frameworks.

This module was created with unit tests in mind, but you can use it to write other kinds of tests. It is even possible to use it in acceptance testing flows with user interface layer integration, as some testing libraries provide helpers to drive tools such as Selenium on top of `unittest`.

Writing a simple unit test for a module using `unittest` is done by subclassing `TestCase` and writing methods with the `test` prefix. The example test module based on usage examples from the *Test-driven development principles* section will look like this:

```python
import unittest

from primes import is_prime


class MyTests(unittest.TestCase):
    def test_is_prime(self):
        self.assertTrue(is_prime(5))
        self.assertTrue(is_prime(7))

        self.assertFalse(is_prime(8))
        self.assertFalse(is_prime(0))
        self.assertFalse(is_prime(1))

        self.assertFalse(is_prime(-1))
        self.assertFalse(is_prime(-3))
        self.assertFalse(is_prime(-6))


if __name__ == "__main__":
    unittest.main()
```

The `unittest.main()` function is the utility that allows you to make the whole module executable as a test suite, as follows:

```
$ python test_is_prime.py -v
test_is_prime (__main__.MyTests) ... ok

----------------------------------------------------------
Ran 1 test in 0.000s

OK
```

The `unittest.main()` function scans the context of the current module and looks for classes that subclass `TestCase`. It instantiates them, then runs all methods that start with the `test` prefix.

A good test suite follows common and consistent naming conventions. For instance, if the `is_prime` function is included in the `primes.py` module, the test class could be called `PrimesTests` and put into the `test_primes.py` file, as follows:

```
import unittest

from primes import is_prime


class PrimesTests(unittest.TestCase):
    def test_is_prime(self):
        ...


if __name__ == '__main__':
unittest.main()
```

From there, every time the `primes` module evolves, the `test_primes` module gets more tests.

In order to work, the `test_primes` module needs to have the `primes` module available in the context. This can be achieved either by having both modules in the same package or by adding a tested module explicitly to the Python path. In practice, the `develop` command of `setuptools` is very helpful here (see Chapter 7, *Writing a Package*).

Running tests over the whole application presupposes that you have a script that builds a **test campaign** out of all test modules. `unittest` provides a `TestSuite` class that can aggregate tests and run them as a test campaign, as long as they are all instances of `TestCase` or `TestSuite`.

In Python's past, there was a convention that the test module provides a `test_suite` function that returns a `TestSuite`. This function would be used in the `__main__` section, when the module is called in command prompt, or automatically collected by a test runner, as follows:

```python
import unittest

from primes import is_prime


class PrimesTests(unittest.TestCase):
    def test_is_prime(self):
        self.assertTrue(is_prime(5))

        self.assertTrue(is_prime(7))

        self.assertFalse(is_prime(8))
        self.assertFalse(is_prime(0))
        self.assertFalse(is_prime(1))

        self.assertFalse(is_prime(-1))
        self.assertFalse(is_prime(-3))
        self.assertFalse(is_prime(-6))


class OtherTests(unittest.TestCase):
    def test_true(self):
        self.assertTrue(True)



def test_suite():
    """builds the test suite."""
    suite = unittest.TestSuite()
    suite.addTests(unittest.makeSuite(PrimesTests))
    suite.addTests(unittest.makeSuite(OtherTests))

    return suite


if __name__ == '__main__':
    unittest.main(defaultTest='test_suite')
```

Running this module from the shell will print the following test campaign output:

```
$ python test_primes.py -v
test_is_prime (__main__.PrimesTests) ... ok
test_true (__main__.OtherTests) ... ok


----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

The preceding approach was required in the older versions of Python when the `unittest` module did not have proper test discovery utilities. Usually, the running of all tests was done by a global script that browsed the code tree, looking for tests to run. This process is often called **test discovery**, and will be covered more extensively later in this chapter. For now, you should only know that `unittest` provides a simple command that can discover all tests from modules and packages that are named with a `test` prefix, as follows:

```
$ python -m unittest -v
test_is_prime (test_primes.PrimesTests) ... ok
test_true (test_primes.OtherTests) ... ok


----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

If you use the preceding command, then there is no need to manually define the __main__ sections and invoke the `unittest.main()` function.

doctest

`doctest` is a module that extracts test snippets in the form of interactive prompt sessions from docstrings or text files, and replays them to check whether the example output is the same as the real one.

For instance, the text file with the following content could be run as a test:

```
Check addition of integers works as expected::

>>> 1 + 1
    2
```

Let's assume that this documentation file is stored in the filesystem under the `test.rst` name. The `doctest` module provides some functions to extract and run the tests from such documentation files, as follows:

```
>>> import doctest
>>> doctest.testfile('test.rst', verbose=True)
Trying:
    1 + 1
Expecting:
    2
ok
1 items passed all tests:
   1 tests in test.rst
1 tests in 1 items.
1 passed and 0 failed.
Test passed.
TestResults(failed=0, attempted=1)
```

Using `doctest` has the following advantages:

- Packages can be documented and tested through examples
- Documentation examples are always up to date
- Using examples in the form of doctests helps to maintain the user's point of view

However, doctests do not make unit tests obsolete; they should be used only to provide human-readable examples in documents or docstrings. In other words, when the tests are concerning low-level matters or need complex test fixtures that would obfuscate the document, they should not be used.

Some Python frameworks such as Zope are using doctests extensively, and they are, at times, criticized by people who are new to the code. Some doctests are really hard to read and understand, since the examples break one of the rules of technical writing—they cannot be taken and run in a simple prompt, and they often need extensive knowledge. So, documents that are supposed to help newcomers are really hard to read if the code examples are based on complex test fixtures or even specific test APIs.

> When you use doctests that are part of the documentation of your packages, be careful to follow the seven rules of technical writing that were explained in `Chapter 11`, *Documenting Your Project.*

I do test

The *I don't test* section should have familiarized you with the basics of test-driven development, but there are some more things you should learn before you will be able to efficiently use this methodology.

This section describes a few problems developers bump into when they write tests, and some ways to solve them. It also provides a quick review of the popular test runners and tools that are available in the Python community.

unittest pitfalls

The `unittest` module was introduced in Python 2.1 and has been massively used by developers since then. But some alternative test frameworks were created in the community by people who were frustrated by the weaknesses and limitations of `unittest`.

The following are the common criticisms that are often made:

- You have to prefix the method names with `test`.
- You are encouraged to use assertion methods provided in `TestCase` instead of plain `assert` statements as existing methods may not cover every use case.
- The framework is hard to extend because it requires massive subclassing of classes or tricks such as decorators.
- Test fixtures are sometimes hard to organize because the `setUp` and `tearDown` facilities are tied to the `TestCase` level, though they run once per test. In other words, if a test fixture concerns many test modules, it is not simple to organize its creation and cleanup.
- It is not convenient to run a test campaign. The default test runner (`python -m unittest`) indeed provides some test discovery but does not provide enough filtering capabilities. In practice, extra scripts have to be written to collect the tests, aggregate them, and then run them in a convenient way.

A lighter approach is needed to write tests without suffering from the rigidity of a framework that looks too much like its big Java brother, JUnit. Since Python does not require working with a 100% class-based environment, it is preferable to provide a more Pythonic test framework that is not primarily based on subclassing.

A slightly better framework would include the following:

- Provide a simple way to mark any function or any class as a test
- Be extendable through a plugin system
- Provide a complete test fixture environment for all test levels: at whole campaign, at module level, and at single test level
- Provide a test runner based on test discovery with an extensive set of options

unittest alternatives

Some third-party tools try to solve the problems we mentioned previously by providing extra features in the shape of `unittest` extensions.

The Python Wiki provides a very long list of various testing utilities and frameworks (see `https://wiki.python.org/moin/PythonTestingToolsTaxonomy`), but there are just the following two projects that are especially popular:

- `nose`, documented at `http://nose.readthedocs.org`
- `py.test`, documented at `http://pytest.org`

nose

`nose` is mainly a test runner with powerful discovery features. It has extensive options that allow you to run all kinds of test campaigns in a Python application.

It is not a part of standard library, but is available on PyPI and can be easily installed with `pip`, as follows:

```
pip install nose
```

In the following sections, we will take a look at a default `nose` test runner, system of fixtures, and integration with setup tools and the plugin system in `nose`.

Test runner

After installing `nose`, a new command called `nosetests` is available at the prompt. Running the tests presented in the first section of this chapter can be done directly with it, as follows:

```
$ nosetests -v
test_true (test_primes.OtherTests) ... ok
test_is_prime (test_primes.PrimesTests) ... ok
builds the test suite. ... ok


----------------------------------------------------------
Ran 3 tests in 0.009s

OK
```

`nose` takes care of discovering the tests by recursively browsing the current working directory, and building a test suite on its own. The preceding example at first glance does not look like any improvement over the simple `python -m unittest` command. The real difference becomes noticeable when you execute `nosetests` with the `--help` switch. You will see that `nose` provides tens of parameters that allow you to finely control test discovery and execution.

Writing tests

`nose` goes a step further when executing tests compared to `unittest` by running all classes and functions whose name matches the regular expression `((?:^|[b_.-])[Tt]est)`. It searches for test units in modules whose names match the same expression. Roughly, all callables that start with `test` and are located in a module that match that pattern will also be executed as a test.

For instance, this `test_ok.py` module will be recognized and run by `nose`:

```
$ cat test_ok.py
def test_ok():
    print('my test')

$ nosetests -v
test_ok.test_ok ... ok


----------------------------------------------------
Ran 1 test in 0.071s

OK
```

Regular `TestCase` classes and `doctests` are executed as well.

Last but not least, `nose` provides assertion functions that are similar to the ones provided in the `unittest.TestCase` class methods. But these are provided as functions, with names that follow the PEP 8 naming conventions, rather than following the Java naming convention that `unittest` uses.

Writing test fixtures

`nose` supports the following three levels of fixtures:

- **Package level**: The `setup` and `teardown` functions can be added in the `__init__.py` module of a test's package containing all test modules
- **Module level**: A test module can have its own `setup` and `teardown` functions
- **Test level**: The callable can also have fixture functions using the `@with_setup()` decorator provided

For instance, to set a test fixture at the module and test level, use this code:

```
def setup():
    # setup code, launched for the whole module
    ...

def teardown():
    # teardown code, launched for the whole module
    ...

def set_ok():
    # setup code used on demand using with_setup decorator
    ...

@with_setup(set_ok)
def test_ok():
    print('my test')
```

Integration with setuptools and plugin system

The `nose` framework integrates smoothly with `setuptools` and so you can use the `setup.py test` command to invoke all the tests. Such integration is extremely useful in ecosystems of packages that require a common test entry point but may use different testing frameworks.

This integration is done by adding the `test_suite` metadata in the `setup.py` script, as follows:

```
setup(
    ...
    test_suite='nose.collector',
)
```

`nose` also uses `setuptools` entry point machinery for developers to write `nose` plugins. This allows you to override or modify the main aspects of the tool, such as the test discovery algorithm or output formatting.

Wrap-up

`nose` is a complete testing tool that fixes many of the issues `unittest` has. It is still designed to use implicit prefix names for tests, which remains a constraint for some developers. While this prefix can be customized, it still requires you to follow a convention.

This convention over configuration statement is not bad, and a lot better than the boilerplate code required in `unittest`. But using explicit decorators, for example, could be a nice way to get rid of the `test` prefix.

Also, the ability to extend `nose` with plugins makes it very flexible, and allows you to customize this tool to meet your own needs.

If your testing workflow requires overriding a lot of `nose` parameters, you can easily add a `.noserc` or a `nose.cfg` file in your home directory or project root. It will specify the default set of options for the `nosetests` command. For instance, a good practice is to automatically look for `doctests` during the test run. An example of the `nose` configuration file that enables running doctests is as follows:

```
[nosetests] with-doctest=1 doctest-extension=.txt
```

py.test

`py.test` is very similar to `nose`. In fact, the latter was inspired by `py.test`, so we will focus mainly on details that make these tools different from each other. The tool was born as part of a larger package called `py`, but now these are developed separately.

Like every third-party package mentioned in this book, `py.test` is available on PyPI and can be installed with `pip` as `pytest`, as follows:

```
$ pip install pytest
```

From there, a new `py.test` command is available in your shell that can be used exactly like `nosetests`. The tool uses similar pattern matching and test discovery algorithms to find tests in your project. The pattern is slightly stricter and will only match the following:

- Classes that start with `Test`, in a file that starts with `test`
- Functions that start with `test`, in a file that starts with `test`

> Be careful to use the right character case. If a function starts with a capital `T`, it will be taken as a class, and thus ignored. And if a class starts with a lowercase `t`, `py.test` will break because it will try to deal with it like a function.

The following are the advantages of `py.test` over other frameworks:

- The ability to easily disable selected test classes
- A flexible and original mechanism for dealing with fixtures
- The built-in ability to distribute tests among several computers

In the following sections, we will take a look at writing test fixtures, disabling test functions and classes, and automated distributed tests in `py.test`.

Writing test fixtures

`py.test` supports two mechanisms to deal with fixtures. The first one, modeled after the xUnit framework, is similar to `nose`. Of course, the semantics differ a bit. `py.test` will look for three levels of fixtures in each test module, as shown in following example, taken from the official documentation:

```
def setup_module(module):
    """ Setup up any state specific to the execution
        of the given module.
    """

def teardown_module(module):
    """ Teardown any state that was previously setup
        with a setup_module method.
    """

def setup_class(cls):
```

```
    """ Setup up any state specific to the execution
        of the given class (which usually contains tests).
    """

def teardown_class(cls):
    """ Teardown any state that was previously setup
        with a call to setup_class.
    """

def setup_method(self, method):
    """ Setup up any state tied to the execution of the given
        method in a class. setup_method is invoked for every
        test method of a class.
    """

def teardown_method(self, method):
    """ Teardown any state that was previously setup
        with a setup_method call.
    """
```

Each function will get the receive module, class, or method as an argument. The test fixture will, therefore, be able to work on the context without having to look for it, as it does with `nose`.

The alternative mechanism for writing fixtures with `py.test` is built on the concept of dependency injection and allows you to maintain the test state in a more modular and scalable way. The non xUnit-style fixtures (setup/teardown procedures) always have unique names and need to be explicitly activated by declaring their use in test functions, methods, and modules.

The simplest implementation of fixtures takes the form of a named function declared with the `pytest.fixture()` decorator. To mark a fixture as used in test, it needs to be declared as a function or method argument. To make it more clear, consider the previous example of the test module for the `is_prime` function rewritten as follows with the use of `py.test` fixtures:

```python
import pytest

from primes import is_prime


@pytest.fixture()
def prime_numbers():
    return [3, 5, 7]
```

```
@pytest.fixture()
def non_prime_numbers():
    return [8, 0, 1]


@pytest.fixture()
def negative_numbers():
    return [-1, -3, -6]


def test_is_prime_true(prime_numbers):
    for number in prime_numbers:
        assert is_prime(number)


def test_is_prime_false(non_prime_numbers, negative_numbers):
    for number in non_prime_numbers:
        assert not is_prime(number)

    for number in negative_numbers:
        assert not is_prime(number)
```

Disabling test functions and classes

`py.test` provides a simple mechanism to disable some tests upon certain conditions. This is called *test skipping,* and the `pytest` package provides the `@mark.skipif` decorator for that purpose. If a single test function or a whole test class decorator needs to be skipped upon certain conditions, you need to define it with this decorator and some expression that verifies if the expected condition was met. Here is an example from the official documentation that skips running the whole test case class if the test suite is executed on Windows:

```
import pytest

@pytest.mark.skipif(
    sys.platform == 'win32',
    reason="does not run on windows"
)
class TestPosixCalls:
    def test_function(self):
        "will not be setup or run under 'win32' platform"
```

You can, of course, predefine the skipping condition in order to share them across your testing modules, as follows:

```
import pytest

skipwindows = pytest.mark.skipif(
    sys.platform == 'win32',
    reason="does not run on windows"
)

@skip_windows
class TestPosixCalls:
    def test_function(self):
        "will not be setup or run under 'win32' platform"
```

If test is marked in such a way, it will not be executed at all. However, in some cases, you want to run an execute specific test that is expected to fail under known conditions. For this purpose, a different decorator is provided. This is called `@mark.xfail`, and it ensures that the test is always run, but it should fail at some point if the predefined condition occurs, as follows:

```
import pytest

@pytest.mark.xfail(
sys.platform == 'win32',
    reason="does not run on windows"
)
class TestPosixCalls:

    deftest_function(self):
        "it must fail under windows"
```

Using `xfail` is much stricter than `skipif`. The test is always executed, and if it does not fail when it is expected to do so, then the whole execution of the test suite will be marked as a failure.

Automated distributed tests

An interesting feature of `py.test` is its ability to distribute the tests across several computers. As long as the computers are reachable through SSH, `py.test` will be able to drive each computer by sending tests to be performed.

However, this feature relies on the network; if the connection is broken, the slave will not be able to continue working since it is fully driven by the master.

A Buildbot or other continuous integration tools are preferable when a project has large and long running test campaigns. But the `py.test` distributed model can be used for the ad hoc distribution of tests when you don't have a proper continuous integration pipeline at hand.

Wrap-up

`py.test` is very similar to `nose` since no boilerplate code is needed to aggregate the tests in it. It also has a good plugin system, and there are a great number of extensions available on PyPI.

`py.test` focuses on making the tests run fast, and is truly superior compared to the other tools in this area. The other notable feature is the original approach to fixtures that really helps in managing a reusable library of fixtures. Some people may argue that there is too much magic involved, but it really streamlines the development of test suites. This single advantage of `py.test` makes it my tool of choice, so I really recommend it.

In the next section, you will learn how to measure how much of your code is actually covered by tests.

Testing coverage

Code coverage is a very useful metric that provides objective information on how well project code is tested. It is simply a measurement of how many and which lines of code are executed during the test execution. 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 is called the `coverage`, and is freely available on PyPI. Its usage is very simple and consists only of two steps. 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, as follows:

```
$ coverage run --source . `which py.test` -v
==================== test session starts ======================
platformdarwin -- Python 3.5.1, pytest-2.8.7, py-1.4.31, pluggy-0.3.1 --
/Users/swistakm/.envs/book/bin/python3 cachedir: .cache rootdir:
/Users/swistakm/dev/book/chapter10/pytest, inifile:  plugins:
capturelog-0.7, codecheckers-0.2, cov-2.2.1, timeout-1.0.0 collected 6
items  primes.py::pyflakes PASSED
```

```
primes.py::pep8 PASSED
test_primes.py::pyflakes PASSED
test_primes.py::pep8 PASSED
test_primes.py::test_is_prime_true PASSED
test_primes.py::test_is_prime_false PASSED
========= 6 passed, 1 pytest-warnings in 0.10 seconds ==========
```

The coverage run also accepts an –m parameter that specifies a runnable module name instead of a program path that may be better for some testing frameworks, as follows:

```
$ coverage run –m unittest
$ coverage run –m nose
$ coverage run –m pytest
```

The next step is to generate a human-readable report of your code coverage from results cashed in the .coverage file. The coverage package supports a few output formats, and the simplest one just prints an ASCII table in your Terminal, as follows:

```
$ coverage report
Name             StmtsMiss  Cover
---------------------------------
primes.py            7     0    100%
test_primes.py      16     0    100%
---------------------------------
TOTAL               23     0    100%
```

The other useful coverage report format is HTML that can be displayed as follows in your web browser:

```
$ coverage html
```

The default output folder of this HTML report is htmlcov/ in your working directory. The real advantage of the coverage html output is that you can browse annotated sources of your project with highlighted parts that have missing test coverage (as shown in the following diagram):
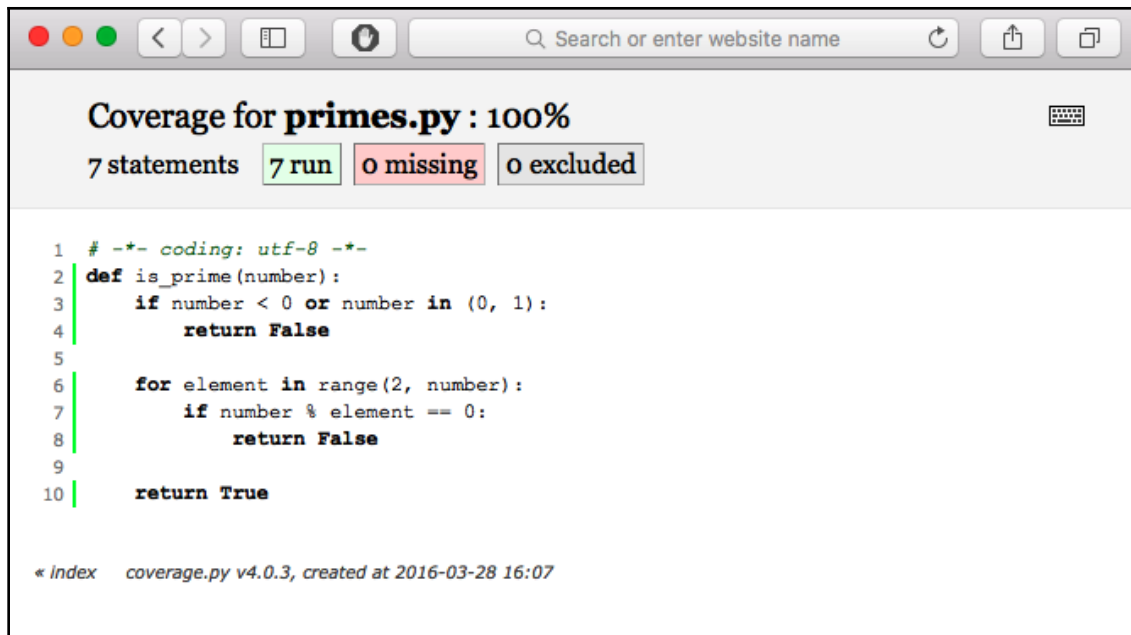
Figure 1 Example of annotated sources in coverage HTML report

You should know that while you should always strive to ensure 100% test coverage, it is never a guarantee that code is tested perfectly and there is no place where it can break. This means that every line of code was reached during execution but not necessarily every possible condition was tested. In practice, it may be relatively easy to ensure full code coverage, but it is really hard to make sure that every branch of code was reached. This is especially true for the testing of functions that may have multiple combinations of `if` statements and specific language constructs like `list/dict/set` comprehensions. You should always care about good test coverage, but should never treat its measurement as the final answer of how good your testing suite is.

In the following sections, we will take a look at other testing techniques and tools that allow you to replace existing third-party dependencies during tests with objects that imitate their behaviors.

Fakes and mocks

Writing unit tests presupposes that you 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 the tests include the following:

- They are concerning with an atomic part of the application, which can be a function, method, class, or interface
- They provide deterministic, reproducible results

Sometimes, the proper isolation of the program component is not obvious. For instance, if the code sends emails, it will probably call Python's smtplib module, which will work with the SMTP server through a network connection. If we want our tests to be reproducible and are just testing if emails have the desired content, then probably no real network connection needs to happen. Ideally, unit tests should run on any computer with no external dependencies and side effects.

Thanks to Python's dynamic nature, it is possible to use the **monkey patching** technique to modify the runtime code from the test fixture (that is, modify software dynamically at runtime without touching the source code), in order to **fake** the behavior of a third-party code or library. In the following sections, we will learn how to build such objects.

Building a fake

A fake behavior in the tests can be created by discovering the minimal set of interactions needed for the tested code to work with the external parts. Then, the output is manually returned, or uses a real pool of data that has been previously recorded.

You can start this by creating an empty class or function and use it as a replacement for a component that has to be substituted. You can then iteratively update your class definition until implementation of this fake object behaves as intended. This is possible thanks to the nature of a Python type system. The object is considered compatible with the given type, as long as it behaves as an expected type, and usually does not need to be related to that type via subclassing. This approach to typing in Python is called duck typing:

If something behaves like a duck it can be treated like a duck.

Let's take a look at the following example module named mailer with a function called `send` that sends emails using `smtplib` library:

```
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)

    server = smtplib.SMTP(server)
    try:
        return server.sendmail(sender, to, message.as_string())
    finally:
        server.quit()
```

> `py.test` will be used to demonstrate fakes and mocks in this section.

The corresponding test can be written as follows:

```
from mailer import send


def test_send():
    res = send(
        'john.doe@example.com',
        'john.doe@example.com',
        'topic',
        'body'
    )
    assert res == {}
```

This test will pass and work as long as there is an SMTP server on the local host. If not, it will fail, like so:

```
$ py.test --tb=short
======================== test session starts ========================
platform darwin -- Python 3.5.1, pytest-2.8.7, py-1.4.31, pluggy-0.3.1
rootdir: /Users/swistakm/dev/book/chapter10/mailer, inifile:
plugins: capturelog-0.7, codecheckers-0.2, cov-2.2.1, timeout-1.0.0
collected 5 items
mailer.py ..
test_mailer.py ..F
============================= FAILURES ==============================
_____ test_send _____
test_mailer.py:10: in test_send
    'body'
mailer.py:19: in send
    server = smtplib.SMTP(server)
.../smtplib.py:251: in __init__
    (code, msg) = self.connect(host, port)
.../smtplib.py:335: in connect
    self.sock = self._get_socket(host, port, self.timeout)
.../smtplib.py:306: in _get_socket
    self.source_address)
.../socket.py:711: in create_connection
    raise err
.../socket.py:702: in create_connection
    sock.connect(sa)
E   ConnectionRefusedError: [Errno 61] Connection refused
======== 1 failed, 4 passed, 1 pytest-warnings in 0.17 seconds ========
```

A patch can be added to fake the SMTP class, as follows:

```
import smtplib
import pytest
from mailer import send


class FakeSMTP(object):
    pass


@pytest.yield_fixture()
def patch_smtplib():
    # setup step: monkey patch smtplib
    old_smtp = smtplib.SMTP
    smtplib.SMTP = FakeSMTP

    yield
```

```
        # teardown step: bring back smtplib to
        # its former state
        smtplib.SMTP = old_smtp


    def test_send(patch_smtplib):
        res = send(
            'john.doe@example.com',
            'john.doe@example.com',
            'topic',
            'body'
        )
        assert res == {}
```

In the preceding code, we used a new `@pytest.yield_fixture()` decorator. It allows us to use a generator syntax to provide both setup and teardown procedures in a single fixture function. Now, our test suite can be run again with the patched version of `smtplib`, as follows:

```
$ py.test --tb=short -v
======================= test session starts ========================
platform darwin -- Python 3.5.1, pytest-2.8.7, py-1.4.31, pluggy-0.3.1 --
/Users/swistakm/.envs/book/bin/python3
cachedir: .cache
rootdir: /Users/swistakm/dev/book/chapter10/mailer, inifile:
plugins: capturelog-0.7, codecheckers-0.2, cov-2.2.1, timeout-1.0.0
collected 5 items

mailer.py::pyflakes PASSED
mailer.py::pep8 PASSED
test_mailer.py::pyflakes PASSED
test_mailer.py::pep8 PASSED
test_mailer.py::test_send FAILED

============================= FAILURES ==============================
_____ test_send _____
test_mailer.py:29: in test_send
    'body'
mailer.py:19: in send
    server = smtplib.SMTP(server)
E   TypeError: object() takes no parameters
======= 1 failed, 4 passed, 1 pytest-warnings in 0.09 seconds =======
```

As we see from the preceding transcript, our `FakeSMTP` class implementation is not complete yet. We need to update its interface to match the original SMTP class.

According to the duck typing principle, we only need to provide interfaces that are required by the tested `send()` function, as follows:

```
class FakeSMTP(object):
    def __init__(self, *args, **kw):
        # arguments are not important in our example
        pass

    def quit(self):
        pass

    def sendmail(self, *args, **kw):
        return {}
```

Of course, the fake class can evolve with new tests to provide more complex behaviors. But it should be as short and simple as possible. The same principle can be used with more complex outputs, by recording them to serve them back through the fake API. This is often done for third-party servers such as LDAP or SQL databases.

It is important to know that special care should be taken when monkey patching any built-in or third-party module. If not done properly, such an approach might leave unwanted side effects that will propagate between tests. Fortunately, many testing frameworks and libraries provide proper utilities that make patching of any code units safe and easy. In our example, we did everything manually and provided a custom `patch_smtplib()` fixture function with separated setup and teardown steps. A typical solution in `py.test` is much easier. This framework comes with a built-in `monkeypatch` fixture that should satisfy most of our patching needs, as follows:

```
import smtplib
from mailer import send


class FakeSMTP(object):
def __init__(self, *args, **kw):
        # arguments are not important in our example
        pass

    def quit(self):
        pass

    def sendmail(self, *args, **kw):
        return {}


def test_send(monkeypatch):
    monkeypatch.setattr(smtplib, 'SMTP', FakeSMTP)
```

```
    res = send(
        'john.doe@example.com',
        'john.doe@example.com',
        'topic',
        'body'
    )
    assert res == {}
```

You should keep in mind that *fakes* have limitations. If you decide to fake an external dependency, you might introduce bugs or unwanted behaviors that the real server wouldn't have, or the other way around.

Using mocks

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 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. It was created as a third-party package, but was soon included into the standard library as a provisional package (refer to `https://docs.python.org/dev/glossary.html#term-provisional-api`). For Python versions older than 3.3, you will need to install it from PyPI, as follows:

```
$ pip install Mock
```

In our following example, using `unittest.mock` to patch SMTP is way simpler than creating a fake from scratch:

```
import smtplib
from unittest.mock import MagicMock
from mailer import send


def test_send(monkeypatch):
    smtp_mock = MagicMock()
    smtp_mock.sendmail.return_value = {}

    monkeypatch.setattr(
        smtplib, 'SMTP', MagicMock(return_value=smtp_mock)
    )

    res = send(
        'john.doe@example.com',
```

```
        'john.doe@example.com',
        'topic',
        'body'
    )
    assert res == {}
```

The `return_value` argument of the mock object or method allows you to define what value will be returned by the call. When the mock object is used, every time an attribute is called by the code, it creates a new mock object for the attribute on the fly. Thus, no exception is raised. This is the case (for instance) for the `quit` method we wrote earlier in the *Building a fake* section. With mocks, we don't need to define such methods anymore.

In the preceding example, we have in fact created the following two mocks:

- The first one mocks the SMTP type object (class) and not its instance. This allows you to easily create a new object, regardless of the expected `__init__()` method. Mocks, by default, return new `Mock()` objects if treated as callable. This is why we needed to provide another mock as its `return_value` keyword argument to have control of the instance interface.
- The second mock is the actual instance that's returned on the patched `smtplib.SMTP()` call. In this mock, we control the behavior of the `sendmail()` method.

In the previous examples, we used the monkey patching fixture available from the `py.test` framework. But `unittest.mock` provides its own patching utilities. In some situations (like patching class objects), it may be simpler and faster to use them instead of your framework-specific tools. Here is an example of monkey patching with the `patch()` context manager provided by the `unittest.mock` module:

```
from unittest.mock import patch
from mailer import send


deftest_send():
    with patch('smtplib.SMTP') as mock:
        instance = mock.return_value
        instance.sendmail.return_value = {}
        res = send(
            'john.doe@example.com',
            'john.doe@example.com',
            'topic',
            'body'
        )
        assert res == {}
```

In the next section, we will discuss testing applications in multiple different environments and under different dependency versions.

Testing environment and dependency compatibility

The importance of environment isolation has already been mentioned in this book many times. By isolating your execution environment on both the application level (virtual environments) and system level (system virtualization), you are able to ensure that your tests run under repeatable conditions. This way, you protect yourself from rare and obscure problems caused by broken dependencies or system interoperability issues.

The best way to allow proper isolation of the test environment is to use good continuous integration systems that support system virtualization or containerization. There are good hosted continuous integration systems such as Travis CI (for Linux and macOS) or AppVeyor (for Windows) that offer such capabilities for open source projects for free. But if you need such a thing for testing proprietary software, it is very likely that you will have to either pay for such a service or host it on your own infrastructure with some existing open source CI tools (such as GitLab CI, Jenkins, or Buildbot).

Dependency matrix testing

Testing matrices for open source Python projects in most cases focuses only on different Python versions and rarely on different operating systems. Not doing your tests and builds on different systems is completely OK for simple projects that are purely Python, and there are no expected system interoperability issues. But some projects, especially distributed as compiled Python extensions, should be definitely tested on various target operating systems. For some open source projects, you may even be forced to use a few independent CI systems to provide builds for just the three most popular ones (Windows, Linux, and macOS). If you are looking for a good example, you can take a look at the small pyrilla project (refer to `https://github.com/swistakm/pyrilla`), which is a simple C audio extension for Python. It uses both Travis CI and AppVeyor in order to provide compiled builds for Windows, macOS, and a large range of CPython versions.

But dimensions of test matrices do not end on systems and Python versions. Packages that provide integration with other software, such as caches, databases, or system services, should be tested on various versions of integrated applications. A good tool that makes such testing easy is tox (refer to `http://tox.readthedocs.org`). It provides a simple way to configure multiple testing environments and run all tests with a single `tox` command. It is a very powerful and flexible tool, but is also very easy to use. The best way to present its usage is to show an example of a configuration file that is in fact the core of tox. Here is the `tox.ini` file from the `django-userena` project (refer to `https://github.com/bread-and-pepper/django-userena`):

```
[tox]
downloadcache = {toxworkdir}/cache/

envlist =
    ; py26 support was dropped in django1.7
    py26-django{15,16},
    ; py27 still has the widest django support
    py27-django{15,16,17,18,19},
    ; py32, py33 support was officially introduced in django1.5
    ; py32, py33 support was dropped in django1.9
    py32-django{15,16,17,18},
    py33-django{15,16,17,18},
    ; py34 support was officially introduced in django1.7
    py34-django{17,18,19}
    ; py35 support was officially introduced in django1.8
    py35-django{18,19}

[testenv]
usedevelop = True
deps =
    django{15,16}: south
    django{15,16}: django-guardian<1.4.0
    django15: django==1.5.12
    django16: django==1.6.11
    django17: django==1.7.11
    django18: django==1.8.7
    django19: django==1.9
    coverage: django==1.9
    coverage: coverage==4.0.3
    coverage: coveralls==1.1

basepython =
    py35: python3.5
    py34: python3.4
    py33: python3.3
    py32: python3.2
    py27: python2.7
```

```
    py26: python2.6

commands={envpython} userena/runtests/runtests.py userenaumessages
{posargs}

[testenv:coverage]
basepython = python2.7
passenv = TRAVIS TRAVIS_JOB_ID TRAVIS_BRANCH
commands=
    coverage run --source=userena userena/runtests/runtests.py
userenaumessages {posargs}
    coveralls
```

This configuration allows you to test `django-userena` on five different versions of Django and six versions of Python. Not every Django version will work on every Python version, and the `tox.ini` file makes it relatively easy to define such dependency constraints. In practice, the whole build matrix consists of 21 unique environments (including a special environment for code coverage collection). It would require tremendous effort to create each of such testing environment without a tool like tox.

Tox is great, but its usage gets more complicated if we want to change other elements of the testing environment that are not plain Python dependencies. This is a situation where we need to test under different versions of system packages and backing services. The best way to solve this problem is to again use good continuous integration systems that allow you to easily define matrices of environment variables and install system software on virtual machines. A good example of doing that using Travis CI is provided by the `ianitor` project (refer to `https://github.com/ClearcodeHQ/ianitor/`) that was already mentioned in `Chapter 11`, *Documenting Your Project*. It is a simple utility for the Consul discovery service. The Consul project has a very active community, and many new versions of its code are released every year. This makes it very reasonable to test against various versions of that service. Such an approach makes sure that the `ianitor` project is still up to date with the latest version of that software, but also does not break compatibility with previous Consul versions. Here is the content of the `.travis.yml` configuration file for Travis CI that allows you to test against three different Consul versions and four Python interpreter versions:

```
language: python

install: pip install tox --use-mirrors
env:
  matrix:
    # consul 0.4.1
    - TOX_ENV=py27    CONSUL_VERSION=0.4.1
    - TOX_ENV=py33    CONSUL_VERSION=0.4.1
    - TOX_ENV=py34    CONSUL_VERSION=0.4.1
```

```
        - TOX_ENV=py35      CONSUL_VERSION=0.4.1

        # consul 0.5.2
        - TOX_ENV=py27      CONSUL_VERSION=0.5.2
        - TOX_ENV=py33      CONSUL_VERSION=0.5.2
        - TOX_ENV=py34      CONSUL_VERSION=0.5.2
        - TOX_ENV=py35      CONSUL_VERSION=0.5.2

        # consul 0.6.4
        - TOX_ENV=py27      CONSUL_VERSION=0.6.4
        - TOX_ENV=py33      CONSUL_VERSION=0.6.4
        - TOX_ENV=py34      CONSUL_VERSION=0.6.4
        - TOX_ENV=py35      CONSUL_VERSION=0.6.4

        # coverage and style checks
        - TOX_ENV=pep8      CONSUL_VERSION=0.4.1
        - TOX_ENV=coverage CONSUL_VERSION=0.4.1

  before_script:
    - wget
https://releases.hashicorp.com/consul/${CONSUL_VERSION}/consul_${CONSUL_VER
SION}_linux_amd64.zip
    - unzip consul_${CONSUL_VERSION}_linux_amd64.zip
    - start-stop-daemon --start --background --exec `pwd`/consul -- agent -
server -data-dir /tmp/consul -bootstrap-expect=1

  script:
    - tox -e $TOX_ENV
```

The preceding example provides 14 unique test environments (including `pep8` and `coverage` builds) for `ianitor` code. This configuration also uses tox to create actual testing virtual environments on Travis VMs. It is actually a very popular approach to integrating tox with different CI systems. By moving as much of the test environment configuration as possible to tox, you are reducing the risk of locking yourself to a single CI vendor. Things like the installation of new services or defining system environment variables are supported by most of Travis CI's competitors, so it should be relatively easy to switch to a different service provider if there is a better product available on the market or Travis changes their pricing model for open source projects.

Document-driven development

doctests are nice things in Python that you can't find in many other programming languages. The fact that documentation can use code examples that are also runnable as tests changes the way TDD can be done, for instance, a part of the documentation can be done through `doctests` during the development cycle. This approach also ensures that the provided examples are always up to date and are really working.

Building software through doctests rather than regular unit tests can be a part of **document-driven development** (**DDD**). Developers explain what the code is doing in plain English, while they are implementing it.

Writing a story

Writing doctests in DDD is done by building a story about how a piece of code works and should be used. These principles are described in plain English and then a few code usage examples are distributed throughout the text. A good practice is to start to write text on how the code works, and then add some code examples.

To see an example of doctests in practice, let's look at the `atomisator` package (refer to `https://bitbucket.org/tarek/atomisator`). The documentation text for its `atomisator.parser` subpackage (under `packages/atomisator.parser/atomisator/parser/docs/README.txt`) is as follows:

```
=================
atomisator.parser
=================

The parser knows how to return a feed content, with
the `parse` function, available as a top-level function::

>>> from atomisator.parser import Parser

This function takes the feed url and returns an iterator
over its content. A second parameter can specify a maximum
number of entries to return. If not given, it is fixed to 10::

>>> import os
>>> res = Parser()(os.path.join(test_dir, 'sample.xml'))
>>> res
<itertools.imap ...>

Each item is a dictionary that contain the entry::
```

```
>>> entry = res.next()
>>> entry['title']
u'CSSEdit 2.0 Released'

The keys available are:

>>> keys = sorted(entry.keys())
>>> list(keys)
    ['id', 'link', 'links', 'summary', 'summary_detail', 'tags',
     'title', 'title_detail']

Dates are changed into datetime::

>>> type(entry['date'])
>>>
```

Later, the doctest will evolve to take into account new elements or the required changes. Such doctests are also good documentation for developers who want to use the package, and should be evolved with this particular usage in mind.

A common pitfall in writing tests in a document is that they can quickly transform it into an unreadable piece of text. If this happens, it should not be considered as part of the documentation anymore.

That said, some developers that are working exclusively through doctests often group their doctests into two categories: the ones that are readable and usable so that they can be a part of the package documentation, and the ones that are unreadable and are just used to build and test the software.

Many developers think that doctests should be dropped for the latter in favor of regular unit tests. Others even use dedicated doctests for bug fixes. So, the balance between doctests and regular tests is a matter of taste and is up to the team, as long as the published part of the doctests is readable. With modern testing frameworks, like `nose` or `py.test`, it's very easy to maintain both collection do doctests and classical function or class-based unit tests at the same time.

> When DDD is used in a project, focus on the readability and decide which doctests are eligible to be a part of the published documentation.

Summary

In this chapter, we have discussed the basic TDD methodology and provided more information on how to efficiently write meaningful automated tests for your software. We have reviewed a few ways of structuring and discovering tests. We have also mentioned popular testing tools that make writing tests fun and easy.

We have mentioned that tests can be used not only to verify application validity but also to assert some claims about its performance. This is a good starting point for the next two chapters, where we will discuss how to diagnose various performance issues of the Python application and also learn some powerful optimization principles and techniques.

Section 4: Need for Speed

This section is all about speed and resources, from plain optimization techniques that aim to squeeze as much as possible from the single-program process to various concurrency models that allow us to distribute and scale a system over many processor cores, or even computers. The reader will learn how to make their code faster and how to process data in a highly distributed manner.

The following chapters are included in this section:

- `Chapter 13`, *Optimization – Principles and Profiling Techniques*
- `Chapter 14`, *Optimization – Some Powerful Techniques*
- `Chapter 15`, *Concurrency*

13

Optimization - Principles and Profiling Techniques

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."

– Donald Knuth

This chapter is all about optimization—its general principles and common profiling techniques. We will discuss the most basic rules of optimization that every developer should be aware of. We will also learn how to identify application performance bottlenecks and use common profiling tools.

In this chapter, we will cover the following topics:

- The three rules of optimization
- Optimization strategy
- Finding bottlenecks

Let's discuss the three rules of optimization.

Technical requirements

Various profiling utilities for Python that are explained in this chapter require the Graphviz package. You can download it from `https://www.graphviz.org/download/`.

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

- `gprof2dot`
- `memprof`
- `memory_profiler`
- `pympler`
- `objgraph`

You can install these packages using the following command:

```
python3 -m pip install <package-name>
```

The code files for this chapter can be found at `https://github.com/PacktPublishing/Expert-Python-Programming-Third-Edition/tree/master/chapter13`.

The three rules of optimization

Optimization has a price, no matter what the results are. And the most important cost of optimization apart from the obvious development time is the increase in software complexity and reduction in maintainability. When a piece of code works, it might be better (sometimes) to leave it alone than to try making it faster at all costs. This means that if the optimization process has to be cost-effective, it must be done reasonably. The following are the three most basic optimization rules to keep in mind when doing any kind of optimization:

- Make it work first
- Work from the user's point of view
- No matter what, keep the code readable

In the following sections, we will explain these rules in detail.

Making it work first

A very common mistake that 's made by many developers is optimizing the code continuously from the very beginning. This is often a pointless and wasteful endeavor because the real bottlenecks are often located where you would have never thought they would be.

Even a seemingly simple application is usually composed of very complex interactions, and it is often impossible to guess how well it will behave until it is actually used by its users in the real production environment.

Of course, this is not a reason to write every function and algorithm with total negligence of performance problems and solve every problem by brute forcing it. Some performance *hot spots* of your application may be obvious from the very beginning, and therefore it makes sense to use performant solutions at an early stage of development. But you should do it very carefully and try to keep your code as simple as possible at all times. Your main goal should always be to make your code work first. This goal should not be hindered by optimization efforts.

For line-level code, the Python philosophy is that there's one and preferably only one way to do it. So, as long as you stick with a Pythonic syntax, as described in `Chapter 3`, *Modern Syntax Elements - Below the Class Level*, and `Chapter 4`, *Modern Syntax Elements - Above the Class Level*, your code (in the micro scale) should be fine. Often, writing less code is better and faster than writing more code.

Until your code works and is ready to be profiled, you should avoid the following practices:

- Any kind of caching or value memoization, even if it's a simple global dictionary
- Externalizing a part of the code in C or hybrid languages, such as Cython
- Using specialized external libraries that are focused mainly on performance optimization

Keep in mind that these are not strict rules. If doing any of the preceding things will, in the end, result in code that is simpler and more readable, you should definitely do that. Sometimes, using libraries like NumPy might ease the development of specific features and produce simpler and faster code in the end. Furthermore, you should not rewrite a function if there is a good library that does it for you. Also, for some very specialized areas, such as scientific calculation or computer games, the usage of specialized libraries and modules written using different languages might also be unavoidable from the beginning.

For instance, Soya 3D, which is a game engine on top of OpenGL (see `http://home.gna.org/oomadness/en/soya3d/index.html`), uses C and Pyrex for fast matrix operations when rendering real-time 3D.

> Optimization is carried out on programs that already work. As Kent Beck says, "*Make it work, then make it right, then make it fast*".

In the next section, we will take a look at how things work from the user's point of view.

Working from the user's point of view

I have once seen a team spend a lot of time and effort on optimizing the startup time of an application server that worked fine when it was already up and running. Once they had finished speeding it up, they announced their achievement to their customers. They were disappointed to notice that their customers didn't really care about it. This was because the optimization effort was not motivated by the user feedback and was only a bottom-up initiative of the developers. The people who built the system were launching the server multiple times every day. So, the startup time meant a lot to them. But sadly it wasn't that important to their customers.

While making a program start faster is a good thing from an absolute point of view, teams should be careful to prioritize the optimization work and ask themselves the following questions:

- Have I been asked to make it faster?
- Who finds the program slow?
- Is it really slow, or acceptable?
- How much will it cost to make it go faster?
- Is it worth it?
- Exactly what parts of the application need to be fast?

Remember that optimization has a cost, and that the developer's point of view is usually meaningless to customers (unless you are writing a framework or a library and the customer is a developer too).

> Optimization is not a game. It should be done only when necessary.

In the next section, we will learn how to keep our code readable and maintainable.

Keeping the code readable and maintainable

Even if Python tries to make the common code patterns the fastest, optimization techniques might obfuscate your code and make it really hard to read, understand, and develop. There's a balance to keep between readability/maintainability and performance.

Remember that optimization usually has no bounds. There will always be something that can be done to make your code a few milliseconds faster. So, if you have reached 90% of your optimization objectives, and the 10% left to be done would make your code utterly unreadable and unmaintainable, it might be a good idea to stop the work there or to look for other solutions.

> Optimization should not make your code unreadable. If it happens, you should look for alternative solutions, such as externalization or redesign. Look for a good compromise between readability and speed.

We'll take a look at optimization strategy in the next section.

Optimization strategy

Let's say your program has a real performance problem you need to resolve. Do not try to guess how to make it faster. Bottlenecks are often hard to find by simply looking at the code, and usually you will have to use a set of specialized tools to find the real problem cause.

A good optimization strategy can start with the following three steps:

- **Look for another culprit**: Make sure a third-party server or resource is not faulty
- **Scale the hardware**: Make sure the resources are sufficient
- **Write a speed test**: Create a scenario with speed objectives

Let's describe the preceding strategies in the following sections.

Looking for another culprit

Often, a performance problem occurs at production level, and the customer alerts you that it is not working as it used to when the software was being tested. Performance problems might occur because the application was not planned to work in the real world with a high number of users and ever-increasing amounts of data.

But if the application interacts with other applications, the first thing to do is to check if the bottlenecks are located on those interactions. For instance, if you use a database server or any kind of external service that needs to be communicated over the network, then it is possible that performance issues are due to service misuse (for example, heavy SQL queries) or many serial network connections that could be easily parallelized.

The physical links between applications should also be considered. Maybe the network link between your application server and another server in the intranet is becoming really slow due to a misconfiguration or congestion.

Good and up-to-date architecture design documentation that contains diagrams of all interactions and the nature of each link is invaluable in providing the overall picture of the whole system. And that picture is essential when trying to resolve performance issues that occur on the boundaries of many networked components.

> If your application uses third-party servers of resources, every interaction should be audited to make sure that the bottleneck is not located there.

Scaling the hardware

When the process requests more physical memory than what is currently available on your system, the system kernel may decide to copy some memory pages to the configured swap device. When some process tries to access the memory page that was already moved to the swap device, it will be copied back to RAM. This process is called swapping. This kind of memory management is common in today's operating systems, and the notion about it is important because most default system configurations use hard disk drives as their swap devices. And hard disks, even SSDs, are extremely slow compared to RAM.

Swapping in general is not a bad occurrence. Some systems may use swap devices for less accessed memory pages, even if there is a lot of free memory available, just to save resources in advance. But if memory pressure is very high and all processes really request and start to use more memory than is available, performance will drop drastically. In such situations, the system kernel might start swapping same memory pages back and forth, and will spend most of its time constantly writing and reading from disk. From a user's point of view, the system is considered dead at this stage. So, if your application is memory-intensive, it is extremely important to scale the hardware to prevent this.

While having enough memory on a system is important, it is also important to make sure that the applications are not acting crazy and eating too much memory. For instance, if a program works on big video files that can weigh in at several hundreds of megabytes, it should not load them entirely in memory, but rather work on chunks or use disk streams.

Note that scaling up the hardware (vertical scaling) has some obvious limitations. You cannot fit an infinite amount of hardware into a single server rack. Also, highly efficient hardware is extremely expensive (law of diminishing returns), so there is also an economical bound for this approach. From this point of view, it is always better to have a system that can be scaled out by adding new computation nodes, or workers (horizontal scaling). This allows you to scale out your service with commodity software that has the best performance/price ratio.

Unfortunately, designing and maintaining highly scalable distributed systems is both hard and expensive. If your system cannot be easily scaled horizontally or it is faster and cheaper to scale vertically, it may be better to scale it vertically instead of wasting time and resources on a total redesign of your system architecture. Remember that hardware invariably tends to be faster and cheaper with time. Many products stay in this sweet spot where their scaling needs to align with the trend of raising hardware performance (for the same price).

Writing a speed test

When starting with optimization work, it is important to work using a workflow similar to test-driven development rather than running some manual tests continuously. A good practice is to dedicate a test module in the application, with test functions that use code components that have to be optimized. Using this approach will help you track your progress while you are optimizing the application.

You can even write a few assertions where you set some speed objectives. To prevent speed regression, these tests can be left after the code has been optimized. Of course, measuring the execution time depends on the power of the CPU used, so it is extremely hard to collect objective measurement in a repeatable way on every environment. This is why speed tests are done best if they are executed on a carefully prepared and isolated environment. It is also crucial to make sure that only one speed test is done at a time. It is also better to concentrate on observing performance trends rather than on using hardcoded time limit assertions. Fortunately, many popular testing frameworks like `pytest` and `nose` have available plugins that can automatically measure test execution time and even compare the results of multiple test runs.

Let's take a look at finding bottlenecks in the next section.

Finding bottlenecks

Finding bottlenecks is usually done as follows:

- Profiling CPU usage
- Profiling memory usage
- Profiling network usage
- Tracing

Profiling is observing code behavior or specific performance metrics within a single process or execution thread working on a single host, and is usually done by the process itself. Adding code to an application that allows it to log and measure different performance metrics is called *instrumentation*. *Tracing* is a generalization of profiling that allows you to observe and measure across many networked processes running on multiple hosts.

Profiling CPU usage is explained in the next section.

Profiling CPU usage

The first source of bottlenecks is your code. The standard library provides all the tools that are needed 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 bit of overhead, but provides a good idea of where the time is consumed. A **statistical profiler**, on the other hand, samples the 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 the 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

Let's discuss the preceding ways to profile the code in the following sections.

Macro-profiling

Macro-profiling is done by running the application in a special mode, where the interpreter is instrumented to collect statistics on the code usage. Python 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. Anyway, 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
        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`: Total number of calls
- `tottime`: Total time spent in the function, excluding time spent in calls of sub functions
- `cumtime`: Total time spent in the function, including time spent in the calls of sub functions

The `percall` column to the left of `tottime` equals the `tottime / ncalls`, and the `percall` column to the left of `cumtime` equals the `cumtime / ncalls`.

These statistics are a print view of a 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 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 (low value of `percall` for `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 in order 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 the following diagram). **gprof2dot** (`https://github.com/jrfonseca/gprof2dot`) can be used to turn profiler data into a dot graph. You can download this simple PyPI script using `pip` and use it on the stats file that was created by the cProfile module (you will also require the open source Graphviz software, see `http://www.graphviz.org/`). The following is an example of `gprof2dot.py` invocation in a Linux shell:

```
$ gprof2dot.py -f pstats myapp.stats | dot -Tpng -o output.png
```

The advantage of gprof2dot is that it tries to be language agnostic. It is not limited to Python `profile` or `cProfile` output and can read from multiple other profiles, such as Linux perf, xperf, gprof, Java HPROF, and many others:
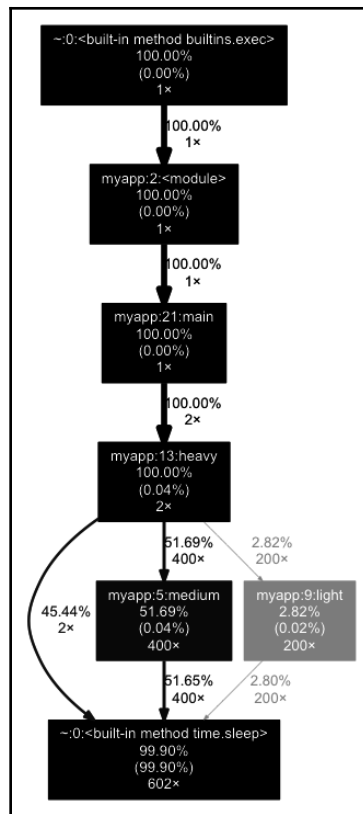
Figure 1: An example of a profiling overview diagram that was generated with gprof2dot

The preceding diagram that was generated by gprof2dot shows different code paths that were executed by the program and the relative time spent in each path. It is great for exploring the performance patterns of large applications. 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, it is sometimes necessary to do more profiling work that tests just a part of the program. This is done by manually instrumenting a part of the code in a speed test.

For instance, the `cProfile` module can be used in a form of decorator, as in 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)

@profile('time')
def main():
    for i in range(2):
        heavy()

if __name__ == '__main__':
    main()
```

This approach allows for testing only selected parts of the application and sharpens the statistics output. This way, you can collect many isolated and precisely targeted profiles on a single application run, as follows:

```
$ python3 cprofile_decorator.py
===== 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)


===== main() profile =====
Wed Apr 10 03:11:57 2019
/var/folders/jy/wy13kx0s7sb1dx2rfsqdvzdw0000gq/T/tmp6c0y2oxj

         62 function calls in 8.663 seconds

   Ordered by: internal time
   List reduced from 27 to 3 due to restriction <3>

   ncalls tottime percall cumtime percall filename:lineno(function)
        1 8.662 8.662 8.662 8.662 {method 'enable' of '_lsprof.Profiler'
objects}
        1 0.000 0.000 0.000 0.000 {built-in method posix.lstat}
        8 0.000 0.000 0.000 0.000
/usr/local/Cellar/python/3.7.2_2/Frameworks/Python.framework/Versions/3.7/l
ib/python3.7/random.py:224(_randbelow)
```

But at this stage, having a list of callees is probably not interesting, as the function has already been pointed out as the one to optimize. The only interesting information is to know how fast it is, and then enhance it.

`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 (`time.time` or `time.clock`), as shown in the following example:

```
>>> from myapp import light
>>> import timeit
>>> t = timeit.Timer('main()')
>>> t.timeit(number=5)
10000000 loops, best of 3: 0.0269 usec per loop
10000000 loops, best of 3: 0.0268 usec per loop
10000000 loops, best of 3: 0.0269 usec per loop
10000000 loops, best of 3: 0.0268 usec per loop
10000000 loops, best of 3: 0.0269 usec per loop
5.6196951866149902
```

This 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 handy to use within an existing application.

> A deterministic profiler will provide results depending on what the computer is doing, and so results may vary each time. Repeating the same test multiple times and making averages provides more accurate results. Furthermore, some computers have special CPU features, such as **SpeedStep**, which might change the results if the computer is idling when the test is launched. So, continually repeating the test is good practice for small code snippets. There are also various caches to keep in mind, such as DNS caches or CPU caches.

The results of `timeit` should be used with caution. It is a very good tool to objectively compare two short snippets of code, but it also allows you to easily make dangerous mistakes that will lead you to confusing conclusions. Here, for example, is the comparison of two innocent snippets of code with the `timeit` module that could make you think that string concatenation by addition is faster than the `str.join()` method:

```
$ python3 -m timeit -s 'a = map(str, range(1000))' '"".join(a)'
1000000 loops, best of 3: 0.497 usec per loop

$ python3 -m timeit -s 'a = map(str, range(1000)); s=""' 'for i in a: s +=
i'
10000000 loops, best of 3: 0.0808 usec per loop
```

From `Chapter 3`, *Modern Syntax Elements - Below the Class Level*, we know that string concatenation by addition is not a good pattern. Despite some minor CPython micro-optimizations that were designed exactly for such use cases, it will eventually lead to quadratic runtime. The problem lies in nuances about the `setup` argument of the `timeit()` call (or the `-s` parameter in the command line) and how the range in Python 3 works. I won't discuss the details of the problem, but will leave it to you as an exercise. Anyway, here is the correct way to compare string concatenation in addition to the `str.join()` idiom under Python 3:

```
$ python3 -m timeit -s 'a = [str(i) for i in range(10000)]' 's="".join(a)'
10000 loops, best of 3: 128 usec per loop
$ python3 -m timeit -s 'a = [str(i) for i in range(10000)]' '
  s = ""
  for i in a:
      s += i
  '
1000 loops, best of 3: 1.38 msec per loop
```

Profiling memory usage is explained in the next section.

Profiling memory usage

Another problem you may encounter when optimizing an application is memory consumption. If a program starts to eat so much memory that the system begins to continuously swap, there is probably a place in your application where too many objects are being created or objects that are not needed anymore are still kept alive by some unintended reference. This kind of resource mismanagement isn't easy to detect through typical CPU profiling techniques. Sometimes, consuming enough memory to make a system swap may involve a lot of CPU work that can be easily detected with ordinary profiling techniques. But usually, performance drop can happen suddenly and in an unexpected moment that is unrelated to the actual programming error. It's often due to memory leaks that gradually consume memory over longer periods of time. This is why memory usage usually has to be profiled with specialized tools of different types.

Let's take a look at how Python deals with memory in the next section.

How Python deals with memory

Memory usage is probably the hardest thing to profile in Python when you use the CPython implementation. While languages such as C allow you to get the memory size of any element, Python won't easily let you know how much memory a given object consumes. This is due to the dynamic nature of the language, and the fact that memory management is not directly accessible to the language user.

Some raw details of memory management were already explained in `Chapter 9`, *Python Extensions in Other Languages*. We already know that CPython uses reference counting to manage object allocation. This is the deterministic algorithm that ensures that object deallocation will be triggered when the reference count of the object goes to zero. Despite being deterministic, this process is not easy to track manually and to reason about (especially in complex code bases). Also, deallocation of objects on the reference count level does not necessarily mean that the actual process heap memory is freed by the interpreter. Depending on the CPython interpreter compilation flags, system environment, or runtime context, the internal memory manager layer might decide to leave some blocks of free memory for future reallocation instead of releasing it completely.

Additional micro-optimizations in CPython implementation also make it even harder to predict actual memory usage. For instance, two variables that point to the same short string or small integer value may or may not point to the same object instance in memory (a mechanism called interning).

Despite being quite scary and seemingly complex, memory management in Python is very well documented in the official Python documentation (refer to `https://docs.python.org/3/c-api/memory.html`). Note that micro memory optimizations like string or integer interning in most cases can be ignored when debugging memory issues. Also, reference counting is roughly based on a simple principle—if a given object is not referenced anymore, it is removed. So, in the context of function execution, every local object will be eventually removed when the interpreter does the following:

- Leaves the function
- Makes sure that the object is not being used anymore

So, the following objects remain in memory for longer:

- Global objects
- Objects that are still referenced in some way

Reference counting in Python is handy and frees you from the obligation of manually tracking object references, and therefore you don't have to manually destroy them. But since developers don't have to care about destroying objects, memory usage might grow in an uncontrolled way if you don't pay attention to the way they use their data structures.

The following are the usual memory eaters:

- Caches that grow uncontrollably.
- Object factories that register instances globally and do not keep track of their usage, such as a database connector factory, which is used on the fly every time a database query is done.
- Threads that are not properly finished.
- Objects with a __del__ method and involved in a cycle are also memory eaters. In older versions of Python (prior to 3.4 version), the garbage collector would not break the reference cycle since it could not be sure which object should be deleted first. Hence, you would leak memory. Using this method is a bad idea in most cases.

Unfortunately, when writing C extensions using the Python/C API, the management of reference counts must be done manually with `Py_INCREF()` and `Py_DECREF()` macros. We discussed the caveats of handling reference counts and reference ownership earlier in `Chapter 9`, *Python Extensions in Other Languages*, so you should already know that it is a pretty hard topic riddled with various pitfalls. This is the reason why most memory issues are caused by C extensions that are not written properly.

Profiling memory is explained in the next section.

Profiling memory

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 reference to memory, you cannot release it, and 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 not needed anymore but were not removed. This stops the interpreter from releasing resources, but is not the same situation as a memory leak in C. Of course, there is always the exceptional case of C extensions, but they are a different kind of beast that need completely different tools to diagnose, and cannot be easily inspected from Python code.

So, memory issues in Python are mostly caused by unexpected or unplanned resource acquiring patterns. It happens very rarely that this is the effect of real bugs caused by 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, and you will deal with them very rarely, if ever. Thus, the most so-called memory leaks in Python are mainly caused by overblown complexity of the software and subtle interactions between its components that are really hard to track. In order to spot and locate such deficiencies of 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. It's a pretty difficult 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 after and before 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. But 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 seems to be no longer 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/`): It is declared to work on Python 2.6, 2.7, 3.1, 3.2, and 3.3, and some POSIX-compliant systems (macOS and Linux). Last updated in December 2016.
- **memory_profiler** (`https://pypi.python.org/pypi/memory_profiler`): It is declared to support the same Python versions and systems as Memprof, but the code repository is tested with Python 3.6. Actively maintained.

- **Pympler** (`http://pythonhosted.org/Pympler/`): It is declared to support Python 2.7, all versions of Python 3 from 3.3 to 3.7, and is OS independent. Actively maintained.
- **objgraph** (`https://mg.pov.lt/objgraph/`): It is declared to support Python 2.7, 3.4, 3.5, 3.6, and 3.7, and is OS independent. Actively maintained.

Note that the preceding information about compatibility is based purely on trove classifiers that are used by the latest distributions of featured packages, and declaration from the documentation and inspection of projects' build pipeline definitions. This could easily have changed since this book was written.

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 only on projects that are known to work well with the latest release of Python (that is, Python 3.7) on different operating systems. This tool is **objgraph**. Its APIs seem to be a bit clumsy and have a very limited set of functionalities. But it works, does what it needs to 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. Because of its wide support of Python versions in OS independence, we will focus only on **objgraph** when discussing examples of memory profiling. The other tools mentioned in this section are also exciting pieces of software, but you need to research them by yourself.

Let's take a look at the `objgraph` module in the next section.

objgraph

`objgraph` is a simple module for creating diagrams of object references that should be useful when hunting memory leaks in Python. It is available on PyPI, but it is not a completely standalone tool and requires Graphviz in order to create memory usage diagrams. For developer-friendly systems like macOS or Linux, you can easily obtain it using your preferred system package manager (for example, `brew` for macOS, `apt-get` for Debian/Ubuntu). For Windows, you need to download the Graphviz installer from the project page (refer to `http://www.graphviz.org/`) and install it manually.

`objgraph` provides multiple utilities that allow you to list and print various statistics about 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 we mentioned previously, `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 diagramming utilities of that library 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 2* and *Figure 3*. Here is the code that was used to create these diagrams:

```
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)
```

The following diagram shows the diagram of all references held by `words` and `quote`, and `counts` objects:
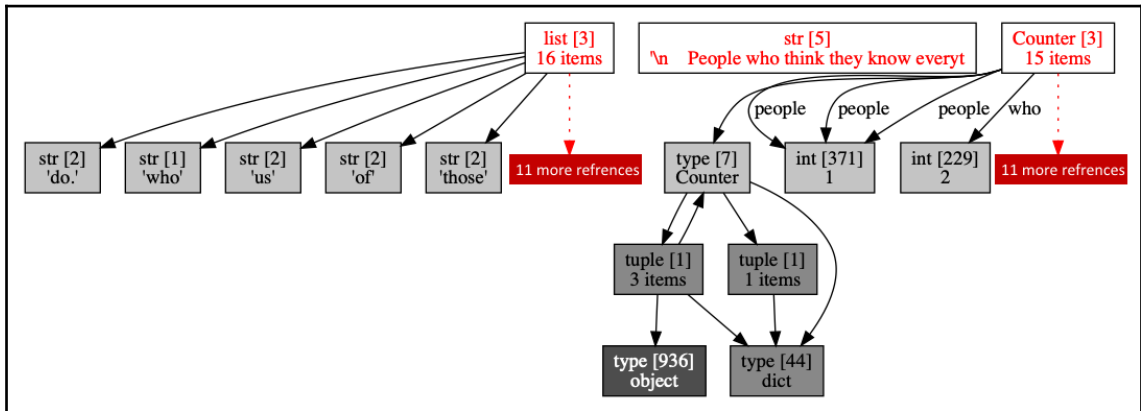
Figure 2: An example result of the show_refs() diagram from the graph_references() function

The following diagram shows only objects that hold references to the objects that we passed to the `show_backrefs()` function. They are called *back references* and are really helpful in finding objects that stop other objects from being deallocated:
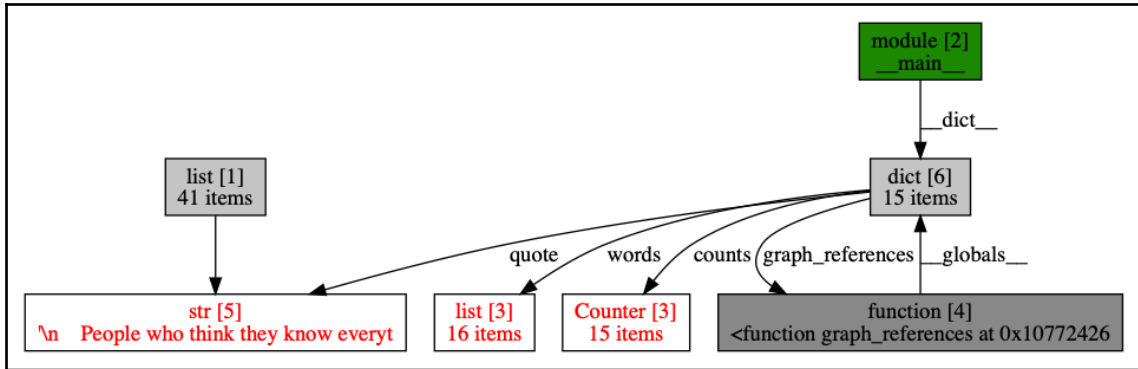
Figure 3: An example result of the show_backrefs() diagram from the graph_references() function

> **TIP**
>
> A basic installation of the `objgraph` package does not install the Graphviz software that is required to generate diagrams in bitmap form. Without Graphviz, it will output diagrams in DOT format special graph description language. Graphviz is a very popular piece of software that is often found in operating system package repositories. You can also download it from `https://www.graphviz.org/`.

In order to show how `objgraph` may be used in practice, let's review an example of code that may create memory issues under certain versions of Python. As we already noted multiple times in this book, CPython has its own garbage collector that exists independently from its reference counting mechanism. It's 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 tracking the number of references. Here is the simplest example:

```
x = []
y = [x]
x.append(y)
```

Such a situation is visually presented in the following diagram. In the preceding case, even if all external references to x and y objects will be 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 the 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:
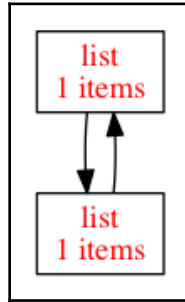
Figure 4: An example diagram of cyclic references between two objects

The real problem starts when at least one of the objects in such a cycle has the custom __del__() method defined. It 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 a part of the language standard, starting from Python 3.4. Anyway, this may still be a problem for packages that worry about backwards compatibility and target a wide spectrum of Python interpreter versions. The following snippet of code allows you to show difference in behavior of 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'))


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 under 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 in order to see if 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 a diagram, which 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 defined in red to make spotting such issues easier:
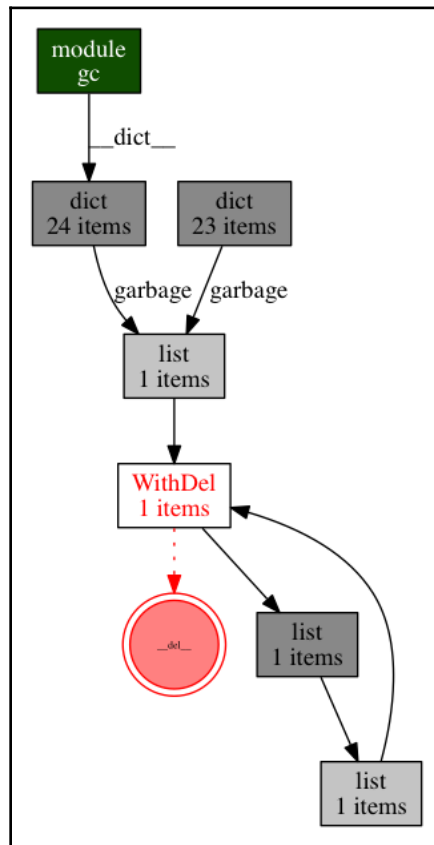
Figure 5: A diagram showing an example of cyclic references that can't be picked by the Python garbage collector prior to version 3.4

In the next section, we will discuss C code memory leaks.

C code memory leaks

If the Python code seems perfectly fine and the 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 some 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:

- Write a separate test for each API unit or functionality of an extension you are suspecting to leak memory
- Perform the test in a loop for an arbitrarily long time in isolation (one test function per run)
- 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 really pays off in the long run. You can always ease your work by reusing some of the testing tools that were introduced in `Chapter 12`, *Test-Driven Development*. Utilities such as `pytest` and `tox` were perhaps not designed exactly 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 memory, you can finally start actual debugging. If you're lucky, a simple manual inspection of the isolated source code section may give 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 fighting memory leaks in compiled code that should be in every programmer's toolbelt is **Valgrind**. It 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.

Profiling network usage is explained in the next section.

Profiling network usage

As I said earlier, an application that communicates with third-party programs such as databases, caches, web services, or an authentication server can be slowed down when those applications are slow. This can be tracked with a regular code profiling method on the application side. But if the third-party software works fine on its own, the culprit may be in the network.

The problem might be misconfigured network hardware, a low-bandwidth network link, or even a high number of traffic collisions that make computers send the same packets several times.

Here are a few elements to get you in. To find out what is going on, there are the following three fields to investigate at first:

- Watch the network traffic using tools such as the following:
    - ntop (Linux only, `http://www.ntop.org`)
    - Wireshark (`www.wireshark.org`)

- Track down unhealthy or misconfigured devices using monitoring tools based on the widely used SNMP protocol (`http://www.net-snmp.org`).
- Estimate the bandwidth between two computers using a statistical tool like `Pathrate` (`https://www.cc.gatech.edu/~dovrolis/bw-est/pathrate.html`).

If you want to delve further into network performance issues, you may also want to read *Network Performance Open Source Toolkit, Wiley* by Richard Blum. This book exposes strategies to tune the applications that are heavily using the network and provides a tutorial to scan complex network problems.

High Performance MySQL, O'Reilly Media by Jeremy Zawodny, is also a good book to read when writing an application that uses MySQL.

Let's take a look at tracing network transactions in the next section.

Tracing network transactions

Nowadays, with the advent of microservice architectures and modern container orchestration systems, it is very easy to build large distributed systems. It happens very often that distributed applications behave slow—not because the network is slow, but because there is too much communication between application components. Complex distributed systems can have tens or even hundreds of communicating services and microservices. Very often, those services are replicated across many computing nodes with various hardware characteristics. These services often communicate with multiple backing services through many middlewares and intermediate layers, like caching proxies and authentication servers. It's not a rare situation when a single user interaction under the hood, like an HTTP API request or web page load, can involve a layered communication happening between multiple servers.

In such highly distributed systems, the hardest thing may be to identify a single service that creates a performance bottleneck. Classic tools for code profiling usually work in an isolated environment and instrument the behavior of a single system process. Some monitoring software can, of course, do non-deterministic profiling on working production code, but it is useful only for general statistical performance analysis and only accidentally allows you to discover problematic hot spots. If you need to diagnose performance issues of a single well-defined interaction scenario, for instance, user login, you'll have to use a completely different approach.

The technique that is extremely helpful in inspecting complex network transactions in a distributed system is called *tracing*. Tracing requires every component in the distributed system to have similar instrumentation code that marks every inbound and outbound communication with unique transaction identifiers. If the instrumented service receives a request with some transaction identifier (or multiple identifiers) and needs to query other services during that request processing, it adds those identifiers to its own requests and creates a new identifier per every request made. In systems where the majority of communication happens through the HTTP protocol, the natural transport mechanism for these transaction identifiers are HTTP headers. Thanks to this, every transaction can be dissected into multiple subtransactions, and this way it is possible to trace the entire network traffic that was required to process every user interaction (see the following screenshot).

Usually, every service logs all processed transactions to a secondary service that's responsible for aggregating tracing data and also adds various metadata, like the start and end time of the transaction, hostname, and number of bytes sent/received. Such time and tag annotated transactions are often called spans. Many tracing systems allow you to define custom metadata that will be included in spans to ease debugging and monitoring:
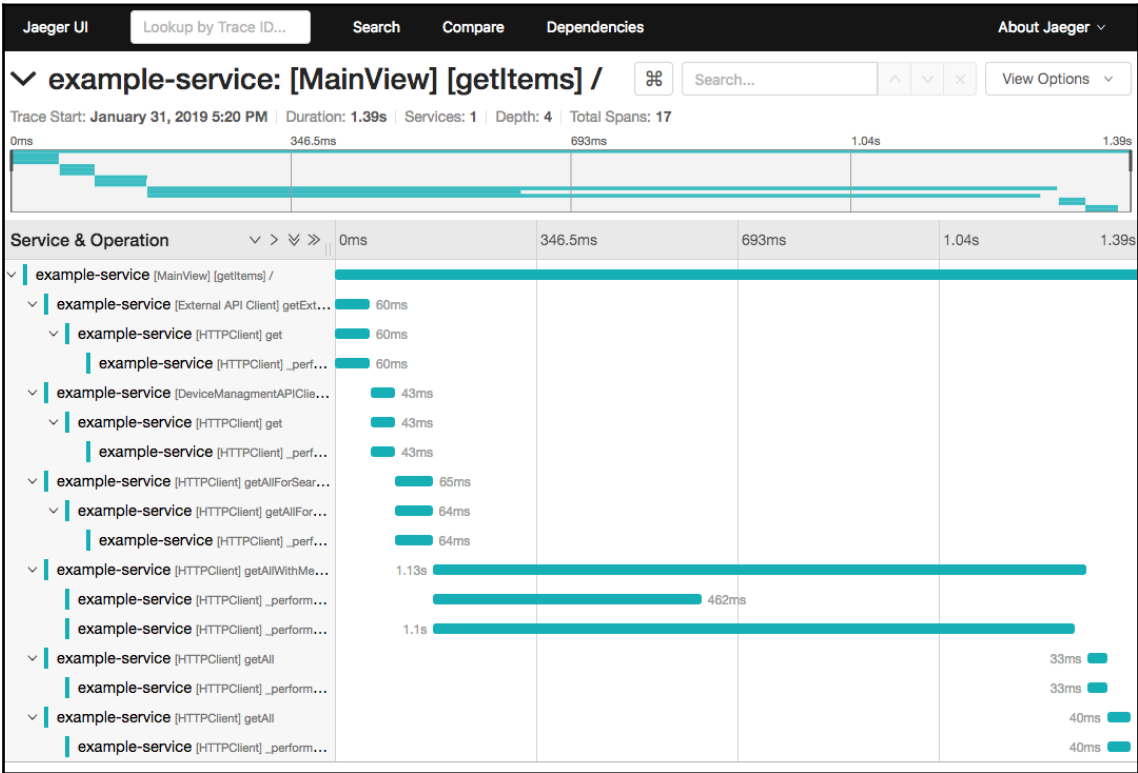
Figure 6: Example of tracing information for some example service presented in the Jaeger tracer

The most important aspect of every tracing solution is the careful selection of a proper tracing protocol and trace collection/aggregation system. A good choice for tracing is OpenTracing (`https://opentracing.io`), which is advertised as a "consistent, expressive, vendor-neutral API for distributed tracing and context propagation". It provides official libraries for nine programming languages (Python, Go, JavaScript, Java, Python, Ruby, PHP, Objective-C, C++, C#), and so it is good even for teams that build their products using different technology stacks. OpenTracing is neither a standard nor a complete working program. It's an API specification; a collection of frameworks and libraries, and documentation. With OpenTracing libraries, you can instrument your code and connect it to *tracers*, which are actual systems for collecting, aggregating, and presenting tracing results. A good tracer implementation to start with is Jaeger (`https://www.jaegertracing.io`). You should be able to local a Jaeger instance in just a few minutes using a pre-built Docker image that is published on Docker Hub under the `jaeger` name.

Summary

In this chapter, we've learned about the following three basic rules of optimization:

- Make it work first
- Work from the user's point of view
- No matter what, keep the code readable

With these rules in mind, we reviewed some of the tools and techniques that allow us to identify performance bottlenecks of various kinds. Now that you know how to diagnose and identify performance issues, you're ready to start mitigating them. In the next chapter, we will review popular and powerful optimization techniques and strategies that can be easily applied to the vast majority of performance issues.

14
Optimization - Some Powerful Techniques

Optimization is the process of making an application work more efficiently without modifying its functionality and accuracy. In the previous chapter, we learned how to identify performance bottlenecks and observe resource usage in code. In this chapter, we will learn how to use that knowledge to make an application work faster and use resources with greater efficiency.

Optimization is not a magical process. It is done by following a simple algorithm synthesized by Stefan Schwarzer at EuroPython 2006. The original pseudocode of this example is as follows:

```
def optimize():
    """Recommended optimization"""
    assert got_architecture_right(), "fix architecture"
    assert made_code_work(bugs=None), "fix bugs"
    while code_is_too_slow():
        wbn = find_worst_bottleneck(just_guess=False,
                                    profile=True)
        is_faster = try_to_optimize(wbn,
                                    run_unit_tests=True,
                                    new_bugs=None)
        if not is_faster:
            undo_last_code_change()

# By Stefan Schwarzer, EuroPython 2006
```

This example may not be the neatest or clearest, but the code captures almost all of the important aspects of an organized optimization procedure. The main things we can learn from it include the following:

- Optimization is an iterative process where not every iteration leads to better results
- The main prerequisite is that code is verified by tests
- Optimizing the current application bottleneck is key

Making your code work faster is not an easy task. In the case of abstract mathematical problems, the solution often lies in choosing the right algorithm and proper data structures. However, it is difficult to provide generic or universal tips and tricks that can be used to solve any algorithmic problem. There are, of course, some generic methodologies for designing a new algorithm, or even meta-heuristics that can be applied to a large variety of problems, but they are generally language-agnostic and are thus beyond the scope of this book.

There is a wide range of performance issues that are caused by either code quality defects or application usage contexts. These kind of problems can often be solved using common programming approaches, either with specific performance-oriented libraries and services or with proper software architecture design. Common non-algorithmic culprits of bad application performance in Python include the following:

- Incorrect usage of basic built-in types
- Too much complexity
- Hardware resource usage patterns that do not match the execution environment
- Long response times from third-party APIs or backing services
- Requiring too much work in time-critical parts of the application

More often than not, solving such performance issues does not require advanced academic knowledge, only good software craftsmanship—and a big part of craftsmanship is knowing when to use the proper tools. Fortunately, there are some well-known patterns and solutions for dealing with performance problems.

In this chapter, we will discuss some popular and reusable solutions that allow you to non-algorithmically optimize your program, by covering the following topics:

- Defining complexity
- Reducing complexity
- Using architectural trade offs
- Caching

Technical requirements

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

- `pymemcached`

You can install these packages using the following command:

```
python3 -m pip install <package-name>
```

The code files for this chapter can be found at `https://github.com/PacktPublishing/Expert-Python-Programming-Third-Edition/tree/master/chapter14`.

Defining complexity

Before we dig further into optimization techniques, let's define what exactly we are going to deal with. From this chapter's introduction, we know that focusing on improving application bottlenecks is critical for successful optimization. A bottleneck is a single component that severely limits the capacity of a program or computer system. Code with performance issues is usually hit with just a single bottleneck. We have discussed some profiling techniques in the previous chapter, so you should already be familiar with the tools required to locate and isolate troublesome pieces of code. If your profiling results show that there are a few places that need immediate improvement, you should try to treat each area as a separate component and optimize them independently.

If there is no explicit or evident bottleneck in your application but your application still under-performs, you may find yourself in a predicament. The success of the optimization process is proportionate to the performance impact of the optimized bottleneck, so optimizing every small component—particularly if it does not substantially impact execution time or resource consumption—may not solve the problem. If your application does not seem to have any real bottlenecks, there's a possibility that you have missed something during application profiling. Try using different profiling strategies or tools, or even looking at the application's performance from a new perspective, such as considering CPU usage, memory, I/O operations, or network throughput. If that does not help, you should consider revising your software architecture.

If you have successfully found a single and integral component that noticeably limits your application's performance, you are ready to start the optimization process. There is a high chance that even a minor code improvement will dramatically improve code execution time or resource usage. As we mentioned earlier, the benefits of optimization is proportionate to the bottleneck size.

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 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	What it means
1 to 10	Not complex
11 to 20	Moderately complex
21 to 50	Really complex
More than 50	Too complex

Cyclomatic complexity is more of a code quality score than a metric that objectively judges its performance. It does not replace the need of 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 about other canonical ways of expressing complexity, including the 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; `pycodestyle` (with the `mccabe` plugin) is a popular measurement tool in Python, which we introduced in `Chapter 6`, *Choosing Good Names*.

The big O notation

The most canonical method of defining function complexity is the **big O notation**. This metric defines how an algorithm is affected by the size of input data. For instance, does an algorithm scale linearly with the size of the input data, or quadratically?

Manually calculating 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 input data. Knowing the complexity of your application's components gives you the ability to detect and focus on aspects that will significantly slow down 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 input data grows. The idea is to try and categorize the algorithm in one of the following categories, even if it is an approximation:

Notation	Type
O(1)	Constant; does not depend on the input data
O(n)	Linear; will grow as n grows
O($n \log n$)	Quasi linear
O(n^2)	Quadratic complexity
O(n^3)	Cubic complexity
O($n!$)	Factorial complexity

For instance, we already know from `Chapter 3`, *Modern Syntax Elements - Below the Class Level*, that a `dict` lookup has an average complexity of O(1); it is considered constant, regardless of how many elements are in the `dict`. However, looking through a list for a particular item is O(n).

To better understand this concept, let's take look at the following example:

```
>>> def function(n):
...     for i in range(n):
...         print(i)
...
```

In the preceding function, the `print` statement will be executed n times. Loop speed will depend on n, so its complexity that's expressed using the big O notation will be $O(n)$.

If the function has conditions, the correct notation to keep is the highest one, as follows:

```
>>> 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 $O(n)$.

When discussing complexity expressed in big O notation, we usually review the worst-case scenario. While this is the best method for defining complexity when comparing two independent algorithms, it may not be the best approach in every practical situation. Many algorithms change the runtime performance, depending on the statistical characteristic of input data, or amortize the cost of worst-case operations by doing 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 `list` in CPython uses an array with overallocation for the internal storage instead of linked lists (see `Chapter 3`, *Modern Syntax Elements - Below the Class Level*, for more information). If an array is already full, appending a new element requires the allocation of a new array and to copy all existing elements (references) to a new area in the memory. If we look at this from the point 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 this complexity over a sequence of operations, we will see that the *average complexity* of `list.append()` is $O(1)$—and this is actually a great result.

When solving problems, we often already know about our input data in great detail, including its size or statistical distribution. When optimizing an application, it is always worth using every bit of available knowledge about the input data. This is where another problem of worst-case complexity can start to show up. The big O notation is intended to analyze the limiting behavior of a function when input tends toward large values or infinity, rather than 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 dumps all the details about both an implementation and its data characteristics to show you how your program will behave asymptotically. It works for arbitrarily large inputs that you may not even need to consider.

For instance, let's assume that you have a problem with data consisting of n independent elements. Let's also suppose that you know two different ways of solving this problem: *program A* and *program B*. You know that *program A* requires $100n^2$ operations to complete the task, and *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 $100\,n^2 > 5\,n^3$ 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.

In the next section, we will take a look at how to reduce complexity by selecting appropriate data structures.

Reducing complexity by choosing proper data structures

To reduce the complexity of code, 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 don't need to perform many element index lookups, but it can have a negative performance impact in some critical code sections—especially if it is done over very large lists.

If you need to search over 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 binary search:

```
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 at least $O(n \log n)$ complexity. This is a worse class than $O(n)$, so sorting the whole list to then perform 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.

In the next section, we will see how to use a set instead of a list.

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 with 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, as in `dict`. `set` 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 if 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.

When you try to reduce the complexity of an algorithm, carefully consider your data structures.

In the next section, we will take a look at collections.

Using collections

The `collections` module provides high-performance alternatives to built-in container types. The main types that are available in this module 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'll discuss these types in the next section.

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 middle 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 $O(1)$. Still, pops and appends are generally faster when performed on linked lists instead of arrays. The situation changes dramatically when the element needs to be added to an arbitrary point of sequence, however. 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, the `deque` in place of `list` may provide substantial performance improvement. Remember to always profile your code before switching from `list` to `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 `list` and `deque` may not be even noticeable, as follows:

```
$ python3 -m timeit \
> -s 'sequence=list(range(10))' \
> 'sequence.append(0); sequence.pop();'
1000000 loops, best of 3: 0.168 usec per loop
$ 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, as follows:

```
$ 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
$ 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 can see, the difference gets bigger as the size of the sequence grows. The following code snippet is an example of the same test performed on lists that contain 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
$ 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. Starting from Python 2.6, there is a separate queue module in Python's standard library that provides basic implementation for FIFO, 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 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 lightly faster than the dict.setdefault() method, as follows:

```
$ python3 -m timeit \
> -s 'd = {}'
> 'd.setdefault("x", None)'
10000000 loops, best of 3: 0.153 usec per loop
$ 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 isn't great in the preceding example because 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 $O(1)$, as we saw in the *Dictionaries* section in Chapter 3, *Modern Syntax Elements - Below the Class Level*. There is no way to have a complexity class lower than $O(1)$, but it is indisputably faster in some situations. 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:

```
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
...
>>> list(d.items())
[('i', 4), ('p', 2), ('s', 4), ('m', 1)]
```

namedtuple

`namedtuple` is a class factory that takes a type name and a list of attributes and creates a class out of it. The class can then be used to instantiate a tuple-like object and also provides accessors for its 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 `tuple`, so gaining access to its elements by 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. 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 a $O(1)$ average complexity of `get` and `set` operations.

The first 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.

The fact that `namedtuple` is based on `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. It is highly optimized to ensure good performance independently from collection size, but as mentioned, *O*(1) complexity is actually only considered an *average* level of 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

> 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 instead of killing the code's 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.

In the next section, we'll look at how we can use architectural trade-offs.

Using 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 acceptable to the user. For example, consider a program that deals with complex optimization problems, such as the **Traveling Salesman Problem** (**TSP**) or **Vehicle Routing Problem** (**VRP**). Both problems are *NP-hard* problems in combinatorial optimization. The exact algorithms that have low complexity for such problems are not known; this means that the size of a problem that can be practically solved is greatly limited. For very large inputs, it is unlikely that you'll be able to provide the correct solution in enough time.

Fortunately, it's likely that a user will be interested not 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. They concentrate on speed, but 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 problems that can solve extremely large TSP problems within a reasonable amount of time. They also have a high probability of producing 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. Some popular metaheuristic algorithms include the following:

- Simulated annealing
- Genetic algorithms
- Tabu search
- Ant colony optimization
- Evolutionary computation

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 be instead dominated by some 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 the following diagram. Here, you cannot reduce the waiting time, but you can change the way users will perceive it:
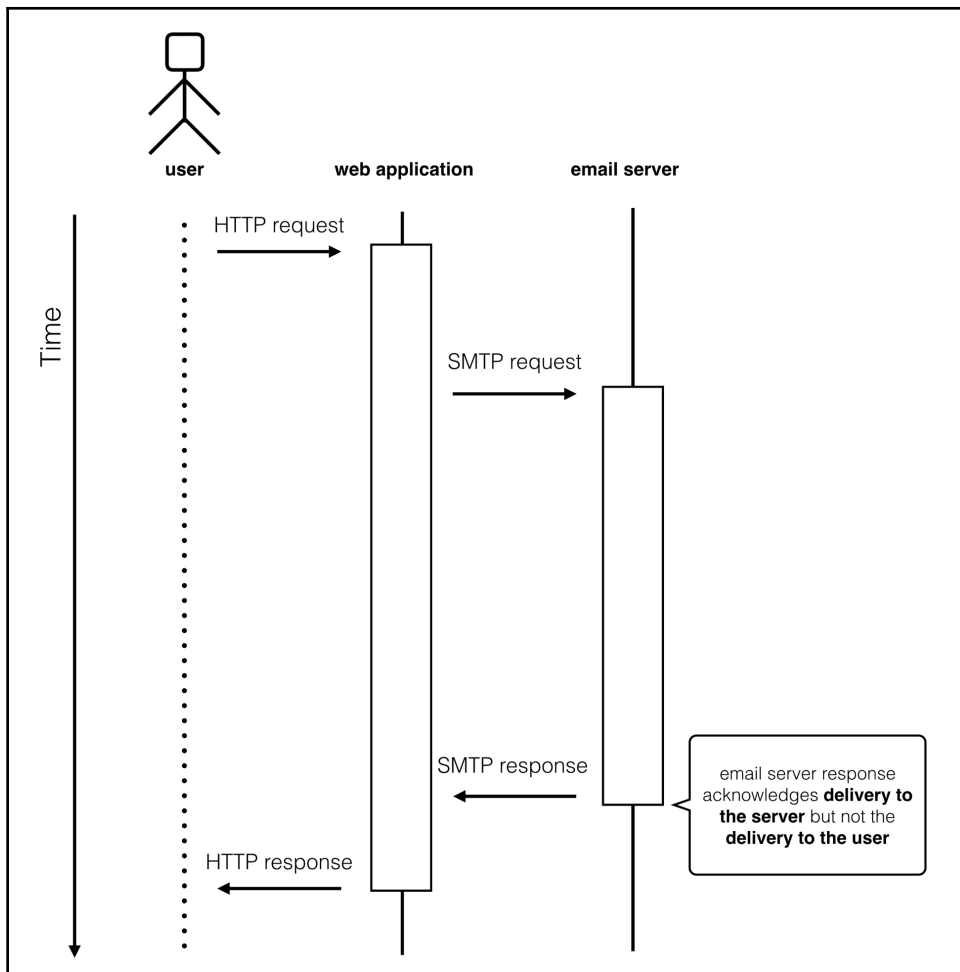
Figure 1: An example of synchronous email delivery in a web application

The usual solution to this is kind of problem is to use message or task queues. 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 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.

The updated flow of processing requests via a task queue is illustrated in the following diagram:
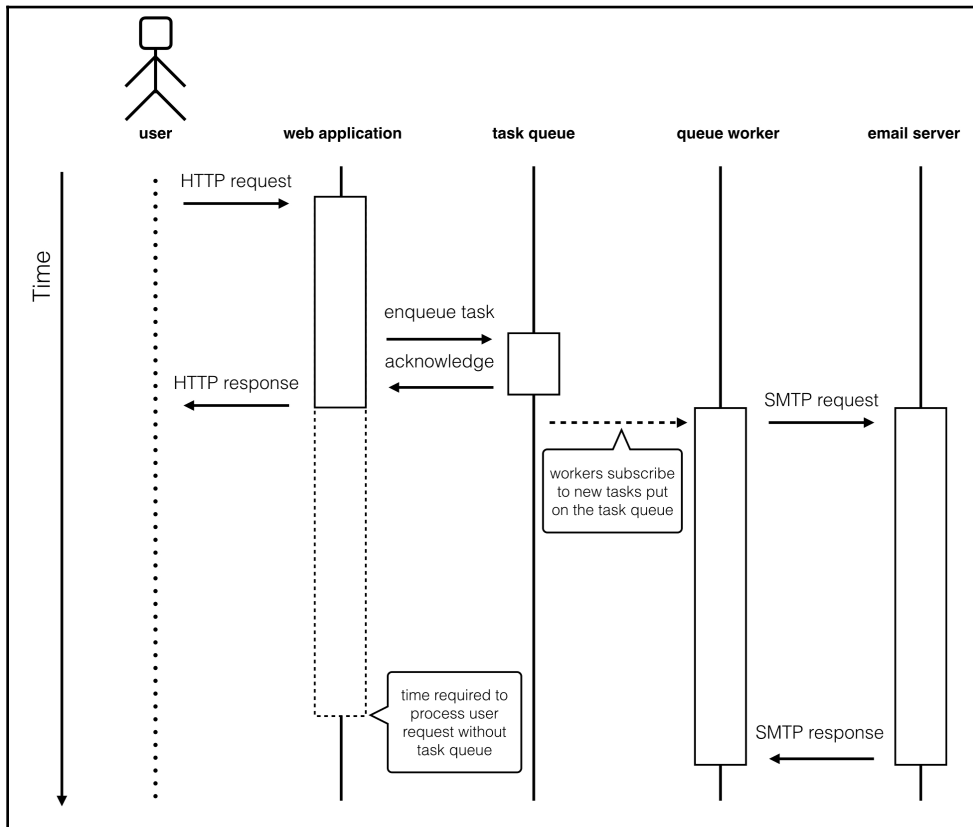
Figure 2: An example of asynchronous email delivery in a web application

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 thus handle greater load.
- 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 the preceding diagram, adding an asynchronous task process to your application inevitably increases the complexity of the whole system's architecture. You will, however, 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. The most popular one among Python developers is **Celery** (http://www.celeryproject.org/). Celery is a fully-fledged task queue framework with support of multiple message brokers that also allows for the scheduled execution of tasks. It can even replace your cron jobs. If you need something simpler, then RQ (http://python-rq.org/) might be a good alternative. RQ is a lot simpler than Celery and uses Redis key/value storage as its message broker (**RQ** actually stands for **Redis Queue**).

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
- Possible *more than once* deliveries
- More services to maintain and monitor
- Larger processing delays
- More difficult logging

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. 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, **HyperLogLog**.

HyperLogLog 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, and tested for membership. HyperLogLog 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 takes 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 the following prerequisites are 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 nondeterministic but continues to be useful and valid for some period of time

In other words, a deterministic function always returns the same result for the same set of arguments, whereas a nondeterministic function returns results that may vary in time. Caching both types of results usually greatly reduces the time of computation and allows you to save a lot of computer 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 as follows:

- 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 needs 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's architecture stack. Sometimes, a cache may be as simple as a single global data structure (usually a `dict`) 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. This section 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 exactly the same input, so you 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`, *Python Extensions in Other Languages*.) 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):
    """ Return nth Fibonacci sequence number computed recursively
    """
    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, and there 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 diagram, which presents an example call tree, 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:
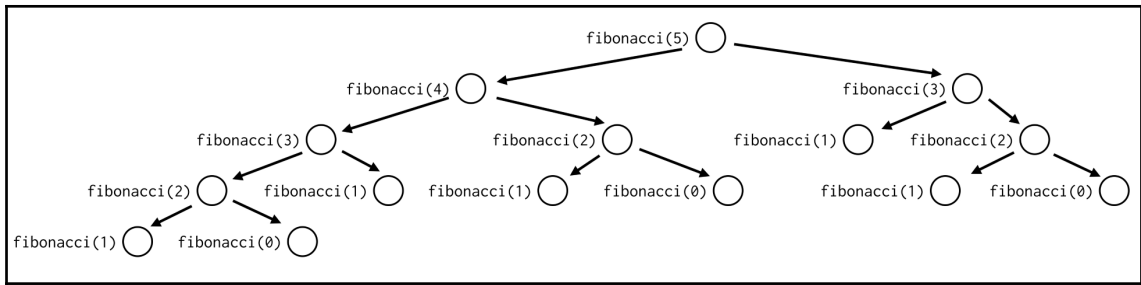
Figure 3: A call tree for the fibonacci(5) execution

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 memoizaton by constructing a very simple decorator, as follows:

```python
def memoize(function):
    """ Memoize the call to single-argument function
    """
    call_cache = {}

    def memoized(argument):
        try:
            return call_cache[argument]
        except KeyError:
            return call_cache.setdefault(
                argument, function(argument)
            )

    return memoized


@memoize
def fibonacci(n):
    """ Return nth Fibonacci sequence number computed recursively
    """
    if n < 2:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

We 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 $O(1)$ complexity, so this greatly reduces 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 the following diagram. 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 $O(n)$:
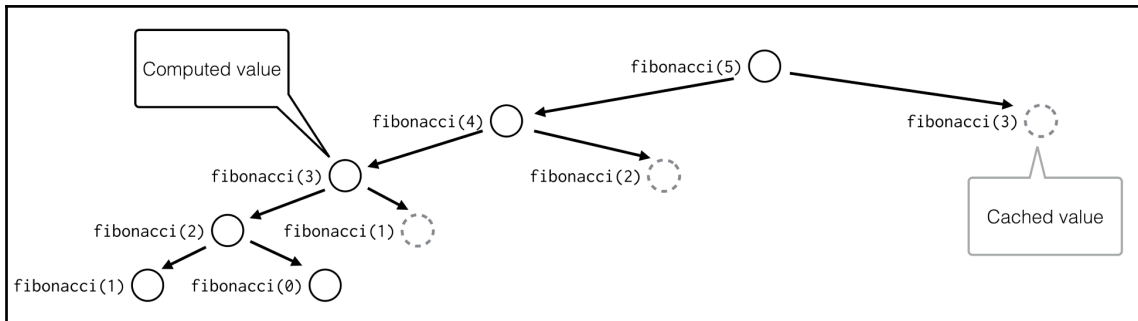
Figure 4: A call tree for the fibonacci(5) execution with memoization

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 need something more generic. Luckily, the Python standard library provides a very simple and reusable utility that can be used in most cases when caching the results of deterministic functions in-memory is required. This utility is the `lru_cache(maxsize, typed)` decorator from the `functools` module. The name comes from the LRU algorithm, which stands for **last recently used**. The following additional parameters allow for finer control of the 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 cached as the same result.

The usage of `lru_cache` in our Fibonacci sequence example would be as follows:

```python
@lru_cache(None)
def fibonacci(n):
    """ Return nth Fibonacci sequence number computed recursively
    """
    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 that memoization. Due to the fact that every execution of such a function may give 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 refreshing with a new value.

Non-deterministic functions that are usually a subject of caching 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 network connection (web APIs)
- Filesystems

So, in other words, non-deterministic caching is performed in any situation where pre-computed results are used temporarily. These results often represent a state that is consistent with the state of other system components—usually, the backing service.

Note that such an implementation is obviously a trade-off, and is therefore related to the techniques we looked at in the *Using architectural trade-offs* section. 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 the accuracy and/or completeness for speed and performance.

Of course, such caching is 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. If you want to save time and resources when communicating with the database, it is worth caching 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.

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.

In the next section, we will take a look at cache services.

Cache services

We have already said that non-deterministic caching can be implemented using local process memory, but it is actually rarely done that way. This is because local process memory is limited in its utility as storage in large applications.

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, it 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 that is already a trade-off between speed and consistency, how you can 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, ten 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 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 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.

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. There are many Python clients that differ slightly from each other, 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, `pymemcached`:

```
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 either as strings or binary blobs, and this isn't compatible with every native Python type. In fact, it is only compatible with one-strings. 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:

```
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 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.

The last problem is that Memcached, like many other caching services, does not tend to like very long key strings. The shorter is better. Long keys 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 are hash collisions. There is no hash function that guarantees that collisions will never occur, so always be sure to know and mitigate any potential risks.

Summary

In this chapter, you learned how to define the complexity of code, and looked at different ways to reduce it. We also looked at how to improve performance using architectural trade-offs. Finally, we explored exactly what caching is and how to use it to improve application performance.

The preceding methods concentrated our optimization efforts inside a single process. We tried to reduce code complexity, choose better data types, and reuse old function results. If that did not help, we tried to make trade-offs using approximations, doing less, or leaving work for later. We also briefly discussed the topic of message queues as a potential solution for performance problems. We will revisit this topic later in `Chapter 16`, *Event-Driven and Signal Programming*.

In the next chapter, we will discuss some techniques for concurrency and parallel-processing in Python that can also be considered tools when improving the performance of your applications.

15
Concurrency

Concurrency and one of its manifestations, parallel processing, is one of the broadest topics in the area of software engineering. It is so huge that it could take dozens of books and we would still not be able to discuss all of its important aspects and models.

This is why I won't try to fool you and from the very beginning state that we will barely touch the surface of this topic. The purpose of this chapter is to show you why concurrency may be required in your application, when to use it, and what the most important concurrency models that you may use in Python are, which are the following:

- Multithreading
- Multiprocessing
- Asynchronous programming

We will also 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 further research and reading. It is here to guide you through the basic ideas and help in deciding if you really need concurrency, and if so, which approach will best suit your needs.

In this chapter, we will cover the following topics:

- Why concurrency?
- Multithreading
- Multiprocessing
- Asynchronous programming

Technical requirements

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

- `aiohttp`

You can install these packages using the following command:

```
python3 -m pip install <package-name>
```

The code files for this chapter can be found
at https://github.com/PacktPublishing/Expert-Python-Programming-Third-Edition/tree/master/chapter15.

Why concurrency?

Before we answer the question w*hy concurrency*, we need to ask, w*hat is concurrency at all?*

The answer to the latter question may be surprising for someone who used to think that it is a synonym for **parallel processing**. First and foremost, concurrency is not the same as parallelism. Concurrency is also not a matter of application implementation. It is a property of a program, algorithm, or problem where parallelism is just one of the possible approaches to the problems that are concurrent.

Leslie Lamport in his *Time, Clocks, and the Ordering of Events in Distributed Systems* paper from 1976, 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 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 other than synchronous ways, but without the need for parallel execution.

So, once we know what concurrency really is, it is time to explain what all the fuss is about. When the 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 for solving large-scale problems or when you need to satisfy the following 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 (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

So, generally approaching concurrent problems concurrently is the best approach when the following scenarios apply:

- The scale of problems is so big that the only way to process them in an acceptable time or in the range of available resources is to distribute execution on multiple processing units that can handle the work in parallel
- Your application needs to maintain responsiveness (accept new inputs), even if it did not finish processing old inputs

These two classes of problems cover most situations where concurrent processing is a reasonable option. The first group of problems definitely needs the parallel processing solution, so it is usually solved with multithreading and multiprocessing models. The second group does not necessarily need be processed in parallel, so the actual solution really depends on the problem details. This group of problems 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 the input on a single processing unit. This is the reason why different and seemingly alternative or conflicting approaches to concurrency may be often 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.

Let's take a look at multithreading in the next section.

Multithreading

Developers often consider threading to be a very complex topic. While this statement is totally true, Python provides high-level classes and functions that ease the usage of threading. CPython implementation of threads unfortunately comes with some inconvenient details that make them less useful than in other languages. They are still completely fine for some sets of problems that you may want to solve, but not for as many as in C or Java.

In this section, we will discuss the 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 his or her work into threads that run simultaneously and share the same memory context. Unless your code depends on third-party resources, multithreading will not speed it up on a single-core processor, and will even add some overhead for thread management. 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 fact that the same context is shared among threads means you must protect data from uncontrolled concurrent accesses. If two intertwined threads update the same data without any protection, there might be a situation where subtle timing variation in thread execution can alter the final result in an unexpected way. To better understand this problem, imagine that there are two threads that increment the value of a shared variable in a non-atomic sequence of steps, for example:

```
counter_value = shared_counter
shared_counter = counter_value + 1
```

Now, let's assume that the `shared_counter` variable has the initial value of 0. Now, imagine that two threads process the same code in parallel, as follows:

Thread 1	Thread 2

`counter_value = shared_counter      #`
`counter_value = 0`
`shared_counter = counter_value + 1  #`
`shared_counter = 0 + 1`
	`counter_value = shared_counter        #`
	`counter_value = 0`
	`shared_counter = counter_value + 1  #`
	`shared_counter = 0 + 1`

Depending on the exact timing and how processor context will be switched, it is possible that the result of running two such threads will be either 1 or 2. Such a situation is called a **race hazard** or **race condition**, and is one of the most hated culprits of hard to debug software bugs.

Lock mechanisms help in protecting data, and thread programming has always been a matter of making sure that the resources are accessed by threads in a safe way. But unwary usage of locks can introduce a set of new issues on its own. The worst problem occurs when, due to a wrong code design, two threads lock a resource and try to obtain a lock on the other resource that the other thread has locked before. They will wait for each other forever. This situation is called a **deadlock,** and is similarly hard to debug. **Reentrant locks** help a bit in this by making sure a thread doesn't get locked by attempting to lock a resource twice.

Nevertheless, when threads are used for isolated needs with tools that were built for them, they might increase the speed of the program.

Multithreading is usually supported at the system kernel level. When the machine has a single processor with a single core, the system uses a **time slicing** mechanism. Here, the CPU switches from one thread to another so fast that there is an illusion of threads running simultaneously. This is done at the processing level as well. Parallelism without multiple processing units is obviously virtual, and there is no performance gain from running multiple threads on such hardware. Anyway, sometimes, it is still useful to implement code with threads, even if it has to execute on a single core, and we will look at a possible use case later.

Everything changes when your execution environment has multiple processors or multiple CPU cores for its disposition. Even if time slicing is used, processes and threads are distributed among CPUs, providing the ability to run your program faster.

Let's take a look at how Python deals with threads.

How Python deals with threads

Unlike some other languages, Python uses multiple kernel-level threads that can each run any of the interpreter-level threads. But the standard implementation of the CPython language comes with a major limitation that renders threads less usable in many contexts. All threads accessing Python objects are serialized by one global lock. This is done because much of the interpreter internal structures, as well as third-party C code, are not thread-safe and need to be protected.

This mechanism is called the **Global Interpreter Lock** (**GIL**), and its implementation details on Python/C API level were already discussed in the *Releasing GIL* section of `Chapter 9`, *Python Extensions in Other Languages*. The removal of GIL is a topic that occasionally appears on the Python-dev emailing list and was postulated by 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 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, there is little point in using threads to speed up the program since the GIL will globally serialize the execution of all threads. But remember that GIL cares only about Python code. In practice, the global interpreter lock is released on a number of blocking system calls and can be released in sections of C extensions that do not use any of Python/C API functions. This means that multiple threads can do I/O operations or execute C code in certain third-party extensions in parallel.

Multithreading allows you to efficiently utilize time when a program is waiting for a third-party resource. This is because a sleeping thread that has explicitly released the GIL can stand by 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 operating system 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 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 and IronPython (see `Chapter 1`, *Current Status of Python*). There has been some development of the GIL-free version of PyPy, but at the time of writing this book, it is still at an experimental stage and the documentation is lacking. It is based on **Software Transactional Memory** and is called PyPy-STM. It is really hard to say when (or if) it will be officially released as a production ready interpreter. Everything seems to indicate that it won't happen soon.

In the next section, we will discuss when we should use threading.

When should we use threading?

Despite the GIL limitation, threads can be really useful in some of the following cases:

- Building responsive interfaces
- Delegating work
- Building multiuser applications

Let's discuss the preceding cases in the next sections.

Building responsive interfaces

Let's say you ask your system to copy files from one folder to another through some program with a graphical user interface. The task will possibly be pushed into the background and the interface window will be constantly refreshed by the program. This way, you get live feedback on the progress of the whole process. You will also be able to cancel the operation. This is less irritating than a raw `cp` or `copy` shell command that does not provide any feedback until the whole work is finished.

A responsive interface also allows a user to work on several tasks at the same time. For instance, Gimp 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 not intended to increase performance but only to make sure that the user can still operate the interface, even if it needs to process some data for a longer period of time.

If such background tasks perform a lot of I/O operations, you are able to still get some benefit from multicore CPUs. Then, it's a *win-win* situation.

Delegating work

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 calls a different external program. For example, one is specialized in PDFs and another one in OpenOffice files.

Instead of processing all files in a sequence, by executing the right program and then storing the results into the database, 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 sum of all the work.

Note that such an approach is somewhat a hybrid between multithreading and multiprocessing. If you delegate the work to external processes (for example, using the `run()` function from the `subprocess` module), you are in fact doing work in multiple processes, so this has symptoms of multiprocessing. Still, in our scenario, we are mainly waiting for the processing of results being handled in separate threads, so it is still mostly multithreading from the perspective of the Python code.

The other common use case for threads is performing multiple HTTP 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, and additional round-trip time delays will be added to every such request. If you are communicating with some efficient service (Google Maps API, for instance), it is highly probable that it can serve most of your requests concurrently without affecting the response times of separate requests. It is then reasonable to perform multiple queries in separate threads. Remember that when doing an HTTP request, the maximum time is spent on reading from the TCP socket. This is a blocking I/O operation, so CPython will release the GIL when performing the `recv()` C function. This allows for great improvement of your application's performance.

Multiuser applications

Threading is also used as a concurrency base for multiuser applications. For instance, a web server will push a user request into a new thread and then will become idle, waiting for new requests. Having a thread dedicated to each request simplifies a lot of work, but requires the developer to take care of locking the shared resources. But this is not a problem when all the shared data is pushed into a relational database that takes care of the concurrency matters. So, threads in a multiuser application act almost like separate independent processes. They are under the same process only to ease their management at the application level.

For instance, a web server will be able to put all requests in a queue and wait for a thread to be available to send the work to it. Furthermore, it allows memory sharing that can boost up some work and reduce the memory load. The two very popular Python WSGI-compliant webservers **Gunicorn** (refer to `http://gunicorn.org/`) and **uWSGI** (refer to `https://uwsgi-docs.readthedocs.org`) allow you to serve HTTP requests with threaded workers in a way that generally follows this principle.

Using multithreading to enable concurrency in multiuser applications is less expensive than using multiprocessing. Separate processes cost more resources 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 GIL isn't such a problem for I/O extensive applications, but there is always a time where 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 often a hybrid of multiprocessing and multithreading—multiple workers (processes) running with multiple threads. Fortunately, many of the WSGI-compliant web servers allow such setup.

In the next section, we will take a look at an example of a threaded application.

An example of a threaded application

To see how Python threading works in practice, let's construct some example applications that can take some benefit from implementing multithreading. We will discuss a simple problem that you may encounter from time to time in your professional practice making multiple parallel HTTP queries. This problem was already mentioned as a common use case for multithreading.

Let's say we need to fetch data from some web service using multiple queries that cannot be batched into a single big HTTP request. As a realistic example, we will use foreign exchange reference rates endpoint from *Foreign exchange rates API*, available at `https://exchangeratesapi.io/`. The reasons for that choice are as follows:

- This service is open and does not require any authentication keys.
- The API of that service is very simple and can be easily queried using popular the `requests` library.
- Code for this API is open sourced and written in Python. So, in case the official service goes down, you should be able to download its source code from the official repository on GitHub, available at `https://github.com/exchangeratesapi/exchangeratesapi`.

In our examples, we will try to exchange rates for selected currencies using multiple currencies as reference points. We will then present 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 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.exchangeratesapi.io/latest?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'}
```

Since our goal is to show how a multithreaded solution of concurrent problems compares to a standard 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:

```
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.exchangeratesapi.io/latest?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))
```

Around the execution of the `main()` function, we added a few statements that are intended to measure how much time it took to finish the job. On my computer, it sometimes takes even more than 1 second to complete the following task:

```
$ python3 synchronous.py
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
time elapsed: 1.13s
```

Every run of our script will always take different times because it mostly depends on a remote service that's accessible through a network connection. So, there area lots of non-deterministic factors affecting the final result. The best approach would be to make longer tests, repeat them multiple times, and also calculate some 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.

In the next section, we will discuss the use of one thread per item.

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 place to geocode 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. And it is not a way to write reliable software that will serve thousands or millions of users. It has some serious issues that we will have to address later. But hey, it works, as we can see from the following code:

```
$ python3 threads_one_per_item.py
1 CZK =    0.044 USD,    0.039 EUR,    0.167 PLN,    0.375 NOK,      1.0 CZK
1 NOK =    0.117 USD,    0.104 EUR,    0.446 PLN,      1.0 NOK,     2.66 CZK
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
time elapsed: 0.13s
```

And it is also considerably faster.

So, when we know that threads have a beneficial effect on our application, it is time to use them in a saner 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. It is mainly due to the way the standard output is buffered in Python. You can experience malformed output when multiple calls to this function interleave 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 the 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.

Using a thread pool

The first issue we will try to solve is the unbound limit 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 workers 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 mentioned in the previous section.

So, the general idea is to start some 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 return and we will be able to exit from the program. A good candidate for our structure to be used to communicate with the workers 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
import time
from queue import Queue, Empty
from threading import Thread

import requests

THREAD_POOL_SIZE = 4

SYMBOLS = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')
BASES = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')

def fetch_rates(base):
    response = requests.get(
        f"https://api.exchangeratesapi.io/latest?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(block=False)
        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))
```

The following result of running a modified version of our program is similar to the previous one:

```
$ python3 threads_thread_pool.py
1 EUR =    1.13 USD,    1.0 EUR,    4.29 PLN,    9.62 NOK,    25.6 CZK
1 NOK =   0.117 USD,   0.104 EUR,   0.446 PLN,    1.0 NOK,    2.66 CZK
1 USD =    1.0 USD,    0.887 EUR,    3.8 PLN,    8.53 NOK,    22.7 CZK
1 PLN =   0.263 USD,   0.233 EUR,    1.0 PLN,    2.24 NOK,    5.98 CZK
1 CZK =   0.044 USD,   0.039 EUR,   0.167 PLN,   0.375 NOK,    1.0 CZK

time elapsed: 0.17s
```

The overall runtime may be slower than in situations with 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 better resource/time balance.

We will look at how to use two-way queues in the next section.

Using two-way queues

The other 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.exchangeratesapi.io/latest?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(block=False)
        except Empty:
            break
        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))
```

This eliminates the risk of malformed inputs that we could experience if the
`present_result()` function would do more `print()` statements or perform some
additional computation. 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. Here is our final output:

```
$ python3 threads_two_way_queues.py
1 USD =     1.0 USD,   0.887 EUR,     3.8 PLN,    8.53 NOK,    22.7 CZK
1 PLN =   0.263 USD,   0.233 EUR,     1.0 PLN,    2.24 NOK,    5.98 CZK
1 EUR =    1.13 USD,     1.0 EUR,    4.29 PLN,    9.62 NOK,    25.6 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

time elapsed: 0.17s
```

Dealing with errors and rate limiting is explained in the next section.

Dealing with errors and rate limiting

The last of the issues mentioned earlier that you may experience when dealing with such types of problems are rate limits that have been 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. Also, it isn't fair to abuse a service that is provided to users completely for free.

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 (this is known as a denial of service attack). The problem is even more serious due to the fact that we did not cover any failure scenario yet, and dealing with exceptions in multithreaded Python code is bit more complicated than usual.

The `request.raise_for_status()` function will raise an exception response and will have a status code indicating any type of error (for instance, rate limiting), and this is actually good news. This exception is 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. This means that our worker threads should gracefully handle possible exceptions and make sure that all items from the queue are processed.

Let's do some minor changes to our code in order to be prepared for any issues that may occur. In case of exceptions in the worker thread, we may put an error instance in to the `results_queue` queue and 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 in `work_queue.join()`. The main thread might then inspect 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):

```
def worker(work_queue, results_queue):
    while not work_queue.empty():
        try:
            item = work_queue.get(block=False)
        except Empty:
            break
        else:
            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)
```

When we are ready to handle exceptions, it is time to break our code. We don't want to abuse our free API and cause a denial of service. So, instead of putting a high load on that API, we will simulate a typical situation that is a result of many service throttling mechanisms. Many APIs return a `429 Too Many Requests` HTTP response when the client exceeds the allowed rate limit. So, we will update the `fetch_rates()` function to override the status code of every few responses in a way that will cause an exception. The following is the updated version of the function that simulates a HTTP error every few requests:

```
def fetch_rates(base):
    response = requests.get(
        f"https://api.exchangeratesapi.io/latest?base={base}"
    )
```

```
            if random.randint(0, 5) < 1:
                # simulate error by overriding status code
                response.status_code = 429

            response.raise_for_status()
            rates = response.json()["rates"]
            # note: same currency exchanges to itself 1:1
            rates[base] = 1.
            return base, rates
```

If you use it in your code, you should get the following similar error:

```
$ python3 threads_exceptions_and_throttling.py
1 PLN =    0.263 USD,   0.233 EUR,     1.0 PLN,    2.24 NOK,    5.98 CZK
1 EUR =     1.13 USD,     1.0 EUR,    4.29 PLN,    9.62 NOK,    25.6 CZK
1 USD =      1.0 USD,   0.887 EUR,     3.8 PLN,    8.53 NOK,    22.7 CZK
Traceback (most recent call last):
  File "threads_exceptions_and_throttling.py", line 136, in <module>
    main()
  File "threads_exceptions_and_throttling.py", line 129, in main
    raise result
  File "threads_exceptions_and_throttling.py", line 96, in worker
    result = fetch_rates(item)
  File "threads_exceptions_and_throttling.py", line 70, in fetch_rates
    response.raise_for_status()
  File "/usr/local/lib/python3.7/site-packages/requests/models.py", line
940, in raise_for_status
    raise HTTPError(http_error_msg, response=self)
requests.exceptions.HTTPError: 429 Client Error: OK for url:
https://api.exchangeratesapi.io/latest?base=NOK
```

Let's forget about our simulated failure and pretend that the preceding exception is not a result of faulty code. In such a situation, our program would be simply a bit too fast for this free service. It makes too many concurrent requests, and, in order to work correctly, we need to have a way to limit the program's pace.

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 some throttling solutions from scratch.

The algorithm we will use is sometimes called token bucket and is very simple. It includes the following functionality:

1. There is a bucket with a predefined amount of tokens.
2. Each token corresponds to a single permission to process one item of work.
3. Each time the worker asks for a single or multiple tokens (permissions), we do the following:
   - We measure how much time was spent from the last time we refilled the bucket
   - If the time difference allows for that, we refill the bucket with the amount of tokens that correspond to this time difference
   - If the amount of stored tokens is bigger or equal to the amount requested, we decrease the number of stored tokens and return that value
   - If the amount 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 never allow it to overfill. 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 quanta 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 quant 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 = 0

    def consume(self, amount=1):
        with self._consume_lock:
            now = time.time()
            # time measment is initialized on first
            # token request to avoid initial bursts
            if self.last == 0:
                self.last = now

            elapsed = now - self.last

            # make sure that quant of passed time is big
```

```
                    # enough to add new tokens
                    if int(elapsed * self.rate):
                        self.tokens += int(elapsed * self.rate)
                        self.last = now

                    # never over-fill the bucket
                    self.tokens = (
                        self.rate
                        if self.tokens > self.rate
                        else self.tokens
                    )

                    # finally dispatch tokens if available
                    if self.tokens >= amount:
                        self.tokens -= amount
                    else:
                        amount = 0

                    return amount
```

The usage of this class is very simple. Let's assume that we created only one instance of `Throttle` (for example, `Throttle(10)`) in the main thread and passed it to every worker thread as a positional argument. Using the same data structure in different threads is safe 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:

```
    def worker(work_queue, results_queue, throttle):
        while True:
            try:
                item = work_queue.get(block=False)
            except Empty:
                break
            else:
                while not throttle.consume():
                    pass

                try:
                    result = fetch_rates(item)
                except Exception as err:
                    results_queue.put(err)
                else:
                    results_queue.put(result)
                finally:
                    work_queue.task_done()
```

Let's take a look at a different concurrency model, which is explained in the next section.

Multiprocessing

Let's be honest, multithreading is challenging—we have already seen that in the previous section. It's a fact that the simplest approach to the problem required only minimal effort. But dealing with threads in a sane and safe manner required a tremendous amount of code.

We had to set up a thread pool, communication queues, gracefully handle exceptions from threads, and also care 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 only assume that this is production ready because there is a 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 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 is the fact that they do not share memory context. So, it is harder to corrupt data and introduce deadlocks 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 interprocess communication. In fact, Python provides some primitives that make communication between processes almost as easy as it is possible 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 systems (UNIX, macOS, and Linux), a fork is a system call that's exposed in Python through the `os.fork()` function, which will create a new child process. The two processes then continue the program in their own right after the forking. Here is an example script that forks itself exactly once:

```
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")
            print("PRNT: the child is pid %d" % child_pid)
            print("PRNT: all the pids i know %s" % pid_list)


    if __name__ == "__main__":
        main()
```

And here is an example of running it in a Terminal:

```
$ python3 forks.py
PRNT: hey, I am the parent
PRNT: the child is pid 21916
PRNT: all the pids i know [21915, 21915]
CHLD: hey, I am the child process
CHLD: all the pids i know [21915, 21916]
```

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, and we can see that the child process added the `21916` value while the parent duplicated its `21915` 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 memory context is copied to the child, each process deals with 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, it needs to be different, depending on the platform. The `os` module also exposes functions that allow you to spawn new processes under Windows, but eventually you will use them rarely. This is also true for `os.fork()`. Python provides the great `multiprocessing` module, which creates a high-level interface for multiprocessing.

The great advantage of this module is that it provides some of the abstractions that we had to code from scratch in the *An example of threaded application* section. It allows you to limit the amount of boilerplate code, so it improves application maintainability and reduces its 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 look at the built-in multiprocessing module in the next section.

The built-in multiprocessing module

multiprocessing 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:

```
from multiprocessing import Process
import os


def work(identifier):
    print(
        'hey, i am a process {}, pid: {}'
        ''.format(identifier, 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 preceding script, when executed, gives the following result:

```
$ python3 processing.py
hey, i am a process 1, pid: 9196
hey, i am a process 0, pid: 8356
hey, i am a process 3, pid: 9524
hey, i am a process 2, pid: 3456
hey, i am a process 4, pid: 6576
```

When processes are created, the memory is forked (on POSIX systems). The most efficient usage of processes is to let them work on their own after they have been created to avoid overhead, and check on their states from the parent process. 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.

The communication between process modules 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 clone of `queue.Queue` that was used earlier for the 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 multiple process environments, rather than with multiple threads, so it uses different internal transports and locking primitives. We already saw how to use Queue with multithreading in the *An example of threaded application* 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 allows you to send any pickable object (using the `pickle` module) instead of just raw bytes.

This allows for a lot easier communication between processes because you can send almost any basic Python type, as follows:

```python
from multiprocessing import Process, Pipe


class CustomClass:
    pass


def work(connection):
    while True:
        instance = connection.recv()

        if instance:
            print("CHLD: {}".format(instance))

        else:
            return


def main():
    parent_conn, child_conn = Pipe()

    child = Process(target=work, 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:

```
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 data corruption, you 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 known from multithreading.

Let's a take a look at how to use process pools in the next section.

Using process pools

Using multiple processes instead of threads adds some substantial 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 is even more of a problematic issue than it is in multithreaded applications.

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 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:

```
import time
from multiprocessing import Pool

import requests


SYMBOLS = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')
BASES = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')

POOL_SIZE = 4
```

```python
def fetch_rates(base):
    response = requests.get(
        f"https://api.exchangeratesapi.io/latest?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 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, the code is now a lot shorter. This means that it is now easier to maintain and debug in case of issues. Actually, there are now only two lines of code that explicitly deal with multiprocessing. This is a great improvement over the situation where we had to build the processing pool from scratch. Now, we don't even need to care about communication channels because they are created implicitly inside of the `Pool` class implementation.

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, are great advantages over the simple tools provided in the `threading` module. But it 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 is 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 that 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 examples. 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)
```

Let's take a look at asynchronous programming in the next section.

Asynchronous programming

Asynchronous programming gained a lot of traction in the last few years. In Python 3.5, it finally got some syntax features that solidify concepts of asynchronous execution. But this does not mean that asynchronous programming is possible only starting from Python's 3.5 version. 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 (see `Chapter 1`, *Current Status of Python*) that concentrated on this single programming approach. Some of these solutions, such as Twisted, Tornado, and Eventlet, still have huge and active communities and are really worth knowing. Anyway, starting from Python 3.5, asynchronous programming is easier than ever before. Therefore, it is expected that its built-in asynchronous features will replace the bigger part of the older tools, or external projects will gradually transform to kind of high-level frameworks based on Python built-ins.

When trying to explain what asynchronous programming is, the easiest way is to think about this approach as something similar to threads, but without system scheduling involved. This means that an asynchronous program can concurrently process problems, but its 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. Most of the solutions use a different kind of concept and, depending on the implementation, it is 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)

These are mainly the same concepts, but often implemented in a slightly different way.

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 the core of asynchronous programming. In this style of computer multitasking, it's not a responsibility of the operating system to initiate a context switch (to another process or thread), but instead every process voluntarily releases the control when it is idle to enable simultaneous execution of multiple programs. This is why it is called *cooperative*. All processes need to cooperate in order to multitask smoothly.

This model of multitasking was sometimes employed in the operating systems, but now it is hardly found as a system-level solution. This is because there is a hazard that one poorly designed service can easily break the whole system's stability. Thread and process scheduling with context switches managed directly by the operating system is now the dominant approach for concurrency on the system level. But cooperative multitasking is still a great concurrency tool on the application level.

When doing cooperative multitasking on 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 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 operation is always related with 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 somehow similar in behavior to how multithreading is implemented in Python. We know that 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 operating system can preempt the currently running thread and give control to the other one at any point of time. In asynchronous programming, tasks are never preempted by the main event loop. This is why this style of multitasking is also called **non-preemptive multitasking**.

Of course, every Python application runs on some operating system where there are other processes competing for resources. This means that the operating system always has the right to preempt the whole process and give control to another one. 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.

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 (refer to `Chapter 3`, *Modern Syntax Elements - Below the Class Level*). In fact, generators need to be used in the older versions of Python whenever you want to implement coroutines. Here is an example of function declaration that uses the `async` keyword:

```
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, for example:

```
>>> 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, as follows:

```
>>> 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 really 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 another object to wait for using the `asyncio.wait()` function. 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

async def print_number(number):
    print(number)


if __name__ == "__main__":
    loop = asyncio.get_event_loop()

    loop.run_until_complete(
        asyncio.wait([
            print_number(number)
            for number in range(10)
        ])
    )
    loop.close()
```

The `asyncio.wait()` function accepts a list of coroutine objects and returns immediately. The result is a generator that yields objects representing future results (so-called **futures**). As the name suggests, it is used to wait until all of the provided coroutines complete. The reason why it returns a generator instead of a coroutine object is backwards compatibility with previous versions of Python, which will be explained later in the *asyncio in the older version of Python* section. The result of running this script may be as follows:

```
$ python asyncprint.py
0
7
8
3
9
4
1
5
2
6
```

As we can see, the numbers are not printed in the same order as the ones we created for our coroutines. But this is exactly what we wanted to achieve.

The second important keyword that was added in Python 3.5 was `await`. It is used to wait for results of coroutines or a future (explained later), and release the control over execution to the event loop. To better understand how it works, we need to review a more complex example of code.

Let's say we want to create the following two coroutines that will perform some 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 has some concurrency issues that we will later try to improve with the additional `await` usage:

```python
import time
import random
import asyncio


async def waiter(name):
    for _ in range(4):
        time_to_sleep = random.randint(1, 3) / 4
        time.sleep(time_to_sleep)
        print(
            "{} waited {} seconds"
            "".format(name, time_to_sleep)
        )


async def main():
    await asyncio.wait([waiter("first"), waiter("second")])


if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
    loop.close()
```

When executed in the Terminal (with `time` command to measure time), it might give the following output:

```
$ time python corowait.py
second waited 0.25 seconds
second waited 0.25 seconds
second waited 0.5 seconds
second waited 0.5 seconds
first waited 0.75 seconds
```

```
first waited 0.75 seconds
first waited 0.25 seconds
first waited 0.25 seconds
real   0m3.734s
user   0m0.153s
sys    0m0.028s
```

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 that 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. We have already used this statement in the first version of the `main()` function, but it was only to improve the clarity of the code. It clearly did not make our implementation more concurrent. Let's see the following improved version of the `waiter()` coroutine that uses `await asyncio.sleep()`:

```python
async def waiter(name):
    for _ in range(4):
        time_to_sleep = random.randint(1, 3) / 4
        await asyncio.sleep(time_to_sleep)
        print(
            "{} waited {} seconds"
            "".format(name, time_to_sleep)
        )
```

If we run the updated script, we can see how the output of two functions interleave with each other:

```
$ time python corowait_improved.py
second waited 0.25 seconds
first waited 0.25 seconds
second waited 0.25 seconds
first waited 0.5 seconds
first waited 0.25 seconds
second waited 0.75 seconds
first waited 0.25 seconds
second waited 0.5 seconds
real   0m1.953s
user   0m0.149s
sys    0m0.026s
```

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.

In the next section, we will take a look at asyncio in older versions of Python.

asyncio in older versions of Python

The `asyncio` module appeared in Python 3.4, so it is the only version of Python that has serious support for asynchronous programming that is older than Python 3.5. Unfortunately, it seems that these two subsequent versions are just enough to introduce compatibility concerns.

Like it or not, the core of asynchronous programming in Python was introduced earlier than syntax elements supporting this pattern. Better late than never, but this created a situation where there are two syntaxes available for working with coroutines.

Starting from Python 3.5, you can use `async` and `await` as follows:

```
async def main():
    await asyncio.sleep(0)
```

But for Python 3.4, you need to use the `asyncio.coroutine` decorator and the `yield from` statement as follows:

```
@asyncio.couroutine
def main():
    yield from asyncio.sleep(0)
```

The other useful fact is that the `yield from` statement was introduced in Python 3.3 and there is an `asyncio` backport available on PyPI. This means that you can use this implementation of cooperative multitasking with Python 3.3, too.

Let's take a look at a 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 simple printing of sequences or asynchronous waiting.

For the sake of consistency, we will try to handle the same problem we solved previously with the help of multithreading and multiprocessing. So, we will try to asynchronously fetch some data about current currency exchange rates from an external resource through the network connection. It would be great if we could use the same requests library as in the previous sections. Unfortunately, we can't. Or to be precise, we can't do that effectively.

Unfortunately, requests do 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 (`grequests`, refer to https://github.com/kennethreitz/grequests) or thread/process pool execution (`requests-futures`, refer to https://github.com/ross/requests-futures). Neither of these solve our problem.

Knowing the limitation of the library that was so easy to use in the 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 some natively asynchronous HTTP library for that job. The standard library of Python in version 3.7 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 that's 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.exchangeratesapi.io/latest?base={base}"
    ) as response:
        rates = (await response.json())['rates']
        rates[base] = 1.

        return base, rates
```

Let's assume that this code is stored in a module named `asyncrates` that we are going to use later. Now, we are ready to rewrite the example used we discussed multithreading and multiprocessing. Previously, we used to split the whole operation into the following two separate steps:

1. Perform all requests to an external service in parallel using the `fetch_place()` function.
2. Display all the results in a loop using the `present_result()` function.

But because cooperative multitasking is something completely different from using multiple processes or threads, we can slightly modify our approach. Most of the issues raised in the *Using one thread per item* section are no longer our concern. Coroutines are non-preemptive, so we can easily display results immediately after HTTP responses are awaited. This will simplify our code and make it clearer as follows:

```
import asyncio
import time

import aiohttp

from asyncrates import get_rates

SYMBOLS = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')
BASES = ('USD', 'EUR', 'PLN', 'NOK', 'CZK')


async def fetch_rates(session, place):
    return await get_rates(session, place)


async def present_result(result):
    base, rates = (await result)

    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:
        await asyncio.wait([
            present_result(fetch_rates(session, base))
            for base in BASES
        ])
```

```
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))
```

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, migration of the 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 the start. In most cases, this will be something similar to the problem mentioned in the *A practical example of asynchronous programming* section—incompatible interfaces and 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 a kind of *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 filesystem or 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 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 operating system.

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 should you 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 workaround, I mean multithreading or multiprocessing.

This may not sound nice, but sometimes the best solution may be the one that we tried to escape from. Parallel processing of CPU-extensive tasks in Python is always done better with multiprocessing. And multithreading may deal with I/O operations equally as well (fast and without 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 look at executors and futures in the next section.

Executors and futures

Before we see how to inject threads or processes in to an asynchronous event loop, we will take a closer look at the `concurrent.futures` module that 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. It 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 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 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 like the following:

```
>>> 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
```

If you want to use the `Executor.map()` method, it does not differ in usage from the `Pool.map()` method of the pool class from the multiprocessing module, as follows:

```
def main():
    with ThreadPoolExecutor(POOL_SIZE) as pool:
        results = pool.map(fetch_rates, BASES)

        for result in results:
            present_result(*result)
```

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 `BaseEventLoop.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 were a coroutine, and it will not block, no matter how long it takes 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 need to even 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 3.7, 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.exchangeratesapi.io/latest?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*.

Summary

It was a long journey, but we successfully struggled through most of the basic 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 base threads with `threading`. 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 or like more. 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 realistic scenarios, you may find yourself using the whole arsenal of concurrency tools and styles in a single application, and this is not uncommon.

The preceding conclusion is a great introduction to the topic of the next chapter, `Chapter 17`, *Useful Design Patterns*. This is because there is no single pattern that will solve all of your problems. You should know as many as possible, because eventually you will end up using all of them on a daily basis.

In next chapter, we will take a look at a topic somehow related to concurrency: event-driven and signal programming. In that chapter, we will be concentrating on various communication patterns that are the backbone of distributed asynchronous and highly concurrent systems.

5
Section 5: Technical Architecture

In this section, we will explore various architectural patterns and paradigms that aim to make software architecture simple and sustainable. The reader will learn various ways to decouple even large and complex systems. They will become familiar with the most common design patterns used by Python developers.

The following chapters are included in this section:

- Chapter 16, *Event-Driven and Signal Programming*
- Chapter 17, *Useful Design Patterns*

16
Event-Driven and Signal Programming

In the previous chapter, we discussed various concurrency implementation models 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 aren't things that **happen**. Events in programming are just 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 have learned what the most common techniques of event-driven programming are in Python and how to extrapolate these techniques to event-driven architectures. You'll be also able to easily identify problems that can be modeled using event-driven programming.

Technical requirements

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

- `flask`
- `blinker`

To run `kinter` 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. This package is usually named `python3-tk`.

You can install these packages using the following command:

```
python3 -m pip install <package-name>
```

The code files for this chapter can be found at `https://github.com/PacktPublishing/Expert-Python-Programming-Third-Edition/tree/master/chapter16`.

What exactly is event-driven programming?

Event-driven programming concentrates on the events (messages) and their flow between different software components. If you think about it longer, you'll notice that the notion of events 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 graphical user interfaces. Everywhere the program needs to wait for some human input, that input can be modeled as events or messages. In such framing, an event-driven program is just a collection of event or message handlers that react to human interaction.

Events also 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. Such requests are, of course, often 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's notice.

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 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 for asynchronous systems 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 that 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 others, 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`
- `UPDATE`
- `DELETE`

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` alternative 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 both 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. Whether triggers coming from different sessions are independent and whether they can be processed asynchronously depends on multiple factors (transaction or not, isolation level, and many more), and is really up to the database system. But it doesn't mean that the system as a whole cannot be modeled as if the events were truly independent.

In the next section, we'll take a look at event-driven programming in GUIs.

Event-driven programming in GUIs

Graphical user interfaces (**GUIs**) are what most people think of when they hear the term event-driven programming. Event-driven programming is an elegant way of coupling user input to code in graphical user interfaces 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 a complex interfaces model, this interaction is 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 Qt library). Still, the general pattern is always the same – every interface component (often called **widgets**) can emit events upon interaction, and these events can be either subscribed to by other components or can be directly attached to event handlers. Depending on the GUI library, events can just be plain named signals stating that something happened (for example, widget *A* is clicked), or be more complex messages containing additional information about interaction context (for example, a keystroke is pressed or the position of the mouse cursor).

We will discuss the differences of actual design patterns later in the *Various styles of event-driven programming* section. Let's take a look at the example Python GUI application that can be created with the use of the built-in `tkinter` module:

```python
import this
from tkinter import *
from tkinter import messagebox

rot13 = str.maketrans(
    "ABCDEFGHIJKLMabcdefghijklmNOPQRSTUVWXYZnopqrstuvwxyz",
    "NOPQRSTUVWXYZnopqrstuvwxyzABCDEFGHIJKLMabcdefghijklm"
)


def main_window(root):
    frame = Frame(root, width=100, height=100)
    zen_button = Button(root, text="Python Zen", command=show_zen)
    zen_button.pack()


def show_zen():
    messagebox.showinfo(
        "Zen of Python",
        this.s.translate(rot13)
    )


if __name__ == "__main__":
    root = Tk()
    main_window(root)
    root.mainloop()
```

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 preceding 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. Although `tkinter` allows specific input events (key presses or mouse button clicks) on widgets to be bound to specific callbacks using the `bind()` method, this is not always useful. Instead of the `bind()` method, we use the `command` argument on the `Button` widget. This will translate specific raw input events (mouse press and release) into a proper function callback while maintaining common interface usability conventions (for example, firing an action only when the mouse is released over the button). Most of the GUI frameworks work in a similar manner – you rarely work with raw keyboard and mouse events, 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, and even more so with the advent of service-oriented and microservice architectures. With event-driven programming, it is easier to split complex systems into isolated components that have a limited set of responsibilities. In service-oriented or microservice architectures, the flow of events happens not between classes or functions inside of single process, but between many networked services. In large distributed architectures, the flow of events between services is often coordinated using special communication protocols (for example, AMQP and ZeroMQ) and/or dedicated services. 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 application as event-based. 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()
```

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, a lot more structured because the HTTP protocol defines multiple types of requests (for example, `POST`, `GET`, `PUT`, and `DELETE`) and a few ways to transfer additional data (query string, request body, and headers).

Of course, in the case of a Flask application, the user does not communicate with it directly, but instead uses a web browser as their interface. But, is this difference really that big? In fact, many cross-platform user interface libraries (such as Tcl/Tk, Qt, and GTK+) are just proxies between your application and system windowing APIs. So, in both cases, we deal with communication and events flowing through different application layers. It is just that, in web applications, layers are more evident and communication is always explicit.

In the next section, we will go through the various styles of event-driven programming.

Various styles of event-driven programming

As we already stated, event-driven programming can be implemented on various levels of a software architecture using multiple different design patterns. 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 a few of the major event-driven programming styles that are the foundation for more concrete patterns.

In the next sections, we will do a brief review of three major styles of event-driven programming that you can 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 relation between event emitters and event handlers.

This style of event-based programming is the dominating pattern among GUI frameworks and libraries. The reason for that is simple – it really captures the way how both users and programmers think about user interfaces. 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, and the `show_zen()` callback (in `tkinter`, callbacks are called **commands**) does not receive any object that would encapsulate the event that invoked its call. This makes sense, because the responsibility of attaching event handlers lies closer to the event emitter (here, it is the button), and the event handler is barely concerned about the actual occurrence 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 the updated excerpt of the previous `tkinter` application that uses this style of event binding:

```
def main_window(root):
    frame = Frame(root, width=100, height=100)

    zen_button = Button(root, text="Python Zen")
    zen_button.bind("<ButtonRelease-1>", show_zen)
    zen_button.pack()


def show_zen(event):
    messagebox.showinfo(
        "Zen of Python",
        this.s.translate(rot13)
    )
```

In the preceding example, the event is no longer implicit, so the `show_zen()` callback must be able to accept event object. It 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 (here, `<ButtonRelease-1>`) from one object (here, `zen_button`) can be bound to only one callback (here, `show_zen()`). 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 new 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 specialized to serve a single emitter and cannot be bound to objects of a different type.

Let's take a look at subject-based style 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 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 (event emitter). The most common emanation of that style is the `Observer` design pattern. We will discuss the `Observer` design pattern in detail in `Chapter 17`, *Useful Design Patterns*, but it is so important that we can't discuss subject-based event programming without introducing it here. So, we will take a sneak peek now just to see how it compares to the callback-based event programming and will discuss the details of that pattern in the next chapter.

In short, the `Observer` design pattern consists of two classes of objects – observers and subjects (sometimes observable). `Subject` is an object that maintains a list of `Observer` that are interested in what happens to `Subject`. So, `Subject` is an event emitter and `Observer` are event handlers.

As simple dummy implementation of the `Observer` pattern could be as follows:

```
class Subject:
    def __init__(self):
        self._observers = []

    def register(self, observer):
        self._observers.append(observer)

    def _notify_observers(self, event):
        for observer in self._observers:
            observer.notify(self, event)


class Observer:
    def notify(self, subject, event):
        print(f"Received event {event} from {subject}")
```

The preceding classes are, of course, just a scaffolding. The `_notify_observers()` method is supposed to be called internally in the `Subject` class whenever something happens that could be interesting to registered observers. This can be any event, but usually subjects inform their observers about their important state changes.

Just for illustration purposes, let's assume that subjects notify all of their subscribed observers about new observers registering. Here are the updated `Observer` and `Subject` classes, which are intended to show the process of event handling:

```
import itertools


class Subject:
    _new_id = itertools.count(1)

    def __init__(self):
        self._id = next(self._new_id)
        self._observers = []

    def register(self, observer):
        self._notify_observers(f"register({observer})")
        self._observers.append(observer)

    def _notify_observers(self, event):
        for observer in self._observers:
            observer.notify(self, event)

    def __str__(self):
        return f"<{self.__class__.__name__}: {self._id}>"
```

```
class Observer:
    _new_id = itertools.count(1)

    def __init__(self):
        self._id = next(self._new_id)

    def notify(self, subject, event):
        print(f"{self}: received event '{event}' from {subject}")

    def __str__(self):
        return f"<{self.__class__.__name__}: {self._id}>"
```

If you try to instantiate and bind the preceding classes in an interactive interpreter session, you may see the following output:

```
>>> from subject_based_events import Subject
>>> subject = Subject()
>>> observer1 = Observer()
>>> observer2 = Observer()
>>> observer3 = Observer()
>>> subject.register(observer1)
>>> subject.register(observer2)
<Observer: 1>: received event 'register(<Observer: 2>)' from <Subject: 1>
>>> subject.register(observer3)
<Observer: 1>: received event 'register(<Observer: 3>)' from <Subject: 1>
<Observer: 2>: received event 'register(<Observer: 3>)' from <Subject: 1>
```

Subject-based event programming allows for multicast event handling. This type of handling is allowed for more reusable and fine-grained event handlers with visible benefits to software 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 of events. With many subjects and subscribers, this can quickly become a problem. 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 amount 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. It's mostly due to the focus on subjects that 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 one point of time, 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.

In the next section, we will take a look at topic-based style.

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 relation. Topic-based event programming is a generalization of previous styles. Event-driven applications written with that 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 often 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 scrapping 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 an 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(name)` constructor with the same `name` value will return the same signal object. It allows you to easily refer to signals at any time. The following is an example of the `SelfWatch` class, which uses named signals to make its instances notified every time their 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}>"
```

The following transcript of the interactive session shows how new instances of the `SelfWatch` class notify the siblings about their initialization:

```
>>> 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 it as a module or class attribute, you type in string literals or accidental signal naming 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 are communicating 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 passing 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 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 be also dynamically adjusted to the number of events being processed in the system currently.
- **Loose coupling**: Systems that are composed of many (preferably small) services communicating over queues tend to be more loosely coupled than monolithic software. 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 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 GIL, which was discussed in `Chapter 15`, *Concurrency*).

Last, but not least, event-driven architectures seem to have a natural affinity to serverless computing. In this cloud-computing execution model, you're not concerned about infrastructure and 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 event and message queues.

Event and message queues

Most single-process implementations of event-driven programming are handled as soon as they appear in a serial fashion. Whether it is a callback-based style 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 whenever one of these events happens.

This style of information passing in distributed applications is usually realized through a request-response communication. A 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 new event connection, more than one service often 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 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 about the messages and their intended delivery mechanism. We've already mentioned a practical usage example of message queues in the *Using task queues and delayed processing* section of `Chapter 14`, *Optimization – Some Powerful Techniques*.

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 information passing 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 message queues are already a great way of inter-thread communication that allows you to avoid locking (see `Chapter 15`, *Concurrency*).

Besides loose coupling, message queues (especially in the form of dedicated services) have many additional capabilities:

- Most message queues are able to provide message persistence. This means that, even if message queues service dies, no messages will be lost.
- 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.
- Message queues are naturally concurrent. With various message distribution semantics (for example, fan-out and round-robin) it is a great basis of 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 example 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 a programming library. The leading and most popular brokerless messaging library is **ZeroMQ** (often spelled as **ØMQ**). The biggest advantage of brokerless messaging is elasticity. They trade operational simplicity (no additional centralized service or cluster of services to maintain) for feature completeness (things like persistence and complex message delivery needs to be implemented inside of services).

Both types of messaging approaches have advantages and disadvantages. In brokered message queues, there is always an additional service to maintain (in case of open source queues running on their own infrastructure) or additional entry on your cloud provider invoice (in case of cloud-based services). Such messaging systems quickly became a critical part of your architecture. If such service stops working, all your systems stop as well because of inter-service communication. What you get in return are usually systems where everything is available out-of-the-box and only a matter of proper configuration or a few API calls.

With brokerless messaging, your communication is often more distributed. What, in code, appears to be a simple event publication to some abstract channel is often just code-level abstraction for peer-to-peer communication that happens under the hood of the brokerless messaging library. 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 with each other. The downside of this approach is that you're usually on your own when it comes to things like message persistency and delivery/processing confirmations or delivery retries. If you have such needs, you will either have to implement such capabilities directly in your services or build your own messaging broker using brokerless messaging libraries.

Summary

In this chapter, we discussed the elements of event-driven programming. We started from 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 that is 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.

As we are nearing the end of this book, you have probably noticed that the farther we go, the less we actually speak about Python. In this chapter, we have discussed elements of event-driven and signal programming, but we have barely talked about Python itself. We have, of course, discussed some examples of Python code, but these were just to illustrate concepts of the event-driven programming paradigm rather than make you better at Python.

This is because Python – like any other programming language – is just a tool. You need to know your tool well to be a good programmer, but you won't be a good programmer just by knowing the language and its libraries. That's why we have started from craftsmanship topics such as modern syntax features, best packaging practices, and deployment strategies, and then steadily reached topics that are more and more abstract and language agnostic.

The next chapter will be in a similar tone, as we will discuss some useful design patterns. We will present a lot more code this time, but it won't matter that much, as design patterns are language agnostic by definition. This **language independence** is what makes design patterns one of the most common topics found in programming books. Still, I hope our take on them won't be a boring read, as we will try to concentrate only on patterns that are relevant to Python developers.

17
Useful Design Patterns

A **design pattern** is a reusable, somewhat language-specific solution to a common problem in software design. The most popular book on this topic is *Design Patterns: Elements of Reusable Object-Oriented Software*, *Addison-Wesley Professional*, written by Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm, also known as the **Gang of Four** (**GoF**). It is considered a major writing in this area and provides a catalog of 23 design patterns with examples in Smalltalk and C++.

While designing an application's code, these patterns help solve common problems. They ring a bell to all developers, since they describe proven development paradigms. But they should be studied with the used language in mind, since some of them do not make sense in some languages or are already built in.

This chapter describes the most useful patterns in Python, and patterns that are interesting to discuss, with implementation examples. The following are the three sections that correspond to design pattern categories defined by the GoF:

- **Creational patterns**: These are patterns that are used to generate objects with specific behaviors.
- **Structural patterns**: These are patterns that help structure the code for specific use cases.
- **Behavioral patterns**: These are patterns that help assign responsibilities and encapsulate behaviors.

In this chapter, you will learn what the most common designs are and how to implement them in Python. You will also learn to recognize problems that can successfully be solved using these patterns in order to improve your application architecture and overall software maintainability.

Technical requirements

The code files for this chapter can be found
at `https://github.com/PacktPublishing/Expert-Python-Programming-Third-Edition/tree/master/chapter17`.

Creational patterns

Creational patterns deal with the object instantiation mechanism. Such a pattern might define a way for object instances to be created or even how classes are constructed.

These are very important patterns in compiled languages such as C or C++, since it is harder to generate types on demand at runtime.

But creating new types at runtime is pretty straightforward in Python. The built-in `type` function lets you define a new type object by code:

```
>>> MyType = type('MyType', (object,), {'a': 1})
>>> ob = MyType()
>>> type(ob)
<class '__main__.MyType'>
>>> ob.a
1
>>> isinstance(ob, object)
True
```

Classes and types are built-in factories. We have already dealt with the creation of new class objects, and you can interact with class and object generation using **metaclasses**. These features are the basics to implement the **factory** design pattern, but we won't describe it further in this section because we extensively covered the topic of class and object creation in `Chapter 3`, *Modern Syntax Elements - Below the Class Level*.

Besides factory, the only other creational design pattern from the GoF that is interesting to describe in Python is singleton.

Singleton

Singleton restricts the instantiation of a class to only a single object instance.

The singleton pattern makes sure that a given class has always only one living instance in the application. This can be used, for example, when you want to restrict resource access to one and only one memory context in the process. For instance, a database connector class can be a singleton that deals with synchronization and manages its data in memory. It makes the assumption that no other instance is interacting with the database in the meantime.

This pattern can simplify a lot of the way that concurrency is handled in an application. Utilities that provide application-wide functions are often declared singletons. For instance, in web applications, a class that is in charge of reserving a unique document ID would benefit from the singleton pattern. There should be one and only one utility doing this job.

There is a popular semi-idiom to create singletons in Python by overriding the `__new__()` method of a class:

```
class Singleton:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if cls._instance is None:
            cls._instance = super().__new__(cls, *args, **kwargs)

        return cls._instance
```

If you try to create multiple instances of that class and compare their IDs, you will find that they all represent the same object:

```
>>> instance_a = Singleton()
>>> instance_b = Singleton()
>>> id(instance_a) == id(instance_b)
True
>>> instance_a == instance_b
True
```

I call this a semi-idiom, because it is a really dangerous pattern. The problem starts when you try to subclass your base singleton class and create an instance of this new subclass, if you already created an instance of the base class:

```
>>> class ConcreteClass(Singleton): pass
...
>>> Singleton()
<Singleton object at 0x000000000306B470>
>>> ConcreteClass()
<Singleton object at 0x000000000306B470>
```

This may become even more problematic when you notice that this behavior is affected by an instance creation order. Depending on your class usage order, you may or may not get the same result. Let's see what the results are if you first create the subclass instance, and only then, the instance of the base class:

```
>>> class ConcreteClass(Singleton): pass
...
>>> ConcreteClass()
<ConcreteClass object at 0x00000000030615F8>
>>> Singleton()
<Singleton object at 0x000000000304BCF8>
```

As you can see, the behavior is completely different and very hard to predict. In large applications, it may lead to very dangerous and hard-to-debug problems. Depending on the runtime context, you may or may not use the classes that you meant to. Because such behavior is really hard to predict and control, the application may break because of changed import order or even user input. If your singleton is not meant to be subclassed, it may be relatively safe to implement that way. Anyway, it's a ticking bomb. Everything may blow up if someone disregards the risk in the future and decides to create a subclass from your singleton object. It is safer to avoid this particular implementation and use an alternative one.

It is a lot safer to use a more advanced technique: **metaclasses**. By overriding the __call__() method of a metaclass, you can affect the creation of your custom classes. This allows the creation of a reusable singleton code:

```
class Singleton(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super().__call__(*args, **kwargs)
        return cls._instances[cls]
```

By using this `Singleton` as a metaclass for your custom classes, you are able to get singletons that are safe to subclass and independent of instance creation order:

```
>>> ConcreteClass() == ConcreteClass()
True
>>> ConcreteSubclass() == ConcreteSubclass()
True
>>> ConcreteClass()
<ConcreteClass object at 0x000000000307AF98>
>>> ConcreteSubclass()
<ConcreteSubclass object at 0x000000000307A3C8>
```

Another way to overcome the problem of trivial singleton implementation is to use what Alex Martelli proposed. He came up with something similar in behavior to a singleton, but completely different in structure. This is not a classical design pattern coming from the GoF book, but seems to be common among Python developers. It is called `Borg` or `Monostate`.

The idea is quite simple. What really matters in the singleton pattern is not the number of living instances a class has, but rather the fact that they all share the same state at all times. So, Alex Martelli came up with a class that makes all instances of the class share the same: `__dict__`:

```
class Borg(object):
    _state = {}

    def __new__(cls, *args, **kwargs):
        ob = super().__new__(cls, *args, **kwargs)
        ob.__dict__ = cls._state
        return ob
```

This fixes the subclassing issue but is still dependent on how the subclass code works. For instance, if `__getattr__` is overridden, the pattern can be broken.

Nevertheless, singletons should not have several levels of inheritance. A class that is marked as a singleton is already specified.

That said, this pattern is considered by many developers as a heavy way to deal with uniqueness in an application. If a singleton is needed, why not use a module with functions instead, since a Python module is already a singleton? The most common pattern is to define a module-level variable as an instance of a class that needs to be a singleton. This way, you also don't constrain the developers to your initial design.

> The singleton factory is an *implicit* way of dealing with the uniqueness in your application. You can live without it. Unless you are working in a framework à la Java that requires such a pattern, use a module instead of a class.

Let's take a look at structural patterns in the next section.

Structural patterns

Structural patterns are really important in big applications. They decide how the code is organized and give developers recipes on how to interact with each part of the application.

For a long time, the most well-known implementation of many structural patterns in the Python world provided the Zope project with its **Zope Component Architecture** (**ZCA**). It implements most of the patterns described in this section and provides a rich set of tools to work with them. The ZCA is intended to run not only in the Zope framework, but also in other frameworks such as Twisted. It provides an implementation of interfaces and adapters among other things. Unfortunately (or not), Zope lost almost all of its momentum and is not as popular as it used to be. But its ZCA may still be a good reference on implementing structural patterns in Python. Baiju Muthukadan wrote a *Comprehensive Guide to Zope Component Architecture*. It is available both in print and free online (refer to `http://muthukadan.net/docs/zca.html`). It was written in 2009, so it does not cover the latest versions of Python but should still be a good read because it provides a lot of rationale for some of the mentioned patterns.

Python already provides some of the popular structural patterns through its syntax. For instance, the class and function decorators can be considered flavors of the **decorator pattern**. Also, support for creating and importing modules is an emanation of the **module pattern**.

The list of common structural patterns is actually quite long. The original *Design Patterns* book featured as many as seven of them and the list was later extended by other literature. We won't discuss all of them but will focus only on the three most popular and recognized ones, which are these:

- Adapter
- Proxy
- Facade

Let's examine these structural patterns in the next sections.

Adapter

The **adapter** pattern allows the interface of an existing class to be used from another interface. In other words, an adapter wraps a class or an object A so that it works in a context intended for a class or an object B.

Creating adapters in Python is actually very straightforward due to how typing in this language works. The typing philosophy in Python is commonly refererred to as **duck typing**:

> *"If it walks like a duck and talks like a duck, then it's a duck!"*

According to this rule, if the value for a function or method is accepted, the decision should not be based on its type but rather on its interface. So, as long as the object behaves as expected, that is, has proper method signatures and attributes, its type is considered compatible. This is completely different than many statically typed languages, where such a thing is rarely available.

In practice, when some code is intended to work with a given class, it is fine to feed it with objects from another class, as long as they provide the methods and attributes used by the code. Of course, this assumes that the code isn't calling `instance` to verify that the instance is of a specific class.

The adapter pattern is based on this philosophy and defines a wrapping mechanism, where a class or an object is wrapped in order to make it work in a context that was not primarily intended for it. `StringIO` is a typical example, as it adapts the `str` type, so it can be used as a `file` type:

```
>>> from io import StringIO
>>> my_file = StringIO('some content')
>>> my_file.read()
'some content'
>>> my_file.seek(0)
>>> my_file.read(1)
's'
```

Let's take another example. A `DublinCoreInfos` class knows how to display the summary of a subset of Dublin Core information (see `http://dublincore.org/`) for a given document provided as `dict`. It reads a few fields such as the author's name or the title, and prints them. To be able to display Dublin Core for a file, it has to be adapted the same way `StringIO` does. The following figure shows an UML-like diagram for such a kind of adapter pattern implementation:
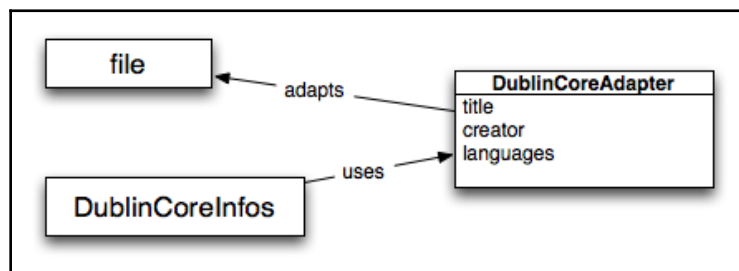
Figure 1: A UML diagram for a simple adapter pattern example

`DublinCoreAdapter` wraps a `file` instance and provides metadata access over it:

```
from os.path import split, splitext class DublinCoreAdapter: def
__init__(self, filename): self._filename = filename @property def
title(self): return splitext(split(self._filename)[-1])[0] @property def
languages(self): return ('en',) def __getitem__(self, item): return
getattr(self, item, 'Unknown') class DublinCoreInfo(object): def
summary(self, dc_dict): print('Title: %s' % dc_dict['title'])
print('Creator: %s' % dc_dict['creator']) print('Languages: %s' % ',
'.join(dc_dict['languages']))
```

And here is the example usage:

```
>>> adapted = DublinCoreAdapter('example.txt')
>>> infos = DublinCoreInfo()
>>> infos.summary(adapted)
Title: example
Creator: Unknown
Languages: en
```

Besides the fact that it allows substitution, the adapter pattern can also change the way developers work. Adapting an object to work in a specific context means making the assumption that the class of the object does not matter at all. What matters is that this class implements what `DublinCoreInfo` is waiting for, and this behavior is fixed or completed by an adapter. So, the code can simply tell somehow whether it is compatible with objects that are implementing a specific behavior. This can be expressed by *interfaces*, and we will take a look at it in the next section.

Interfaces

An **interface** is a definition of an API. It describes a list of methods and attributes that a class should have to implement with the desired behavior. This description does not implement any code, but just defines an explicit contract for any class that wishes to implement the interface. Any class can then implement one or several interfaces in whichever way it wants.

While Python prefers duck typing over explicit interface definitions, it may be better to use the latter sometimes. For instance, an explicit interface definition makes it easier for a framework to define functionalities over interfaces.

The benefit is that classes are loosely coupled, which is considered as a good practice. For example, to perform a given process, class A does not depend on class B, but rather on an interface I. Class B implements I, but it could be any other class.

The support for such a technique is built in in many statically typed languages such as Java or Go. The interfaces allow the functions or methods to limit the range of acceptable parameter objects that implement a given interface, no matter what kind of class it comes from. This allows for more flexibility than restricting arguments to given types or their subclasses. It is like an explicit version of duck typing behavior—Java uses interfaces to verify a type safety at compile time, rather than using duck typing to tie things together at runtime.

Python has a completely different typing philosophy than Java, so it does not have native support for interfaces. Anyway, if you would like to have more explicit control of application interfaces, there are generally two solutions to choose from:

- Use some third-party framework that adds a notion of interfaces.
- Use some of the advanced language features to build your methodology for handling interfaces.

Let's take a look at the some of the solutions in the next sections.

Using zope.interface

There are 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, the `zope.interface` package is still one of the main components of the Twisted framework.

The core class of the `zope.interface` package is the `Interface` class. It allows you to explicitly define a new interface by subclassing. Let's assume that we want to define the obligatory interface for every implementation of a rectangle:

```python
from zope.interface import Interface, Attribute


class IRectangle(Interface):
    width = Attribute("The width of rectangle")
    height = Attribute("The height of rectangle")

    def area():
        """ Return area of rectangle
        """

    def perimeter():
        """ Return perimeter of rectangle
        """
```

Here are some important things to remember when defining interfaces with
`zope.interface`:

- The common naming convention for interfaces is to use `I` as the name prefix.
- The methods of an interface must not take the `self` parameter.
- As the interface does not provide concrete implementation, it should consist only of empty methods. You can use the `pass` statement, raise `NotImplementedError`, or provide `docstring` (preferred).
- An interface can also specify the required attributes using the `Attribute` class.

When you have such a contract defined, you can then define new concrete classes that provide an implementation for our `IRectangle` interface. In order to do that, you need to use the `implementer()` class decorator and implement all of the defined methods and attributes:

```python
@implementer(IRectangle)
class Square:
    """ Concrete implementation of square with rectangle interface
    """

    def __init__(self, size):
        self.size = size

    @property
    def width(self):
        return self.size

    @property
    def height(self):
        return self.size

    def area(self):
        return self.size ** 2

    def perimeter(self):
        return 4 * self.size


@implementer(IRectangle)
class Rectangle:
    """ Concrete implementation of rectangle
    """
    def __init__(self, width, height):
        self.width = width
        self.height = height
```

```
        def area(self):
            return self.width * self.height

        def perimeter(self):
            return self.width * 2 + self.height * 2
```

It is common to say that the interface defines a contract that a concrete implementation needs to fulfil. The main benefit of this design pattern is being able to verify consistency between contract and implementation before the object is being 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.

Since we have defined our interface and two concrete implementations, let's verify their contracts in an interactive session:

```
>>> from zope.interface.verify import verifyClass, verifyObject
>>> verifyObject(IRectangle, Square(2))
True
>>> verifyClass(IRectangle, Square)
True
>>> verifyObject(IRectangle, Rectangle(2, 2))
True
>>> verifyClass(IRectangle, Rectangle)
True
```

This is nothing impressive. The `Rectangle` and `Square` classes carefully follow the defined contract, so there is nothing more to see than a successful verification. But what happens when we make a mistake? Let's see an example of two classes that fail to provide full `IRectangle` interface implementation:

```
@implementer(IRectangle)
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
@implementer(IRectangle)
class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius ** 2

    def perimeter(self):
        return 2 * math.pi * self.radius
```

The `Point` class does not provide any method or attribute of the `IRectangle` interface, so its verification will show inconsistencies already on the class level:

```
>>> verifyClass(IRectangle, Point)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "zope/interface/verify.py", line 102, in verifyClass
    return _verify(iface, candidate, tentative, vtype='c')
  File "zope/interface/verify.py", line 62, in _verify
    raise BrokenImplementation(iface, name)
zope.interface.exceptions.BrokenImplementation: An object has failed to
implement interface <InterfaceClass __main__.IRectangle>
        The perimeter attribute was not provided.
```

The `Circle` class is a bit more problematic. It has all the interface methods defined, but breaks the contract on the `instance` attribute level. This is the reason, in most cases, that you need to use the `verifyObject()` function to completely verify the interface implementation:

```
>>> verifyObject(IRectangle, Circle(2))
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "zope/interface/verify.py", line 105, in verifyObject
    return _verify(iface, candidate, tentative, vtype='o')
  File "zope/interface/verify.py", line 62, in _verify
    raise BrokenImplementation(iface, name)
zope.interface.exceptions.BrokenImplementation: An object has failed to
implement interface <InterfaceClass __main__.IRectangle>
        The width attribute was not provided.
```

Using `zope.inteface` 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 inheritances, and also allows you to catch inconsistencies early. But the biggest downside of this approach is the requirement to explicitly define that the given class follows some interface in order to be verified. This is especially troublesome if you need to verify instances coming from the external classes of built-in libraries. `zope.interface` provides some solutions for that problem and you can, of course, handle such issues on your own by using the adapter pattern, or even monkey-patching. Anyway, the simplicity of such solutions is at least debatable.

Using function annotations and abstract base classes

Design patterns are meant to make problem-solving easier, and not to provide you with more layers of complexity. The `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 for building a lightweight alternative to the 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:

- **Abstract base classes** (**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, consider the following:

```
assert type(instance) == list
```

Comparing types in functions or methods this way completely breaks the ability to pass class subtype as an argument to the function. The 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 way of 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 abstract base classes 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

So, let's assume we want to define an interface that ensures that a class has a `push()` method. We need to create a new abstract base class using a special `ABCMeta` metaclass and an `abstractmethod()` decorator from the standard `abc` module:

```
from abc import ABCMeta, abstractmethod


class Pushable(metaclass=ABCMeta):

    @abstractmethod
    def push(self, x):
```

```
        """ Push argument no matter what it means
        """
```

The `abc` module also provides an ABC base class that can be used instead of the metaclass syntax:

```
from abc import ABCMeta, abstractmethod


class Pushable(metaclass=ABCMeta):
    @abstractmethod
    def push(self, x):
        """ Push argument no matter what it means
        """
```

Once it is done, we can use that `Pushable` class as a base class for concrete implementation and it will guard us against instantiation of objects that would have an incomplete implementation. Let's define `DummyPushable`, which implements all interface methods and `IncompletePushable` that breaks the expected contract:

```
class DummyPushable(Pushable):
    def push(self, x):
        return


class IncompletePushable(Pushable):
    pass
```

If you want to obtain the `DummyPushable` instance, there is no problem because it implements the only required `push()` method:

```
>>> DummyPushable()
<__main__.DummyPushable object at 0x10142bef0>
```

But if you try to instantiate `IncompletePushable`, you will get `TypeError` because of missing implementation of the `interface()` method:

```
>>> IncompletePushable()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class IncompletePushable with
abstract methods push
```

The preceding approach is a great way to ensure implementation completeness of base classes but is as explicit as the `zope.interface` alternative. The `DummyPushable` instances are of course also instances of `Pushable` because `Dummy` is a subclass of `Pushable`. But how about other classes with the same methods but not descendants of `Pushable`? Let's create one and see:

```
>>> class SomethingWithPush:
...     def push(self, x):
...         pass
...
>>> isinstance(SomethingWithPush(), Pushable)
False
```

Something is still missing. The `SomethingWithPush` class definitely has a compatible interface but is not considered as an instance of `Pushable` yet. So, what is missing? The answer is the `__subclasshook__(subclass)` method that 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 it by yourself, as `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. The standard procedure is to check whether the set of defined methods are available somewhere in the MRO of the given class:

```
from abc import ABCMeta, abstractmethod


class Pushable(metaclass=ABCMeta):

    @abstractmethod
    def push(self, x):
        """ Push argument no matter what it means
        """
    @classmethod
    def __subclasshook__(cls, C):
        if cls is Pushable:
            if any("push" in B.__dict__ for B in C.__mro__):
                return True
        return NotImplemented
```

With the `__subclasshook__()` method defined that way, you can now confirm that the instances that implement the interface implicitly are also considered instances of the interface:

```
>>> class SomethingWithPush:
...     def push(self, x):
...         pass
...
>>> isinstance(SomethingWithPush(), Pushable)
True
```

Unfortunately, this approach to 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 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.

The two other features that complement abstract base classes are functioning annotations and type hints. Function annotation is the syntax element described briefly in Chapter 2, *Modern Python Development Environments*. It allows you to annotate functions and their arguments with arbitrary expressions. As explained in Chapter 2, *Modern Python Development Environments*, this is only a feature stub that does not provide any syntactic meaning. There is utility in the standard library that uses this feature to enforce any behavior. Anyway, you can use it as a convenient and lightweight way to inform the developer of the expected argument interface. For instance, consider this `IRectangle` interface rewritten from `zope.interface` to abstract the base class:

```
from abc import (
    ABCMeta,
    abstractmethod,
    abstractproperty
)


class IRectangle(metaclass=ABCMeta):

    @abstractproperty
    def width(self):
        return

    @abstractproperty
    def height(self):
        return
```

```
    @abstractmethod
    def area(self):
        """ Return rectangle area
        """

    @abstractmethod
    def perimeter(self):
        """ Return rectangle perimeter
        """

    @classmethod
    def __subclasshook__(cls, C):
        if cls is IRectangle:
            if all([
                any("area" in B.__dict__ for B in C.__mro__),
                any("perimeter" in B.__dict__ for B in C.__mro__),
                any("width" in B.__dict__ for B in C.__mro__),
                any("height" in B.__dict__ for B in C.__mro__),
            ]):
                return True
        return NotImplemented
```

If you have a function that works only on rectangles, let's say `draw_rectangle()`, you could annotate the interface of the expected argument as follows:

```
def draw_rectangle(rectangle: IRectangle):
    ...
```

This adds nothing more than information to the developer about expected information. And even this is done through an informal contract because, as we know, bare annotations contain no syntactic meaning. But they are accessible at runtime, so we can do something more. Here is an example implementation of a generic decorator that is able to verify interface from function annotation if it is provided using abstract base classes:

```
def ensure_interface(function):
    signature = inspect.signature(function)
    parameters = signature.parameters

    @wraps(function)
    def wrapped(*args, **kwargs):
        bound = signature.bind(*args, **kwargs)
        for name, value in bound.arguments.items():
            annotation = parameters[name].annotation

            if not isinstance(annotation, ABCMeta):
                continue
```

```
                if not isinstance(value, annotation):
                    raise TypeError(
                        "{} does not implement {} interface"
                        "".format(value, annotation)
                    )

            function(*args, **kwargs)

        return wrapped
```

Once it is done, we can create some concrete class that implicitly implements the
`IRectangle` interface (without inheriting from `IRectangle`) and updates the
implementation of the `draw_rectangle()` function to see how the whole solution works:

```python
class ImplicitRectangle:
    def __init__(self, width, height):
        self._width = width
        self._height = height

    @property
    def width(self):
        return self._width

    @property
    def height(self):
        return self._height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return self.width * 2 + self.height * 2


@ensure_interface
def draw_rectangle(rectangle: IRectangle):
    print(
        "{} x {} rectangle drawing"
        "".format(rectangle.width, rectangle.height)
    )
```

If we feed the `draw_rectangle()` function with an incompatible object, it will now raise `TypeError` with a meaningful explanation:

```
>>> draw_rectangle('foo')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 101, in wrapped
TypeError: foo does not implement <class 'IRectangle'> interface
```

But if we use `ImplicitRectangle` or anything else that resembles the `IRectangle` interface, the function executes as it should:

```
>>> draw_rectangle(ImplicitRectangle(2, 10))
2 x 10 rectangle drawing
```

This is our example implementation of `ensure_interface()` based on the `typechecked()` decorator from the `typeannotations` project that tries to provide runtime-checking capabilities (refer to `https://github.com/ceronman/typeannotations`). Its source code might give you some interesting ideas about how to process type annotations to ensure runtime interface checking.

The last feature that can be used to complement this interface pattern landscape is type hints. Type hints are described in detail by PEP 484 and were added to the language quite recently. They are exposed in the new `typing` module and are available from Python 3.5. Type hints are built on top of function annotations and reuse this slightly forgotten syntax feature of Python 3. They are intended to guide type hinting and checking for various yet-to-come Python type checkers. The `typing` module and PEP 484 document aim to provide a standard hierarchy of types and classes that should be used for describing type annotations.

Still, type hints do not seem to be something revolutionary because this feature does not come with any type checker built in into the standard library. If you want to use type checking or enforce strict interface compatibility in your code, you'll have to integrate some third-party libraries. This is why we won't dig into the details of PEP 484. Anyway, type hints and the documents describing them are worth mentioning because if some extraordinary solution will emerge in the field of type checking in Python, it is highly probable to be based on PEP 484.

Using collections.abc

Abstract base classes (**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. Work that may never pay off.

This is the reason custom abstract base classes are not used so often. Despite that, the collections.abc module provides a lot of predefined ABCs that allow for 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 ABCMeta.

The most common abstract base classes from the collections.abc that you will use from time to time are as follows:

- Container: This interface means that the object supports the in operator and implements the __contains__() method.
- Iterable: This interface means that the object supports the 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 key in dictionaries) and implements the __hash__ method.
- Sized: This interface means that the object has 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 (refer to https://docs.python.org/3/library/collections.abc.html).

Proxy

Proxy provides indirect access to an expensive or distant resource. Proxy sits between a **Client** and a **Subject**, as shown in the following diagram:
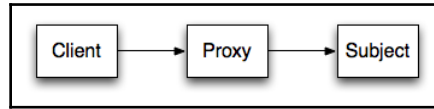
It is intended to optimize **Subject** accesses if they are expensive. For instance, the `memoize()` and `lru_cache()` decorators described in `Chapter 12`, *Test-Driven Development*, can be considered proxies.

A proxy can also be used to provide smart access to a subject. For instance, big video files can be wrapped into proxies to avoid loading them into memory when the user just asks for their titles.

An example is given by the `urllib.request` module. `urlopen` is a proxy for the content located at a remote URL. When it is created, headers can be retrieved independently from the content itself without the need to read the rest of the response:

```
>>> class Url(object):
...     def __init__(self, location):
...         self._url = urlopen(location)
...     def headers(self):
...         return dict(self._url.headers.items())
...     def get(self):
...         return self._url.read()
...
>>> python_org = Url('http://python.org')
>>> python_org.headers().keys()
dict_keys(['Accept-Ranges', 'Via', 'Age', 'Public-Key-Pins', 'X-Clacks-
Overhead', 'X-Cache-Hits', 'X-Cache', 'Content-Type', 'Content-Length',
'Vary', 'X-Served-By', 'Strict-Transport-Security', 'Server', 'Date',
'Connection', 'X-Frame-Options'])
```

This can be used to decide whether the page has been changed before getting its body to update a local copy, by looking at the `last-modified` header. Let's take an example with a big file:

```
>>> ubuntu_iso =
Url('http://ubuntu.mirrors.proxad.net/hardy/ubuntu-8.04-desktop-i386.iso')
>>> ubuntu_iso.headers()['Last-Modified']
'Wed, 23 Apr 2008 01:03:34 GMT'
```

Another use case of proxies is **data uniqueness**.

For example, let's consider a website that presents the same document in several locations. Extra fields specific to each location are appended to the document, such as a hit counter and a few permission settings. A proxy can be used in that case to deal with location-specific matters, and also to point to the original document instead of copying it. So, a given document can have many proxies, and if its content changes, all locations will benefit from it without having to deal with version synchronization.

Generally speaking, the proxy pattern is useful for implementing a local handle of something that may live somewhere else. Popular reasons for doing so are as follows:

- Makes the process faster
- Avoids external resource access
- Reduces memory load
- Ensures data uniqueness

Facade

Facade provides high-level, simpler access to a subsystem.

A facade is nothing but a shortcut to use the functionality of the application, without having to deal with the underlying complexity of a subsystem. This can be done, for instance, by providing high-level functions at the package level.

A facade is usually done on existing systems, where a package's frequent usage is synthesized in high-level functions. Usually, no classes are needed to provide such a pattern, and simple functions in the `__init__.py` module are sufficient.

A good example of the project that provides a big facade over complicated and complex interfaces is the `requests` package (refer to `http://docs.python-requests.org/`). It really simplifies the madness of dealing with HTTP requests and responses in Python by providing a clean API that is easily readable to developers. It is actually even advertised as *HTTP for humans*. Such ease of use always comes at some price but eventual trade-offs and additional overhead do not scare most of the people from using the `Requests` project as their HTTP tool of choice. In the end, it allows us to finish projects quicker, and the developer's time is usually more expensive than hardware.

Facade simplifies the usage of your packages. Facades are usually added after a few iterations with user feedback.

Let's take a look at behavioral patterns in the next section.

Behavioral patterns

Behavioral patterns are intended to simplify the interactions between classes by structuring the processes of their interaction.

This section provides three examples of popular behavioral patterns that you may want to consider when writing Python code:

- Observer
- Visitor
- Template

Let's examine these three examples in the next sections.

Observer

The **observer** pattern is used to notify a list of objects about a state change of the observed component. We have already discussed this pattern briefly in the previous chapter, but here we will discuss some practical examples of a situation where this pattern could be applied.

For instance, let's imagine we have an application that stores marketing materials (briefs, presentations, videos, and flyers) and legal documents in digital form for the sales department of a large company. The company is large, with many sales representatives, and has multiple documents to maintain. A system performs multiple tasks to process these digital documents and make sure that sales representatives are always aware which materials have been updated and that documents can be easily shared with their prospects:

- Video materials are converted to files of different sizes and re-encoded with portable audio-video codecs.
- PDF documents have generated previews that are used as thumbnails in the company's CMS system.
- All new documents are collected in a weekly newsletter broadcast to all sales department employees.

- New confidential materials are encrypted to provide additional data safety.
- Users that have downloaded specific materials previously are notified about updates.

Therefore, almost every component of the system is concerned about events related to every document lifetime. We could design our application in such a way that every component receives information about modifications that are done to documents. The observer pattern is especially good in this situation because it's an observer's responsibility to decide which types of events are interesting for it. From there, every independent component will get notified every time there's an event that it has subscribed to. Of course, this requires that all the code that deals with the actual state of an observed object (for example, creating, modifying, or deleting documents) is triggering such events. But this is way easier than maintaining a manually long list of hooks to call on every time something happens to the observed object. A popular web framework that supports this programming pattern is **Django**, with its mechanism of signals.

An Event class can be implemented for registration of observers in Python by working at the class level:

```python
class Event:
    _observers = []

    def __init__(self, subject):
        self.subject = subject

    @classmethod
    def register(cls, observer):
        if observer not in cls._observers:
            cls._observers.append(observer)

    @classmethod
    def unregister(cls, observer):
        if observer in cls._observers:
            cls._observers.remove(observer)

    @classmethod
    def notify(cls, subject):
        event = cls(subject)
        for observer in cls._observers:
            observer(event)
```

The idea is that observers register themselves using the `Event` class method and get notified with `Event` instances that carry the subject that triggered them. Here is an example of the concrete `Event` subclass with some observers subscribed to its notifications:

```
class WriteEvent(Event):
    def __repr__(self):
        return 'WriteEvent'


def log(event):
    print(
        '{!r} was fired with subject "{}"'
        ''.format(event, event.subject)
    )


class AnotherObserver(object):
    def __call__(self, event):
        print(
            "{!r} trigerred {}'s action"
            "".format(event, self.__class__.__name__)
        )


WriteEvent.register(log)
WriteEvent.register(AnotherObserver())
```

And here is an example result of firing the event with the `WriteEvent.notify()` method:

```
>>> WriteEvent.notify("something happened")
WriteEvent was fired with subject "something happened" WriteEvent trigerred
AnotherObserver's action
```

This implementation is simple and serves only an illustrational purpose. To make it fully functional, it could be enhanced by the following:

- Allowing the developer to change the order or events
- Making the `event` object hold more information than just the subject

De-coupling your code is fun and the observer is the right pattern to do it. It modularizes your application and makes it more extensible. If you want to use an existing tool, try **Blinker**. It provides fast and simple object-to-object and broadcast signaling for Python objects.

Visitor

Visitor helps separate algorithms from data structures and has a similar goal to that of the observer pattern. It allows extending the functionalities of a given class without changing its code. But the visitor goes a bit further by defining a class that is responsible for holding data, and pushes the algorithms to other classes called **visitors**. Each visitor is specialized in one algorithm and can apply it to the data. This behavior is quite similar to the MVC paradigm, where documents are passive containers pushed to views through controllers, or where models contain data that is altered by a controller.

The `visitor` pattern is implemented by providing an entry point in the data class that can be visited by all kinds of visitors. A class that exposes its data through the `visitor` pattern will be called `visitable` and a class that accesses its data will be called the `visitor`.

The `visitable` class decides how it calls the `visitor` class, for instance, by deciding which method is called. For example, a `visitor` in charge of printing built-in type content can implement the `visit_TYPENAME()` methods, and each of these types can call the given method in its `accept()` method:

```
class VisitableList(list):
    def accept(self, visitor):
        visitor.visit_list(self)


class VisitableDict(dict):
    def accept(self, visitor):
        visitor.visit_dict(self)


class Printer(object):
    def visit_list(self, instance):
        print('list content: {}'.format(instance))

    def visit_dict(self, instance):
        print('dict keys: {}'.format(
            ', '.join(instance.keys()))
        )
```

This is done as shown in the following example:

```
>>> visitable_list = VisitableList([1, 2, 5])
>>> visitable_list.accept(Printer())
list content: [1, 2, 5]
>>> visitable_dict = VisitableDict({'one': 1, 'two': 2, 'three': 3})
>>> visitable_dict.accept(Printer())
dict keys: two, one, three
```

But this pattern means that each visited class needs to have an `accept` method to be visited, which is quite painful.

Since Python allows code introspection, a better idea is to automatically link visitors and visited classes:

```
>>> def visit(visited, visitor):
...     cls = visited.__class__.__name__
...     method_name = 'visit_%s' % cls
...     method = getattr(visitor, method_name, None)
...     if isinstance(method, Callable):
...         method(visited)
...     else:
...         raise AttributeError(
...             "No suitable '{}' method in visitor"
...             "".format(method_name)
...         )
...
>>> visit([1,2,3], Printer())
list content: [1, 2, 3]
>>> visit({'one': 1, 'two': 2, 'three': 3}, Printer())
dict keys: two, one, three
>>> visit((1, 2, 3), Printer())
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 10, in visit
AttributeError: No suitable 'visit_tuple' method in visitor
```

This pattern is used in this way in the `ast` module, for instance, by the `NodeVisitor` class that calls the visitor with each node of the compiled code tree. This is because Python doesn't have a match operator like Haskell.

Another example is a directory walker that calls `Visitor` methods depending on the file extension:

```
>>> def visit(directory, visitor):
...     for root, dirs, files in os.walk(directory):
...         for file in files:
...             # foo.txt → .txt
...             ext = os.path.splitext(file)[-1][1:]
...             if hasattr(visitor, ext):
...                 getattr(visitor, ext)(file)
...
>>> class FileReader(object):
...     def pdf(self, filename):
...         print('processing: {}'.format(filename))
...
```

```
>>> walker = visit('/Users/tarek/Desktop', FileReader())
processing slides.pdf
processing sholl23.pdf
```

If your application has data structures that are visited by more than one algorithm, the visitor pattern will help separate concerns. It is better for a data container to focus only on providing access to data and holding it, and nothing else.

Template

Template helps design a generic algorithm by defining abstract steps, which are implemented in subclasses. This pattern uses the **Liskov Substitution Principle** (**LSP**), which is defined by Wikipedia as follows:

> *"If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program."*

In other words, an abstract class can define how an algorithm works through steps that are implemented in concrete classes. The abstract class can also give a basic or partial implementation of the algorithm, and let developers override its parts. For instance, some methods of the Queue class in the queue module can be overridden to make its behavior vary.

Let's implement an example template for a class that deals with text indexing. Indexer is an indexer class that processes text in five steps. This general process is common to every concrete indexing technique:

1. Text normalization
2. Text split
3. Stop words removal
4. Stem words
5. Frequency

Indexer provides a partial implementation for the process algorithm, but requires _remove_stop_words and _stem_words to be implemented in a subclass. BasicIndexer implements the strict minimum, while LocalIndex uses a stop word file and a stem words database. FastIndexer implements all steps and could be based on a fast indexer such as **Xapian** or **Lucene**.

A toy implementation could be as follows:

```python
from collections import Counter

class Indexer:
    def process(self, text):
        text = self._normalize_text(text)
        words = self._split_text(text)
        words = self._remove_stop_words(words)
        stemmed_words = self._stem_words(words)

        return self._frequency(stemmed_words)

    def _normalize_text(self, text):
        return text.lower().strip()

    def _split_text(self, text):
        return text.split()

    def _remove_stop_words(self, words):
        raise NotImplementedError

    def _stem_words(self, words):
        raise NotImplementedError

    def _frequency(self, words):
        return Counter(words)
```

From there, a `BasicIndexer` implementation could be as follows:

```python
class BasicIndexer(Indexer):
    _stop_words = {'he', 'she', 'is', 'and', 'or', 'the'}

    def _remove_stop_words(self, words):
        return (
            word for word in words
            if word not in self._stop_words
        )

    def _stem_words(self, words):
        return (
            (
                len(word) > 3 and
                word.rstrip('aeiouy') or
                word
            )
            for word in words
        )
```

And as always, here is an example usage for the preceding example code:

```
>>> indexer = BasicIndexer()
>>> indexer.process("Just like Johnny Flynn said\nThe breath I've taken and
the one I must to go on")
Counter({"i'v": 1, 'johnn': 1, 'breath': 1, 'to': 1, 'said': 1, 'go': 1,
'flynn': 1, 'taken': 1, 'on': 1, 'must': 1, 'just': 1, 'one': 1, 'i': 1,
'lik': 1})
```

A template should be considered for an algorithm that may vary and can be expressed as isolated substeps. This is probably the most used pattern in Python and does not always need to be implemented via subclassing. For instance, a lot of built-in Python functions that deal with algorithmic problems accept arguments that allow to delegate part of the implementation to the external implementation. For instance, the `sorted()` function allows for an optional `key` keyword argument that is later used by the sorting algorithm. This is also the same for `min()` and `max()` functions that find the minimal and maximal values in the given collection.

Summary

Design patterns are reusable, somewhat language-specific, solutions to common problems in software design. They are a part of the culture of all developers, no matter what language they use.

So, having implementation examples for the most used patterns for a given language is a great way to document it. In many sources (web articles and books), you will easily find an implementation for every design pattern mentioned in GoF books. This is why we concentrated only on patterns that are most common and popular in the context of the Python language.

We covered the three most important groups of design patterns (creational, structural, and behavioral) with some practical examples of their implementation. This short and opinionated selection of patterns should already help you to improve your application structure. But this list is not complete. Fortunately, after reading this book, you are ready to explore this, and every other Python-related topic, completely on your own.

reStructuredText Primer

This chapter provides a brief tutorial on how to use the **reStructuredText** (**reST**) markup language (refer to `http://docutils.sourceforge.net/rst.html`). It is a plain text markup language widely used in the Python community to document packages. The great thing about reST is that the text is still readable, since the markup syntax does not obfuscate the text as LaTeX would.

In this chapter, we will cover the following topics:

- reStructuredText
- Section structure
- Lists
- Inline markup
- Literal block
- Links

reStructuredText

reST comes in `docutils`, a package that provides a suite of scripts to transform a reST file into various formats, such as HTML, LaTeX, XML, or even S5, Eric Meyer's slide show system (refer to `http://meyerweb.com/eric/tools/s5`).

Here's a sample of such a document:

```
=====
Title
=====

Section 1
=========
This *word* has emphasis.

Section 2
=========

Subsection
::::::::::

Text.
```

Writers can focus on the content and then decide how to render it, depending on their needs. For instance, Python itself is documented in reST, which is then rendered in HTML and various other formats. You can visit the official Python documentation via `http://docs.python.org`.

The minimum elements you should know to start writing reST are these:

- Section structure
- Lists
- Inline markup
- Literal block
- Links

This section is a really quick overview of the syntax. A quick reference is available for more information at `http://docutils.sourceforge.net/docs/user/rst/quickref.html`, which is a good place to start working with reST.

To install reStructuredText, install `docutils`:

```
$ pip install docutils
```

For instance, the `rst2html` script provided by the `docutils` package will produce HTML output given a reST file:

```
$ cat text.txt
Title
=====

content.

$ rst2html.py text.txt
<?xml version="1.0" encoding="utf-8" ?>
(...)
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
(...)
</head>
<body>
<div class="document" id="title">
<h1 class="title">Title</h1>
<p>content.</p>
</div>
</body>
</html>
```

Let's take a look at all of the elements that we need to keep in mind in the next sections.

Section structure

The document's title and its sections are underlined using **non-alphanumeric** characters. They can be overlined and underlined, and a common practice is to use a double markup for the title and keep a simple underline for the section headings.

The most used characters to underline a section title are in the following order of precedence: =, −, _, :, #, +, and ^.

When a character is used for a section, it is associated with its level and it has to be used consistently throughout the document.

Consider the following code, for example:

```
==============
Document title
==============

Introduction to the document content.


Section 1
=========

First document section with two subsections.

Note the ``=`` used as heading underline.


Subsection A
------------

First subsection (A) of Section 1.

Note the ``-`` used as heading underline.


Subsection B
------------
Second subsection (B) of Section 1.


Section 2
=========
```

```
Second section of document with one subsection.


Subsection C
------------

Subsection (C) of Section 2.
```

The following screenshot is the output of the code:
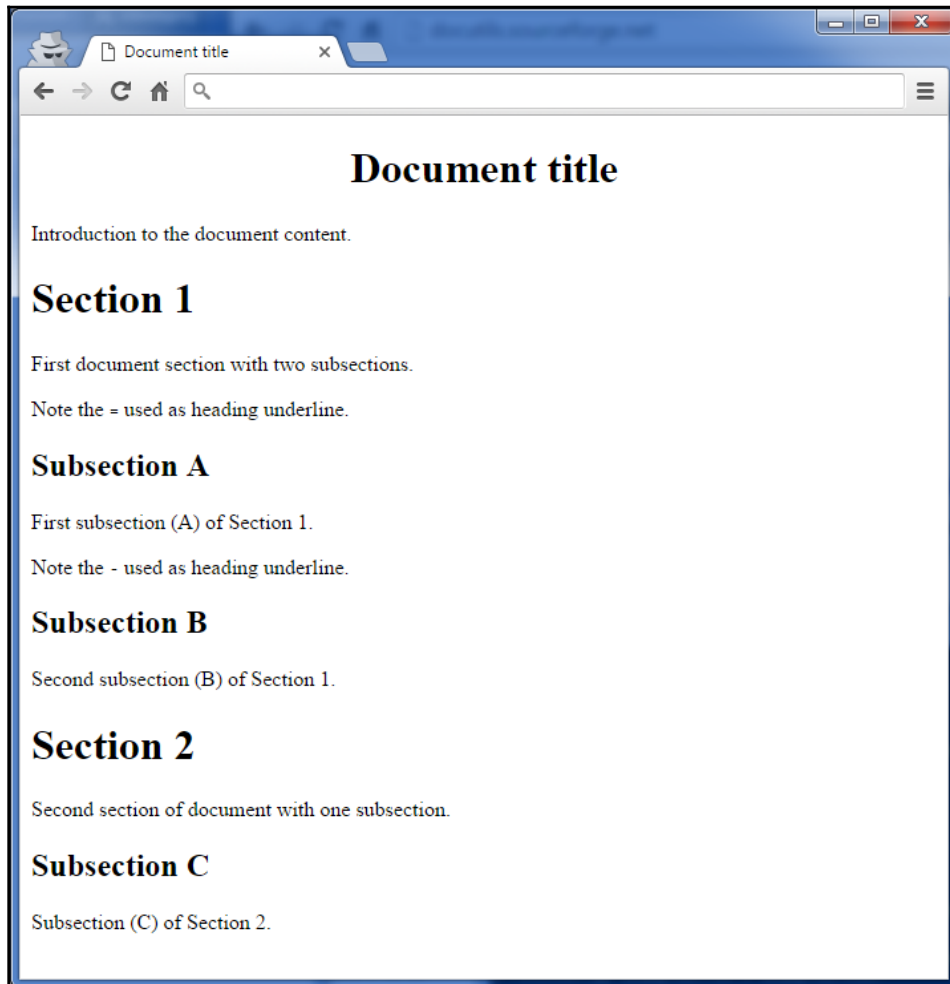
Figure 1: reStructuredText converted into HTML and rendered in the browser

Let's take a look at lists in the next section.

Lists

reST provides readable syntax for bullet lists, enumerated lists, and definition lists with auto-enumeration features. This is shown in the following code example:

```
Bullet list:

- one
- two
- three


Enumerated list:

1. one
2. two
#. auto-enumerated


Definition list:

one
    one is a number.

two
    two is also a number.
```

The output of the code is shown in the following screenshot:
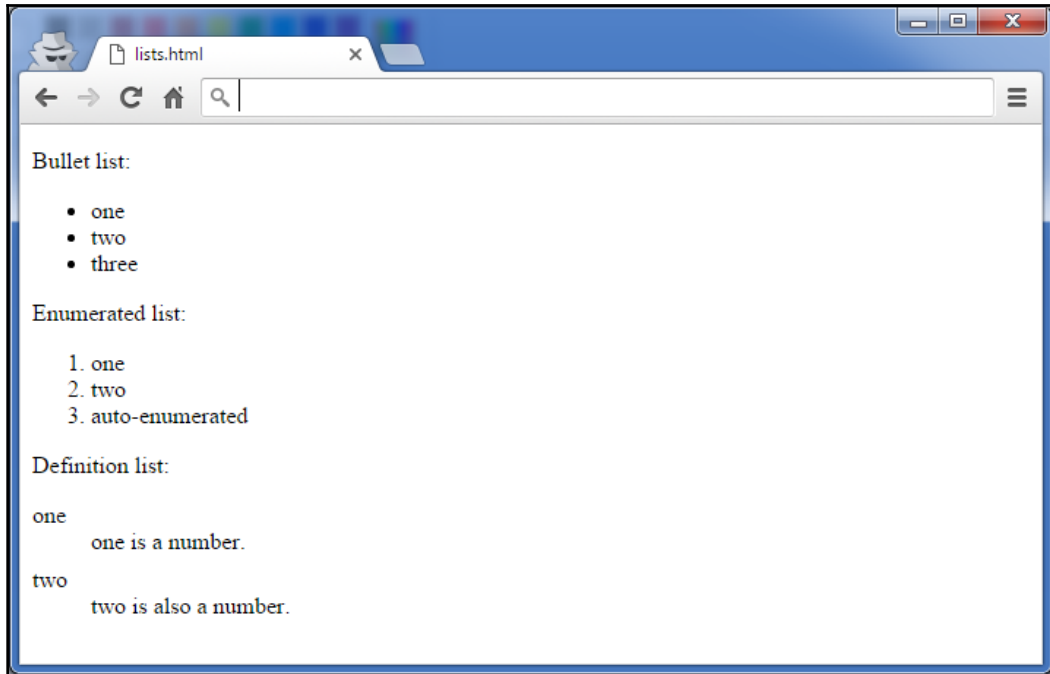
Figure 2: Different types of lists rendered as HTML

The next section talks about inline markup.

Inline markup

The text can be styled using an inline markup:

- `*emphasis*`: Italics.
- `**strong emphasis**`: Boldface.
- ` ``inline preformatted`` `: Inline preformatted text (usually monospaced, terminal-like).

- `` `a text with a link`_ ``: This will be replaced by a hyperlink as long as it is provided in the document (see the *Links* section).

Literal block is described in the next section.

Literal block

When you need to present some code examples, a **literal block** can be used. Two colons are used to mark the block, which is an indented paragraph:

```
This is a code example

::

    >>> 1 + 1
    2

Let's continue our text
```

> Don't forget to add a blank line after :: and after the block otherwise, it will not be rendered.

Notice that the colon characters can be put in a text line. In that case, they will be replaced by a single colon in various rendering formats:

```
This is a code example:

    >>> 1 + 1
    2

Let's continue our text
```

If you don't want to keep a single colon, you can insert a space between the example and `::`. In that case, `::` will be interpreted and totally removed:
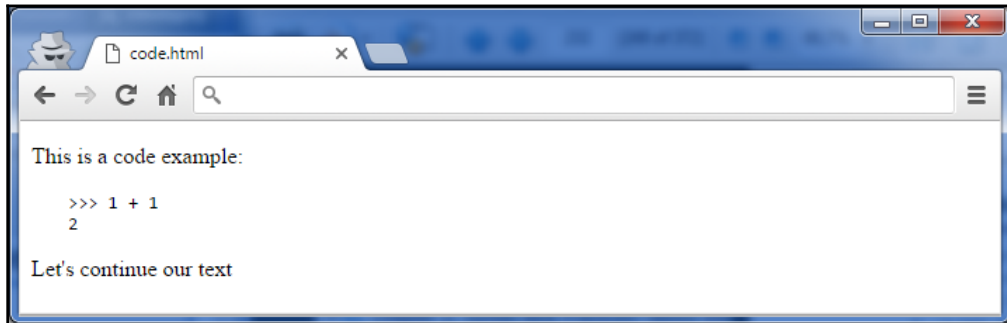
Figure 3: Code samples in reST rendered as HTML

We'll take a look at links in the next section

Links

A text can be changed into an external **link** with a special line starting with two dots, as long as it is provided in the document:

```
Try `Plone CMS`_, it is great ! It is based on Zope_.

.. _`Plone CMS`: http://plone.org
.. _Zope: http://zope.org
```

The usual practice is to group the external links at the end of the document. When the text to be linked contains spaces, it has to be surrounded with ` (backtick) characters.

Internal links can also be used by adding a marker to the text:

```
This is a code example

.. _example:

::

    >>> 1 + 1
    2

Let's continue our text, or maybe go back to
the example_.
```

Sections are also targets that can be used:

```
==============
Document title
==============

Introduction to the document content.


Section 1
=========

First document section.


Section 2
=========

-> go back to `Section 1`_
```

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:
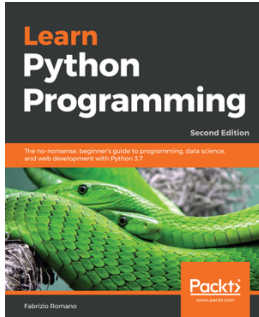
Learn Python Programming - Second Edition
Fabrizio Romano

ISBN: 978-1-78899-666-2

- Get Python up and running on Windows, Mac, and Linux
- Explore fundamental concepts of coding using data structures and control flow
- Write elegant, reusable, and efficient code in any situation
- Understand when to use the functional or OOP approach
- Cover the basics of security and concurrent/asynchronous programming
- Create bulletproof, reliable software by writing tests
- Build a simple website in Django
- Fetch, clean, and manipulate data
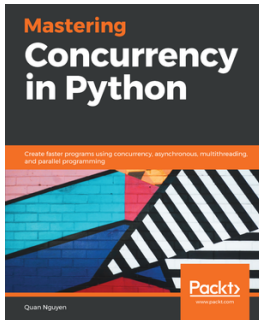
Mastering Concurrency in Python
Quan Nguyen

ISBN: 978-1-78934-305-2

- Explore the concepts of concurrency in programming
- Explore the core syntax and features that enable concurrency in Python
- Understand the correct way to implement concurrency
- Abstract methods to keep the data consistent in your program
- Analyze problems commonly faced in concurrent programming
- Use application scaffolding to design highly-scalable programs

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

E

F

G