EXPERT INSIGHT

Fully updated with Python 3.13

python™

Python Object-Oriented Programming

Learn how and when to apply OOP principles to build scalable and maintainable Python applications

Fifth Edition

Steven F. Lott
Dusty Phillips

packt

Python Object-Oriented Programming

Fifth Edition

Learn how and when to apply OOP principles to build scalable and maintainable Python applications

Steven F. Lott

Dusty Phillips

Python Object-Oriented Programming

Fifth Edition

Portfolio Director: Kunal Chaudhari
Relationship Lead Dhruv J. Kataria
Project Manager: K. Loganathan
Content Engineer: Deepayan Bhattacharjee
Technical Editor: Irfa Ansari
Copy Editor: Safis Editing
Indexer: Hemangini Bari
Proofreader: Deepayan Bhattacharjee
Presentation Designer: Salma Patel
Growth Lead: Mansi Shah

First published: July 2010
Second edition: August 2015
Third edition: October 2018
Fourth edition: June 2021
Fifth edition: November 2025

Production reference: 1261125

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-83664-259-6

www.packt.com

Contributors

About the authors

Steven Lott has been programming since computers were large, expensive, and rare. Working for decades in high tech has given him exposure to a lot of ideas and techniques—some bad, but most are useful and helpful to others.

Steven has been working with Python since the '90s, building a variety of tools and applications. He's written a number of titles for Packt Publishing, including *Mastering Object-Oriented Python*, *Modern Python Cookbook*, and *Functional Python Programming*.

He's a technomad, and travels extensively around the US. He tries to live by words his mother shared.

> *"Don't come home until you have a story."*

Dusty Phillips is a Canadian author and software developer. His storied career has included roles with the world's biggest government, the world's biggest social network, a two person startup, and everything in between. In addition to *Python Object-Oriented Programming*, Dusty wrote *Creating Apps In Kivy* (O'Reilly) and is now focused on writing fiction.

Thank you to Steven Lott, for finishing what I started, to all my readers for appreciating what I write, and to my wife, Jen Phillips, for everything else.

About the reviewer

Alejandro Rodas de Paz is a computer engineer from Seville, Spain. He has been developing Python projects in both professional and academic settings since 2010, including work on AI algorithms and DevOps automation.

Prior to this publication, Alejandro wrote Packt's titles *Python Game Development by Example* and *Tkinter GUI Application Development Cookbook*. He also collaborated as a technical reviewer on the books *Tkinter GUI Application Development Hotshot* and *Python GUI Programming with Tkinter*.

I would like to dedicate my contribution to this book to Frank, Montse and Oliver. I wish you all the best in this new stage of your lives.

Join our community Discord space

Join our Python Discord workspace to discuss and find out more about the book:

`https://packt.link/dHrHU`

Table of Contents

Preface

The Python programming language is extremely popular and used for a variety of applications. The Python language is designed to make it relatively easy to create small programs. To create more sophisticated software, we need to acquire a number of important programming and software design skills.

This book describes the **object-oriented** approach to creating programs in Python. It introduces the terminology of object-oriented programming, demonstrating software design and Python programming through step-by-step examples. It describes how to make use of inheritance and composition to build software from individual elements. It shows how to use Python's built-in exceptions and data structures, as well as elements of the Python standard library. A number of common design patterns are described with detailed examples.

This book covers how to write automated tests to confirm that our software works. It also shows how to use the various concurrency libraries available as part of Python, making effective use of multiple cores and multiple processors in a modern computer.

Who this book is for

This book targets people who are new to object-oriented programming in Python. It assumes basic Python skills and familiarity with Python's tools, including **PIP** for installing packages. Starting with the *fifth edition*, we're assuming the reader has been through the first 8 sections of https://docs.python.org/3/tutorial/. For readers with a background in another object-oriented programming language, this book will expose many distinctive features of Python's approach.

Because of Python's use for data science and data analytics, this book touches on a few math and statistics concepts. Some knowledge in these areas can help to make the applications of the concepts more concrete.

What this book covers

This book is divided into four overall sections. The first six chapters provide the core principles and concepts of object-oriented programming and how these are implemented in Python. The next three chapters take a close look at Python built-in features through the lens of object-oriented programming. This is followed by three chapters focused on common design patterns and how these can be handled in Python. The final section covers two additional topics: testing and concurrency.

Chapter 1, introduces the core concepts underlying object-oriented design. This provides a road map through the ideas of state and behavior, attributes and methods, and how objects are grouped into classes. This chapter looks at the basic principles of object-oriented design.

Chapter 2, shows how class definitions work in Python. This will include the type annotations, called type hints, class definitions, modules, and packages. We'll talk about practical considerations for class definition and encapsulation. We'll touch on how to extend Python with additional libraries and how to manage virtual environments.

Chapter 3, addresses how classes are related to each other. This will include how to make use of inheritance and multiple inheritance. We'll look at the concept of polymorphism among the classes in a class hierarchy. We'll look closely at how Python's approach of "duck typing" works.

Chapter 4, looks closely at Python's exceptions and exception handling. We'll look at the built-in exception hierarchy. We'll also look at how unique exceptions can be defined to reflect a unique problem domain or application.

Chapter 5, dives more deeply into design techniques. It looks closely at data and behavior and how these are reflected in class design. This chapter will look at how attributes can be implemented via Python's properties.

Chapter 6, is a deep dive into the idea of abstraction, and how Python supports abstract base classes. This will involve comparing **duck typing** with more formal methods of `Protocol` definition. It will include techniques for overloading Python's built-in operators. We will also look at metaclasses and how these can be used to modify class construction.

Chapter 7 examines type hints and how they're used in object-oriented programming. We'll look at tools for validating the type annotations. We'll look at packages to help with run-time type checking, also.

Chapter 8, examines a number of Python's built-in collections. This chapter examines tuples, dictionaries, lists, and sets. It also looks at how dataclasses and named tuples can simplify a design

by providing a number of common features of a class.

Chapter 9, looks at Python constructs that aren't simply class definitions. While all of Python is object-oriented, function definitions allow us to create callable objects without the clutter of a class definition. We'll also look at Python's context manager construct and the `with` statement.

Chapter 10, describes the ubiquitous concept of iteration in Python. All the built-in collections are iterable, and this design pattern is central to a great deal of how Python works. We'll look at Python comprehensions and generator functions, also.

Chapter 11, looks at some common object-oriented design patterns. This will include the Decorator, Observer, Strategy, Command, State, and Singleton patterns.

Chapter 12, looks at some more advanced object-oriented techniques. This will include the Adapter, Façade, Flyweight, Abstract Factory, Composite, and Template design patterns.

Chapter 13, shows how to use tools like `unittest` and `pytest` to provide an automated unit test suite for a Python application. This will look at some more advanced testing techniques, including the use of mock objects to isolate the unit under test.

Chapter 14, looks at how we can make effective use of multi-core and multi-processor computer systems to do computations rapidly. These techniques can help create software that is responsive to external events. We'll look at threads and multiprocessing, as well as Python's `asyncio` module.

To get the most out of this book

All the examples were tested with Python 3.12.5. The **pyright** tool, version 1.1, was used to confirm that the type hints were consistent.

Some of the examples depend on an internet connection to gather data. These interactions with websites generally involve small downloads.

Some of the examples involve packages that are not part of Python's built-in standard library. In the relevant chapters, we note the packages and provide the install instructions. All of these extra packages are in the Python Package Index, at https://pypi.org.

Download the example code files

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/Python-Object-Oriented-Programming-5E.

We also have other code bundles from our rich catalog of books and videos available at `https:`

`//github.com/PacktPublishing/` Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://static.packt-cdn.com/downloads/9781801077262_ColorImages.pdf

Conventions used

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, and user input are shown as follows: "You can confirm Python is running by importing the `antigravity` module at the `>>>` prompt."

A block of code is set as follows:

```
class Fizz:
    def member(self, v: int) -> bool:
        return v % 5 == 0
```

Any Python interactive sessions are written as follows:

```
[pycon]
>>> import math
>>> math.factorial(52)
80658175170943878571660636856403766975289505440883277824000000000000
```

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

```
python -m pip install tox
```

New terms and important words are shown in **bold**.

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

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

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

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

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

Share your thoughts

Once you've read *Python Object-Oriented Programming, Fifth Edition*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

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

Free Benefits with Your Book

This book comes with free benefits to support your learning. Activate them now for instant access (see the "*How to Unlock*" section for instructions). Here's a quick overview of what you can instantly unlock with your purchase:

PDF and ePub Copies

- Access a DRM-free PDF copy of this book to read anywhere, on any device.
- Use a DRM-free ePub version with your favorite e-reader.

Next-Gen Web-Based Reader

- Multi-device progress sync: Pick up where you left off, on any device.
- Highlighting and note-taking: Capture ideas and turn reading into lasting knowledge.
- Bookmarking: Save and revisit key sections whenever you need them.
- Dark mode: Reduce eye strain by switching to dark or sepia themes.

How to Unlock

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

Note: Keep your invoice handy. Purchases made directly from Packt don't require one

1

Object-Oriented Design

In software development, design is often considered the step that comes *before* programming. This isn't true; in reality, analysis, programming, and design tend to overlap, combine, and interweave. Throughout this book, we'll be covering a mixture of design and programming issues without trying to parse the two into separate buckets. One of the advantages of a language such as Python is its ability to express the design clearly.

In this chapter, we will talk a little about how we can move from a good idea toward writing software. We'll create some design artifacts — such as diagrams — that can help clarify our thinking before we start writing code. We'll cover the following topics:

- What object-oriented means
- The difference between object-oriented design and object-oriented programming
- Some basic principles of object-oriented design
- Basic **Unified Modeling Language** (**UML**) and when it isn't evil

Technical requirements

The code for this chapter can be found in the `PacktPublishing` repository: `https://github.com/PacktPublishing/Python-Object-Oriented-Programming-5E`. Within

that repository's files, we'll focus on the `ch_01` directory.

All of the examples were tested with Python 3.12 and 3.13. The **uv** tool can be used to test the code: `uvx tox`.

What object-oriented means

Outside of the world of software, an object is a tangible thing that we can sense, feel, and manipulate. The earliest objects we interact with are typically baby toys. Wooden blocks, plastic shapes, and over-sized puzzle pieces are common first objects. Babies learn quickly that certain objects do certain things: bells ring, buttons are pressed, and levers are pulled.

The definition of an object in software development is not terribly different. Software objects may not be tangible things that you can pick up, sense, or feel, but they are models of something that has an internal state, can do certain things, and responds when things are done to it. Formally, an object is a collection of **data** and associated **behaviors**.

Object-oriented programming means writing code directed toward modeling objects. This is one of many techniques used to describe the actions of complex systems. The overall behavior emerges from collaboration between objects. A complicated internal state is decomposed into the states of separate objects.

To do object-oriented programming well, there are some additional disciplines. These include *object-oriented analysis*, *object-oriented design*, and even the combined and streamlined *object-oriented analysis and design*. All of these disciplines use the foundational concept of software objects and their interactions to analyze a problem and design a solution. Object interactions include creating objects, changing their values, and associating them with other objects.

Analysis, design, and programming are all stages of software development. Calling them object-oriented clarifies what kind of modeling techniques will be employed.

Object-oriented analysis (**OOA**) is the process of looking at a problem and identifying the objects and interactions between those objects. The analysis stage is all about describing *what* needs to be done.

The output of the analysis stage is a description of a problem and a solution to the problem, often in the form of *requirements*. If we were to complete the analysis stage in one step, we would have turned a task, *I am a botanist and need a website to help users classify plants so I can help with correct identification*, into a set of required features. As an example, here are some requirements for what a

website visitor might need to do. Each item is an action bound to an object; we've written them with *italics* to highlight the actions, and **bold** to highlight the objects:

- *Browse* **previous uploads**
- *Upload new* **known examples**
- *Test* for **quality**
- *Browse* **products**
- *See* **recommendations**

In some ways, the term *analysis* can be a misnomer. A baby interacting with objects doesn't analyze the blocks and puzzle pieces. Instead, they explore their environment, manipulate shapes, and see where they might fit. A better turn of phrase might be *object-oriented exploration*. In software development, the initial stages of analysis include exploration: interviewing customers, studying their processes, and eliminating possibilities that don't solve the problem.

Object-oriented design (**OOD**) is the process of converting such requirements into an implementation specification. The designer must name the objects, define the behaviors, and formally specify which objects can activate specific behaviors in other objects. The design stage is all about transforming *what* should be done into *how* it should be done.

The output of the design stage is an implementation specification. If we were to complete the design stage in a single step, we would have turned the requirements into a set of specifications for object classes. The specification might include state transition diagrams, collaboration diagrams, activity diagrams, and other useful details to describe state and behavior. These are ready to be implemented in (ideally) an object-oriented programming language.

Object-oriented programming (**OOP**) is the process of converting a design into a working program. This program does what the product owner originally requested during the analysis phase.

It would be lovely if the world met this ideal. If it did, we could follow these stages one by one, in order, as a well-defined method for producing software. As usual, the real world is much murkier. No matter how hard we try to separate these stages, we'll always find things that need further analysis while we're designing. When we're programming, we find features that need clarification from further design.

Most modern software development practices recognize that a cascade (or waterfall) of stages doesn't work out well. What seems to be better is an *iterative* development model. In iterative development, a small part of the task is analyzed, designed, and programmed. The developers can be

said to form a scrum (or "scrummage", as in the game of rugby), where the team members all push in one direction in concert. The resulting product increment is reviewed. Iterative development uses repeated cycles of improving existing features and adding new features. The scrum methodology emphasizes this periodic reset of the team followed by the focused pursuit of a near-term goal. At some point, the product is usable. Development is never really finished, but at some point it becomes clear that the cost of making any improvement will outweigh the benefits.

The rest of this book is about OOP. In this chapter, we will cover the basic object-oriented principles in the context of design. This allows us to understand concepts without having to work through Python language features at the same time.

Objects and classes

An object is a collection of data that defines an internal state with associated behaviors. How do we differentiate between categories of objects? Apples and oranges are both objects, but it is a common adage that they cannot be compared. Apples and oranges aren't modeled very often in computer programming, but let's pretend we're creating an inventory application for a fruit farm.

One of the earliest things we learn in our analysis is that apples go in barrels and oranges go in baskets. The problem domain we've uncovered so far has four kinds or categories or varieties of objects: apples, oranges, baskets, and barrels. In object-oriented modeling, the preferred term used for a *kind of object* is **class**. We seem to have four classes of objects.

It's important to understand the difference between an object and a class. The idea is that objects can be classified based on common states or behaviors. Classes describe related objects. A class definition is like a blueprint for creating individual objects. You might have three oranges sitting on the table in front of you. Each orange is a distinct object, but all three have the attributes and behaviors associated with one class: the general class of oranges.

The relationship between the four classes of objects in our inventory system can be described using the **Unified Modeling Language** (invariably referred to as **UML**, because three-letter acronyms never go out of style). Specifically, using a UML class diagram. *Figure 1.1* is our first *class diagram*:

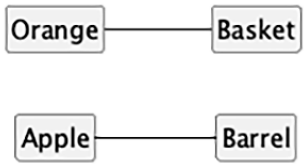


Figure 1.1: Class diagram

This class diagram shows that instances of the **Orange** class (usually called "oranges") are somehow associated with a **Basket**. It also shows how instances of the **Apple** class ("apples") are also somehow associated with a **Barrel**. *Association* is the most basic way for objects (instances of classes) to be related. By limiting ourselves to vague associations, we're avoiding assumptions. An association will often need further clarification. We could, for example, clutter the model with additional attributes — color, size, or any of a large number of other aspects. The diagram helps narrow the discussion to fruit categories and containers.

The syntax of a UML diagram is generally pretty obvious; you don't have to read a tutorial to (mostly) understand what is going on when you see one. UML is also fairly easy to draw. After all, many people, when describing classes and their relationships, will naturally draw boxes with lines between them. Having a standard based on these intuitive diagrams makes it easy for programmers to communicate with designers, product owners, and each other.

Note that the UML diagram generally depicts the class definitions; we often need to describe attributes of the objects. UML diagrams can show a class with attributes and methods; it also allows us to elide details. The diagram shows the class of Apple and the class of Barrel, telling us that any given apple is in some specific barrel. While we can use UML to depict individual objects, that's rarely necessary. Showing the class relationships in a diagram tells us some important details about the objects that are members of each class. It doesn't tell us everything; the model serves to highlight selected details of the overall problem domain.

Some programmers disparage UML as a waste of time. Citing iterative development, they will argue that formal specifications done up in fancy UML diagrams are going to be redundant before they're implemented. Further, they complain that maintaining these formal diagrams will only waste time and not benefit anyone.

Every programming team consisting of more than one person will occasionally have to sit down and hash out the details of the components being built. The higher-performing the team — as a whole — the more often this kind of information is shared. UML is extremely useful for ensuring

quick, easy, and consistent communication. Even those organizations that scoff at formal class diagrams tend to use some informal version of UML in their design meetings or team discussions.

Furthermore, the most important person you will ever have to communicate with is your future self. We all think we can remember the design decisions we've made, but there will always be *Why did I do that?* moments hiding in our future. If we keep the scraps of papers we did our initial diagramming on, when we started a design, we'll eventually find them to be a useful reference.

This chapter, however, is not meant to be a tutorial on UML. There are many of those available on the internet, as well as numerous books on the topic. UML covers far more than class and object diagrams; it also has a syntax for use cases, deployment, state changes, and activities. We'll be dealing with some common class diagram syntax in this discussion of object-oriented design. You can pick up the structure by example, and then you'll subconsciously choose the UML-inspired syntax in your own team or personal design notes.

Our initial diagram, while correct, does not remind us that apples go **in** barrels or how many barrels a single apple can go in. It only tells us that apples are somehow associated with barrels. Sometimes the association between classes is obvious and needs no further explanation. Often, we have to add further clarification.

The beauty of UML is that most things are optional. We only need to specify as much information in a diagram as makes sense for the current situation. In a quick whiteboard session, we might just draw simple lines between boxes. In a more formal, permanent document, we might go into more detail.

In the case of apples and barrels, we can be fairly confident that the association is **many apples go in one barrel**. To make sure nobody confuses the association with **one apple spoils one barrel**, we can enhance the diagram.

Figure 1.2 (next page) shows more detail.

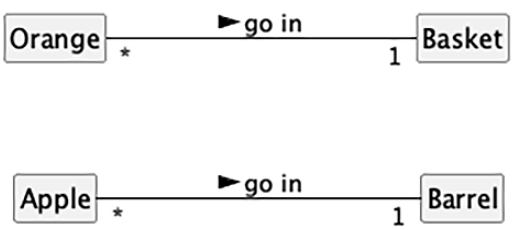


Figure 1.2: Class diagram with more detail

This diagram tells us that oranges **go in** baskets, with a little arrow showing what goes in what. Reading it the other way, a basket contains oranges; this is the foundational "has-a" relationship that we see almost everywhere. There are numerous other kinds of relationships, but we'll focus on this relationship because it's common and easy to visualize. The diagram also tells us the number of an object that can be used in the association on both sides of the relationship. One **Basket** object can contain many **Orange** objects, represented by annotating the line with a *. Any one **Orange** can go in exactly one **Basket**. This number is referred to as the *multiplicity* of the association, which specifies how many instances of one class can be linked to another. While multiplicity is sometimes confused with *cardinality*, the latter defines an exact count of items. In a UML context, multiplicity is how we define the allowed range of concrete cardinality values. A "more-than-one instance" feature of a relationship is better described as the multiplicity.

We may sometimes forget which end of the relationship line is supposed to have which multiplicity annotation. The multiplicity annotation nearest to a class represents how many objects of that class can be associated with any one object at the other end of the association. For the apple goes in a barrel association, reading from left to right, many instances of the **Apple** class (that is, many **Apple** objects) can go in any one **Barrel** object. Reading from right to left, exactly one **Barrel** can be associated with any one **Apple**.

We've seen the basics of classes, and how they can specify relationships among objects. Now, we need to talk about the attributes that define an object's state, and the behaviors of an object that may involve state change or interaction with other objects.

Readers with experience in object-oriented design will note that we haven't described any common features of the **Barrel** or **Basket** classes of objects. We're intentionally avoiding the search from common features for a moment. A premature leap into searching for commonality can sometimes obscure more nuanced distinctions among classes of objects.

Specifying attributes and behaviors

We now have a grasp of some basic object-oriented terminology. Objects are instances of classes that can be associated with each other. A specific orange on the table in front of us is said to be an instance of the general class of oranges. As we'll see in the following sections, the behaviors are implemented as "methods" of the class, introducing another bit of terminology to the mix.

The orange has a state, for example, ripe or raw; we implement the state of an object via the values of specific attributes. An orange also has behaviors. By themselves, oranges are generally passive.

State changes are imposed on them. (Consider how alarming an active orange would be.) Let's dive into the meaning of those two words, *state* and *behaviors*.

Data describes object state

Let's start with data. Data represents the individual characteristics of a certain object; its current state. A class defines the characteristics of all objects that are its members. Any specific object can have different data values for a given characteristic. For example, the three oranges on our table (if we haven't eaten any) could each weigh a different amount. The orange class could have a weight attribute to represent that datum. All instances of the orange class have a weight attribute, but each orange has a different value for this attribute. Attributes don't have to be unique, though; any two oranges may weigh the same amount.

Attributes are frequently referred to as **members** or **properties**. Some authors suggest that the terms have different meanings, usually that attributes are settable, while properties are read-only. A Python property can be defined as read-only, but the value will be based on attribute values that are — ultimately — writable, making the concept of *read-only* rather pointless; throughout this book, we'll see the two terms used interchangeably. In addition, as we'll discuss in *Chapter 5*, the `property` keyword has a special meaning in Python for a particular kind of attribute.

In Python, we can also call an attribute an **instance variable**. This can help clarify the way attributes work. They are variables with unique values for each instance of a class. Python has other kinds of attributes, but we'll focus on the most common kind to get started.

In our fruit inventory application, the fruit farmer may want to know what orchard the orange came from, when it was picked, and how much it weighs. They might also want to keep track of where each **Basket** is stored. Apples might have a color attribute, and barrels might come in different sizes.

We'll often notice that multiple classes have the same properties; we may want to know when apples are picked, too. For this first example, we'll add a few different attributes to our class diagram.

Figure 1.3 shows some attributes:

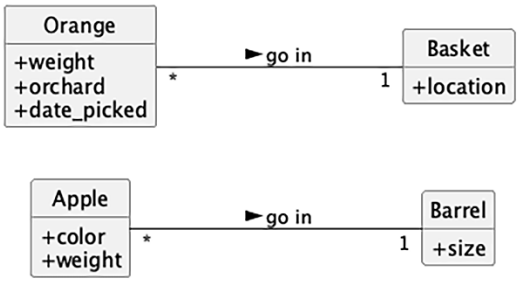


Figure 1.3: Class diagram with attributes

Depending on how detailed our design needs to be, we can also specify the type for each attribute's value. In UML, attribute types are often generic names common to many programming languages, such as integer, floating-point number, string, byte, or Boolean. However, they can also represent generic collections such as lists, trees, or graphs, or most notably, other, non-generic, application-specific classes. This is one area where the design stage can overlap with the programming stage. The various primitives and built-in collections available in one programming language may be different from what is available in another.

Figure 1.4 shows some attributes with (mostly) Python-specific type hints:

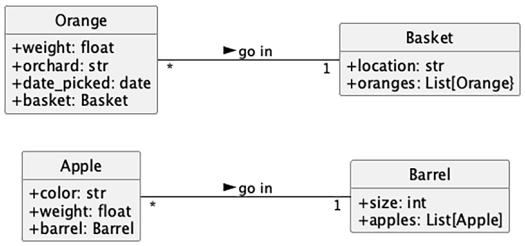


Figure 1.4: Class diagram with attributes and their types

Usually, we don't need to be overly concerned with data types at the design stage, as implementation-specific details are chosen during the programming stage. Generic names are normally sufficient for design; that's why we included `date` as a placeholder for a Python type such as `datetime.datetime`. If our design calls for a list container type, Java programmers can choose to use a `LinkedList` or an `ArrayList` when implementing it, while Python programmers (that's us!) might specify

`list[Apple]` as a type hint, and use the `list` type for the implementation.

In our fruit-farming example so far, our attributes are all built-in primitive types. However, there are some implicit attributes that we can make explicit; these implement the associations. For a given orange, we have an attribute referring to the basket that holds that orange, the `basket` attribute, with a type hint of `Basket`.

Behaviors are actions

Now that we know how data defines the object's state, the last undefined term we need to look at is *behavior*. Behaviors are actions that can occur on an object. The behaviors that can be performed on members of a class are expressed as the **methods** of a class. At the programming level, methods are essentially functions with access to an object's attributes — in Python, these are the instance variables of an object. Like functions, methods can also accept **parameters** and return **values**.

A method's parameters define objects that need to be **passed** into that method. The actual object instances that are passed into a method during a specific invocation are referred to as the **argument values**. These objects are bound to **parameter** variable names in the method body. They are used by the method to perform whatever behavior or task it is meant to do. Returned values are the results of that task. Internal state changes are a possible side-effect of evaluating a method. (Some folks like to talk about "calling" a method or "executing" a method; these are all synonyms.)

We've stretched our *comparing apples and oranges* example into a basic (if far-fetched) inventory application. Let's stretch it a little further and see whether it breaks. The idea is to capture enough details of the problem domain; the software we need to write may not implement every detail of the initial model. One action that can be associated with oranges is the conceptual **pick** action. As we think about implementation details for this class, a `pick` method might need to do two things:

- Place the orange in a basket by updating the **Basket** attribute of the orange.
- Add the orange to the **Orange** list on the given **Basket**.

So, this **pick** method may need to know what basket it is dealing with. We do this by giving the method a **Basket** parameter. Since our fruit farmer also sells juice, we can add a **squeeze** method to the **Orange** class. When evaluated, the **squeeze** method might return the amount of juice retrieved, while also removing the **Orange** from the **Basket** it was in.

The class **Basket** can have a **sell** action. When a basket is sold, our inventory system might update some data on as-yet-unspecified objects for accounting and profit calculations. Alternatively, our basket of oranges might go bad before we can sell them, so we may also need to add a **discard**

method.

Figure 1.5 adds methods to our diagram:

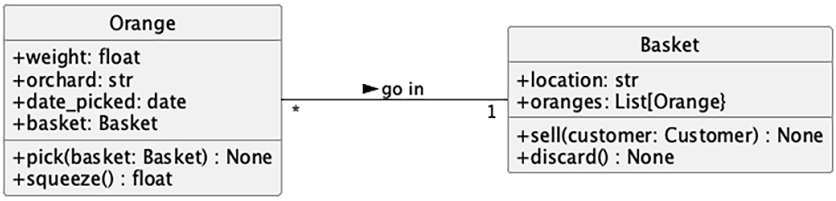


Figure 1.5: Class diagram with attributes and methods

Do we really need all of these methods and the related associations? Unsurprisingly, the answer is often "no": the application software doesn't need to model all of these behaviors. For now, we want to throw ideas at the model to explore what's possible. Later, we'll prune this back to what's necessary.

Adding attributes and methods to individual objects allows us to create a **system** of interacting objects. Each object in the system is a member of a certain class. These classes specify what types of data the object can hold and what methods can be invoked on it. The data in each object can be in a different state from other instances of the same class; each object may react to method calls differently because of the differences in state.

Object-oriented analysis and design are all about figuring out what those objects are and how they should interact. Each class has responsibilities and collaborations. The next section describes principles that can be used to make those interactions as intuitive as possible.

Note that selling a basket is not unconditionally a feature of the Basket class. It may be that some other class (not shown) cares about the various baskets and where they are. We often have boundaries around our design. We will also have questions about responsibilities allocated to various classes. The responsibility allocation problem doesn't always have a tidy technical solution, forcing us to draw (and redraw) our UML diagrams more than once to examine alternative designs.

In many contexts, the analysis process can also be enlightening for product owners and users. What may seem — at first — like an insurmountable business problem, requiring lots of expensive software, may turn out to be a failure of two organizations to collaborate. The process of creating an object-oriented analytical model may reveal details that aren't really solvable with more software. It's not unusual for a software project to proceed through a great deal of object-oriented model building, and then end with a successful outcome for the ultimate users, but little or no software

being written. The models (and the resulting insight) were adequate to understand what's really going on.

Hiding details and creating the public interface

The key purpose of modeling an object in object-oriented design is to determine what the public **interface** of that object will be. The interface is the collection of attributes and methods that other objects can access to interact with that object. Objects do not need, and in some languages are not allowed, to access the internal workings of objects of another class.

A common real-world example is the television (or any appliance, really). Our interface to the television is the remote control. A button on the remote control represents a method that can be called on the television object. When we, as the calling object, access these methods, we do not know or care whether the television is getting its signal from a cable connection, a satellite dish, or an internet-enabled device. The notion here is that "television" is often a complex conglomerate of components. We don't care what electronic signals are being sent to adjust the volume, or whether the sound is destined for speakers or headphones. If we open the television to access its internal workings, for example, to split the output signal to both external speakers and a set of headphones, we may void the warranty.

This process of hiding the implementation details of an object is suitably called **information hiding**. It is also sometimes referred to as **encapsulation**. Encapsulated data is not necessarily hidden. Encapsulation is, literally, creating a capsule (or wrapper) on the attributes. The television's external case encapsulates the state and behavior of the television. We have access to the external screen, the speakers, and the remote. We don't have direct access to the wiring of the amplifiers or receivers within the television's case.

When we buy a component entertainment system, we change the level of encapsulation, exposing more of the interfaces between components. If we're an Internet of Things maker, we may decompose this even further, opening cases and breaking the information hiding attempted by the manufacturer.

The distinction between encapsulation and information hiding is nuanced at the design level. Many practical references use these terms interchangeably. As Python programmers, we don't actually have or need information hiding via completely private, inaccessible variables (we'll discuss the reasons for this in *Chapter 2*), so the more encompassing definition for encapsulation is suitable.

The public interface for a class is very important. It needs to be carefully designed as it can be difficult to change after software has been written and other classes depend on it. We can change

the internals all we like, for example, to make it more efficient, or to access data over the network as well as locally, and the client objects will still be able to talk to it, unmodified, using the public interface. On the other hand, if we alter the interface by changing publicly accessed attribute names or the order or types of arguments that a method can accept, all client classes will also have to be modified. When designing public interfaces, keep it simple. Always design the interface of an object based on how easy it is to use, not how hard it is to code (this advice applies to user interfaces as well). For this reason, you'll sometimes see Python variables with a leading _ in their name as a warning that these aren't part of the public interface.

Remember, program objects may represent real objects, but that does not make them real objects. They are models. One of the greatest gifts of modeling is the ability to ignore irrelevant details. The model car one of the authors built as a child looked like a real 1956 Thunderbird on the outside, but it obviously didn't run. When they were too young to drive, these details were overly complex and irrelevant. The model is an **abstraction** of a real concept.

Abstraction is another object-oriented term related to encapsulation and information hiding. Abstraction means dealing with the level of detail that is most appropriate for a given task. It is the process of extracting a public interface from the inner details. A car's driver needs to interact with the steering, accelerator, and brakes. The workings of the motor, drive train, and brake subsystem don't matter to the driver. A mechanic, on the other hand, works at a different level of abstraction, tuning the engine and bleeding the brakes. *Figure 1.6* (next page) shows two abstraction levels for a car.

Now, we have several new terms that refer to similar concepts. Let's summarize all this jargon in a couple of sentences: abstraction is the process of encapsulating information with a separate public interface. Any private elements can be subject to information hiding. In UML diagrams, we might use a leading - instead of a leading + to suggest that it's not part of a public interface.

The important lesson to take away from all these definitions is to make our models understandable to other objects that have to interact with them. This can mean paying careful attention to small details.

Ensure methods and properties have sensible names. When analyzing a system, objects typically represent nouns in the original problem, while methods are normally verbs. Attributes may show up as adjectives or more nouns. Name your classes, attributes, and methods accordingly.

When designing the interface, imagine you are the object; you want clear definitions of your responsibilities and you have a very strong preference for privacy to meet those responsibilities.

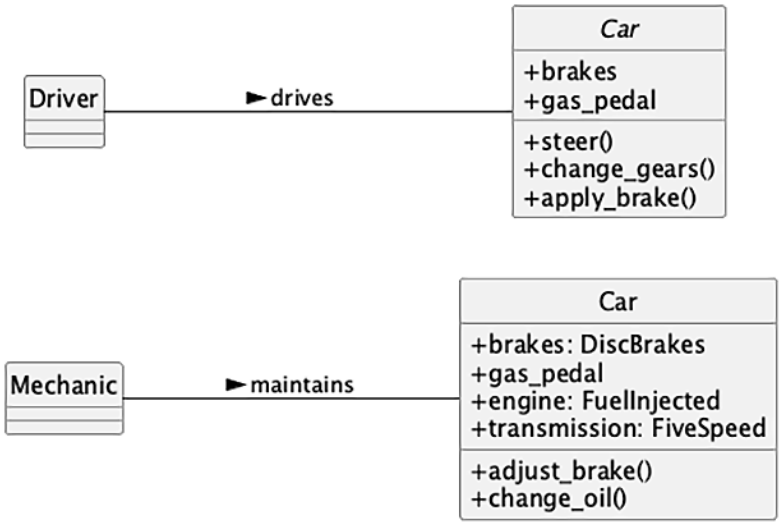


Figure 1.6: Abstraction levels for a car

Don't let other objects have access to data about you unless you feel it is in your best interest for them to have it. Don't give other classes an interface to force you to perform a specific task unless you are certain it's your responsibility to do that.

Design principles

Object-oriented design isn't easy. No design is particularly easy. There are a number of guiding principles that can help make decisions. One of the more famous sets of principles is called **SOLID**. This is a handy acronym for five ideas that can help transform a design from a tangle of loose threads into a tightly knit (and warm) garment.

We'll use these principles throughout the book. This is only a superficial introduction. The five principles are these:

1. **Single Responsibility Principle**
2. **Open/Closed Principle**
3. **Liskov Substitution Principle**
4. **Interface Segregation Principle**
5. **Dependency Inversion Principle**

These principles apply widely in an object-oriented design. We need to note that the Liskov substitution principle is focused on inheritance and the "is-a" relationship, something we've avoided in the previous examples.

The **SOLID** ordering is handy for remembering the principles, but it isn't the most useful way to understand them. We'll talk about them in a more practical sequence.

Interface Segregation Principle

We are talking about this principle first because it is essential for understanding the boundaries around a class definition. When wondering what to encapsulate, it helps to keep the interface as small as possible. When an object is too complicated, the interface can grow to reflect that complication, and the collaborating classes are forced to depend on methods and attributes that they don't actually need.

The goal, then, is to keep the interface small. This will minimize the intellectual burden of understanding the class. It will also ensure that other classes can evolve and change without disastrous problems stemming from unwanted (or unexpected) dependencies.

Open/Closed Principle

One of the key ingredients in a well-done design is class definitions that are open to extension but closed to modification. We want to be able to add features to a class using techniques such as inheritance and composition. We'll look at these design techniques closely in *Chapter 3*. We don't want to have to "tweak" the implementation code.

One aspect of this principle is relying on compositions of multiple objects to create complex behavior. This fits with the **Interface Segregation Principle** by separating features into distinct classes. We can extend one of those classes without the risk of breaking all the other classes in the application or library.

The other aspect is to create classes that **inherit** features from a base class. Via inheritance, the **subclass** is usable wherever the base class is expected, but it does something more specialized — more useful or appropriate — than the base class did. There's more to this inheritance idea, captured in the **Liskov Substitution Principle**.

Liskov Substitution Principle

This principle — named after Barbara Liskov, inventor of the one of the first object-oriented programming languages, **CLU** — offers advice to constrain how inheritance is used. We'll set the details aside for now to look at the overall goals of well-done design.

If we have a base class, such as **Container**, we want all of the subclasses, **Barrel**, **Basket**, and anything else we might need to invent, to have the same interface as the base class. They're all containers, each with unique implementation details. By having the same interface, any of the subclasses can be used in place of the base class.

When we're using tools such as **mypy** or **pyright** to check our type annotations, these tools will warn us of Liskov Substitution problems. The errors will pinpoint the places where a subclass interface doesn't match the promise made by the base class. We'll look at this in detail in *Chapter 7*.

Dependency Inversion Principle

The name for this principle is a little confusing: inverted with respect to what? If we don't know what the "right side up" is, how can we judge whether the dependency is "upside down"?

The easy, obvious dependency is to have one class explicitly name a class of objects with which it collaborates. This is easy and fun for tutorial examples and introductory programming classes. In the long run, however, when one class directly depends on another class, we have problems with making changes.

Imagine the Python code for the **Apple** class that directly names the **Barrel** class. This turns into an **Open/Closed Principle** problem. When we need to start shipping apples in large packing crates, the new **PackingCrate** class is an extension to the **Barrel** class. It was, in turn, an extension on some abstract base class, **Container**. We really don't want to edit a lot of code to add the new **PackingCrate**.

The idea of dependency "inversion" tells us that the **Apple** class should only name the base **Container** class. That way, any kind of container in the family tree can be associated with an **Apple** instance. The concrete relationship between the **Apple** class and the **Barrel** class should be something configured at runtime; it shouldn't be defined in the software at its foundation.

The idea of using the base class echoes the **Liskov Substitution Principle**. It helps implement the **Open/Closed Principle**. As we'll see later, this principle is a kind of implementation detail and helps make sure that the other principles are followed.

Single Responsibility Principle

This principle, given first, seems like it's really a summary of the others. A class will have a single responsibility.

To get to this ideal, we'll need to start by segregating the interfaces. Once we've decomposed a class into pieces to simplify the interfaces, we need to make sure to keep the design open to extension but closed to modification.

After these two initial steps, we need to review the details to make sure that Liskov Substitution will work. And, of course, we need to avoid "hard-wired" dependencies that require code modifications. To do this, we need to be flexible, inverting the dependencies so that the code depends only on base classes.

Once we've thought through our design, using these design principles, we'll find that our classes have a single, easy-to-summarize responsibility. We can then work on how the objects collaborate to create the desired software features.

This is — of course — only a sketch of the principles. Each has consequences and considerations that, well, fill this book.

Collaboration among objects

We've addressed two important ingredients that are part of object-oriented programming: class and responsibility. We classify objects based on their internal state and their behavior. We also strive to define a clear, focused responsibility for each class.

The remaining ingredient is *collaboration*. Once we decompose a problem into separate classes, the final application's behavior will emerge from collaboration among objects of those classes. The final application — like a good sauce — is a blend of ingredients that complement each other.

There are a variety of ways to think about collaboration among classes. We'll defer the details of different kinds of collaboration until *Chapter 3*. Here, we'll introduce two useful concepts:

- Composition
- Inheritance

When we think about following the **Interface Segregation Principle**, we often decompose something complicated into a group of simpler things. In many cases, we can design a model of the original (complicated) thing as a composition of the simpler things. We've already seen how composition works when talking about cars. A fossil-fueled car is composed of an engine,

transmission, starter, headlights, and windshield, among numerous other parts. Each of these can be further decomposed into the active, stateful component. The headlight subsystem, for example, is off, on, or bright. A control changes the state of this system. There may be an automated sensor to turn lights on at night, and dim the bright lights when there's oncoming traffic.

Decomposition exposes a potential problem. What if we have several things with common aspects? We have headlights, interior lights, and an entertainment system, all of which use fuses. Do we want to repeat the *depends on a fuse* aspect all over the class hierarchy? That sounds like a lot of copy-and-pasting, and potential nightmares when there are implementation changes.

To avoid repetition, we can use inheritance. It's helpful to think of **inheritance** as an "*is a*" relationship. We can extract a feature, such as *protected by a fuse*. Each of the electrical components within an automobile is a component protected by a fuse: the headlight system *is a* component protected by a fuse; the entertainment system *is a* component protected by a fuse.

We'll do this be defining a base class: `Fused`. Then, we'll define subclasses that extend this base class. Many folks use the term *superclass* instead of *base class*. The UML diagram often shows the base class at the top of the figure. The class is, however, foundational, and the other classes build on it.

The Liskov Substitution Principle (and the Open/Closed Principle) provides guidelines for inheritance. We need to be sure that each subclass has the same interface as the base class: the idea is that any subclass can replace the superclass.

When thinking about headlights and the entertainment system both having fuses, this can seem a little odd. We don't turn on the stereo when driving at night. (Well, maybe we do, but it doesn't help us see the pavement.)

The subclasses don't all do the same thing. They merely have a consistent interface. The headlights and the entertainment system both have an interface (a pair of wires) to the fuse and to the electrical ground. This interface fits the Liskov Substitution Principle.

When we add running lights to the headlight system, we are consistent with Liskov Substitution because the running lights use the same fuse. The use of long, flexible wires leaves the car's systems open to extension. The use of a tightly sealed headlamp fixture makes the lights closed to modification: if we want to make a change, we have to replace the whole fixture; we can't just push in a new bulb.

Now that we've started looking at object-oriented design, we'll take a quick look at some legacy

software that didn't follow object-oriented design principles. We can start thinking about ways to refactor it from a mess of statements to some easier-to-understand class definitions.

A potential mess

It's common for folks who know some Python but haven't extensively made use of the OOP features to wonder whether all the brain-calories that are burned really do lead to better software. We'll touch on a few questions (really, objections phrased as a questions) first. Then, we can look at a concrete example of restating a script as objects.

One common question is: "Isn't a class just a bunch of functions with shared data?" The short answer is "yes." The object-oriented feature that's important here is bundling the data and the related functions into a single namespace, called a class definition. When we write a batch of closely related functions, we often give them similar-looking names to be sure that the relationship is obvious. This is the purpose of a class: it provides a common container name for related functions.

Additionally, a class definition lets us create multiple instances of the shared data. This helps us encapsulate the processing for multiple objects with similar behavior but distinct states.

Another question is: "Why is a collection of class definitions easier to understand than one long function?" The short answer is "chunking." To keep complicated ideas in our heads, we break things into chunks. For example, we don't read a long number as a haphazard string of digits; we decompose it into blocks of digits. This is why we throw punctuation into things such as telephone numbers. In North America, we write "(111)222-3333" to break a 10-digit phone number into three small chunks. When we talk about an automobile's "interior" or "engine," we're decomposing the complicated whole into more intellectually manageable chunks.

A long script or a long function is generally hard to understand. The programmer will often break the long script into sections using comments. Sometimes, the comments are big billboards announcing major steps in the processing. Each of these sections could have been a smaller function. Smaller functions that are closely related often manage the state of a single object; these are methods of a class.

Reading a big script

Imagine a long Python script that summarizes details from a number of files in JSON format. It opens files, parses the JSON content, locates the details, and accumulates a summary. It does a lot of things, and reflects poorly managed complexity. Here's an outline of the code:

```
import json
from pathlib import Path
import shlex

def main():

    optional = {"type"}

    result_dir = Path.cwd() / "data"
    for file in result_dir.glob("*.json"):
        # 1. Load file
        result = json.loads(file.read_text())
        # 2. Set Outcome
        app_name = file.stem
        env_outcome = None
        # 3. Examine environments
        for env_name, env in result['testenvs'].items():
            # 2a. Skip special names
            if env_name.startswith("."):
                continue
            # 2b. Accumulate outcomes
            if env:
                if env['result']['success']:
                    if env_outcome is None:
                        env_outcome = "ok"
                else:
                    for step in env['test']:
                        if step['retcode'] != 0:
                            command = Path(step['command'][0]).stem
                            args = shlex.join(step['command'][1:])
                            message = f"{env_name} failed {command} {args}"
                            if env_outcome is None or env_outcome == "ok":
                                env_outcome = message
                            else:
                                env_outcome = f"{env_outcome}, {message}"
            else:
                if env_outcome is None:
                    env_outcome = f"{env_name} did not run"
                elif env_outcome == "ok" and env_name in optional:
                    env_outcome = f"ok (except {env_name})"
                else:
                    env_outcome = f"{env_outcome}, {env_name} did not run"
        # 4. Write summary
```

```
        print(f"{app_name:20s} {env_outcome}")
```

This script is just shy of 50 lines of code. Within this function, there are numerous shifts in focus: first the paths, then the JSON document on each path, then the environments that were tested, and then the commands that were executed. Ultimately, there are some complicated rules that define a final status that's printed. These shifts can make sense to the original author, but they are very hard for anyone else to grasp.

Further, of course, the complexity is quite difficult to test.

Buried in the clutter of processing details are a few essential ideas. This is for a suite of **application** instances, where each application is tested with the **tox** tool. The tool produces the JSON-formatted files with the details of the test outcomes for each application. (This tool is one of many available in the PyPI repository that are commonly added to projects to automate testing.) The tool will exercise each application in a number of **environments**. Each **environment** has a number of **commands**, using tools such as **pytest**, **pyright**, and **ruff**. An environment can have a result where the `success` attribute is true, meaning all the commands worked. Otherwise, at least one command failed.

Note that we started highlighting the key concepts that may need to be implemented as classes of objects: an **application**, several **environment** instances, and several **command** instances. A command, for example, has a JSON representation as a list of strings. An environment has a JSON representation as a simple string, `"3.13"`, with a dictionary of supporting details.

The script dives into details of the file, environment, and command. It's rare for a script like this to provide any sort of overview to help clarify the three varieties of outcomes for each application that's being tested:

- All commands in all environments were successful. The application is ready for deployment.
- A command failed in an environment. The application needs debugging.
- Something else went wrong and there's no JSON file at all. This also suggests the application isn't ready for deployment. Or, it may suggest something else is wrong with the entire test framework.

We have three classes of objects in the problem domain:

- An **application**, associated with one or more environments. The application is also associated with a summary that reduces the environment and command details to a final decision.

- An **environment**, associated with one or more commands. The environment object will have a summary of the commands, in the form of a status of success or failure.
- A **command**, which has details of each step performed. These are mostly interesting when they record a failure.

When looking at the script, we see a lot of navigation through JSON data structures. While this is an important implementation detail, it tends to obscure the overall objective of understanding applications and environments.

Note that it's common to gloss over some other categories of objects that are part of the implementation details:

- The **Path** object with a **glob()** method to locate all the files
- The **dict** objects created by the **json** module
- The **list** objects that contain the commands within an environment

When programming in Python, it helps to recognize that these implementation classes — the `pathlib.Path`, `dict`, and `list` classes — are the essence of object-oriented programming. These are classes we did not write, but we use them to create our applications. It turns out that parts of any Python script are already object-oriented, even if the script — as a whole — doesn't seem to have a robust design.

Revising the script permits us to emphasize the processing of applications, environments, and commands. You should consider ways to adjust this code to be object-oriented. We won't dive into these details. Instead, we'll look more broadly at the ideas behind refactoring throughout this book. We'll end the chapter having exposed the problem and a path toward a solution. There are two interrelated concepts here:

- Python is already object-oriented; the built-in types are all based on class definitions
- Good object-oriented design is a shift in focus from implementation details to the concepts behind the problem being solved

Recall

The following were some key points covered in this chapter:

- Analyzing problem requirements in an object-oriented context
- How to draw **UML** diagrams to communicate how the system works

- Discussing object-oriented systems using the correct terminology and jargon
- Understanding the distinction between class, object, attribute, and behavior

Exercises

This is a practical book. As such, we're not assigning a bunch of fake object-oriented analysis problems to create designs for you to analyze and design. Instead, we want to give you some ideas that you can apply to your own projects. If you have previous object-oriented experience, you won't need to put much effort into this chapter. However, they are useful mental exercises if you've been using Python for a while, but have never really cared about all that class stuff.

First, think about a recent programming project that you've completed. Identify the most prominent object in the design. Try to think of as many attributes for this object as possible. Did it have the following: Color? Weight? Size? Profit? Cost? Name? ID number? Price? Style?

Think about the attribute types. Were they primitives or classes? Were some of those attributes actually behaviors in disguise? Sometimes, what looks like data is actually calculated from other data on the object, and you can use a method to do those calculations. What other methods or behaviors did the object have? Which objects called those methods? What kinds of relationships did they have with this object?

Now, think about an upcoming project. It doesn't matter what the project is; it might be a fun free-time project or a multi-million-dollar contract. It doesn't have to be a complete application; it could just be one subsystem. Perform a basic object-oriented analysis. Identify the requirements and the interacting objects. Sketch out a class diagram featuring the highest level of abstraction on that system. Identify the major interacting objects. Identify minor supporting objects. Go into detail about the attributes and methods of some of the most interesting ones.

The goal is not to design a system right now (although you're certainly welcome to do so if inclination meets both ambition and available time). The goal is to think about object-oriented design from the perspective of class, responsibility, and collaboration among objects. It can help to focus on projects that you have worked on, or are expecting to work on in the future; this makes it less of a hypothetical exercise and more practical.

Lastly, visit your favorite search engine and look up some tutorials on UML. There are dozens, so find one that suits your preferred method of study. Sketch some class diagrams or a sequence diagram for the objects you identified earlier. Don't get too hung up on memorizing the syntax (after all, if it is important, you can always look it up again); just get a feel for the language. Something

will stay lodged in your brain, and it can make communicating a bit easier if you can quickly sketch a diagram for your next OOP discussion.

The UML diagrams in this book were prepared with the PlantUML tool. The documentation for this tool includes numerous example diagrams that can help show how to describe objects and their relationships. Some people like to use *Mermaid* (`https://mermaid.live/`) to create UML diagrams.

Summary

In this chapter, we took a whirlwind tour through the terminology of the object-oriented paradigm, focusing on object-oriented design. We can separate different objects into a taxonomy of different classes and describe the attributes and behaviors of those objects via the class interface. Encapsulation and information hiding are highly related concepts. Objects can be classified; they have responsibilities. The behavior of an application — as a whole — emerges from collaboration among objects. UML syntax can be both a useful and fun method of communication.

In the next chapter, we'll explore how to implement classes and methods in Python.

2

Objects in Python

We have a design in hand and are ready to turn that design into a working program! Of course, it doesn't usually happen this way. We'll be seeing examples and suggestions for good software design throughout the book, but our focus is on object-oriented programming. So, let's have a look at the Python syntax that allows us to create object-oriented software.

After completing this chapter, we will understand the following:

- Python's type hints
- Creating classes and instantiating objects in Python
- Using composition techniques to create more complicated objects
- Organizing classes into packages and modules
- Accessing class members wisely, including ways to suggest that collaborating objects don't clobber an object's internal state
- Working with third-party packages available from the Python Package Index, **PyPI**
- Managing your virtual environments

Technical requirements

The code for this chapter can be found in the `PacktPublishing` repository: `https://github.com/PacktPublishing/Python-Object-Oriented-Programming-5E`. Within that repository's files, we'll focus on the `ch_02` directory.

This chapter will use the **mypy** tool, which is installed separately. Commands such as `python -m pip install mypy` will install this. If you're using **uv** to manage your environment, then `uvx tool install mypy` will add mypy.

All of the examples were tested with Python 3.12 and 3.13. The **uv** tool can be used to test the code: `uvx tox`.

Introducing types and classes

Before we can look closely at creating classes, we need to talk a little bit about what a class is and how to be sure we're using it correctly. One central idea is everything in Python is an object.

When we write literal values such as `"Hello, world!"` or `42`, we're actually creating objects that are instances of built-in classes. (Some languages have "primitive types" which aren't objects; Python doesn't have this complication.) We can fire up interactive Python and use the built-in `type()` function on the class that defines the properties of these objects:

```
>>> type("Hello, world!")
<class 'str'>
>>> type(42)
<class 'int'>
```

The point of *object-oriented* programming is to solve a problem via a collaboration of objects. When we write `6 * 7`, the multiplication of the two objects is handled by a method of the built-in `int` class. For more complex behaviors, we'll often need to write unique, new classes.

Here are the first two core rules of how Python objects work:

- Everything in Python is an object
- Every object is defined by being an instance of at least one class

These rules have many interesting consequences. A class definition that we write using the `class` statement creates a new object of class `type`. When we create an **instance** of a class, the resulting class object will be used to first create and then initialize the instance object being created.

What's the distinction between class and type? The `class` statement lets us define new types. (Yes, that is the way it works.) Because the `class` statement is what we use, we'll call them classes throughout the text. See *Python objects, types, classes, and instances — a glossary* by `Eli Bendersky`, `https://eli.thegreenplace.net/2012/03/30/python-objects-types-classes-and-instances-a-glossary`,
for this useful quote:

> *The terms "class" and "type" are an example of two names referring to the same concept.*

For type hints, there's a similar common-usage principle, not as clearly articulated. The terms hints and annotations are — essentially — the same concept. We'll often follow common usage and call the annotations **type hints**.

There's another important rule:

- A variable is a reference to an object. Think of a yellow sticky note, with a name scrawled on it, slapped on a thing.

This doesn't seem too earth-shattering but it's actually pretty cool. It means the type information — what an object is — is defined by the class(es) associated with the object. This type information is not attached to the *variable* in any way. This leads to code like the following being both valid and confusing Python:

```
>>> a_string_variable = "Hello, world!"
>>> type(a_string_variable)
<class 'str'>
>>> a_string_variable = 42
>>> type(a_string_variable)
<class 'int'>
```

We created an object using a built-in class, `str`. We assigned a long name, `a_string_variable`, to the object. Then, we created an object using a different built-in class, `int`. We assigned this new object the original name.

Figure 2.1 shows two steps, side by side, to illustrate how the variable is moved from object to object:

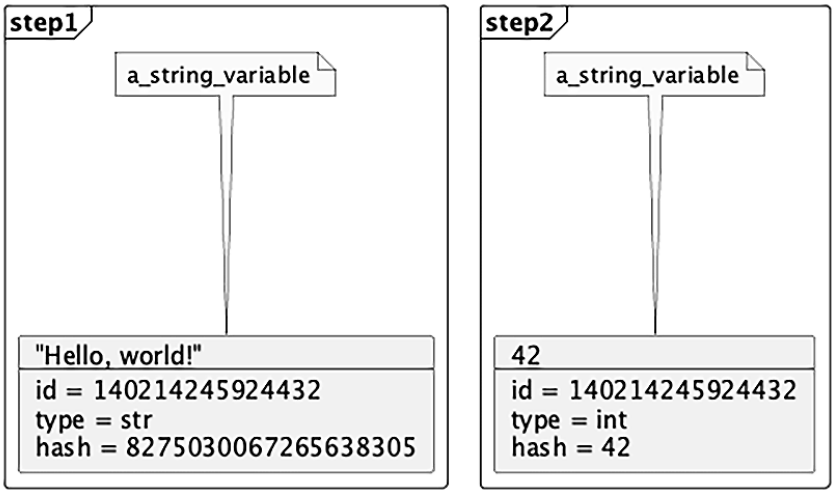


Figure 2.1: Variable names and objects

The various property values are part of the object, not the variable. When we check the type of a variable with `type()`, we see the type of the object the variable currently references. A variable doesn't have a type of its own;it's nothing more than a name. This means a function isn't defined by the parameter types or the return type; it's only a name that's bound to a "callable" object. (Callable objects include functions and methods of classes, along with a few other things we'll address in *Chapter 8* and *Chapter 11*.) Similarly, asking for the `id()` of a variable shows the ID of the object the variable refers to. Clearly, the name `a_string_variable` is a highly misleading if we assign the name to an integer object.

Further, tools such as **mypy** and **pyright** will have trouble working out whether or not the variable is used properly. Consistency is an important part of clarity.

We'll show type hints in most of the examples. We'll defer details on hints and how to check them until *Chapter 7*.

If you've never seen them before, here's how type hints look when we write them in a function:

```
def odd(n: int) -> bool:
    return n % 2 != 0
```

The hints suggest the argument value for the `n` parameter should be an integer. They also suggest the result will be one of the two values of the `bool` type.

Annotation syntax is relatively easy to understand. We can follow a variable name with a colon, `:`, and a type. We can do this in the parameters to functions (and methods). We can also do this in assignment statements. Further, when defining functions, we can also add `->`*type* to the definition to explain the expected return type.

These annotations have no runtime impact. Because Python politely ignores annotations, they're optional. People reading your code, however, will be more than delighted to see them. They are a great way to inform the reader of your intent. You can omit them while you're learning, but you'll love them when you go back to expand something you wrote earlier.

Hint-checking tools such as **pyright** and **mypy** can analyze the type hints to locate places where the hints are not used properly. This is an essential part of the development cycle, as important as writing test cases. These tools for checking hints are not built into Python, and require a separate download and install. The more sophisticated Integrated Development Environments (**IDEs**) include some type-checking tools. We'll talk about virtual environments and installation of tools in the *Third Party Libraries* section.

For now, it's helpful to be familiar with the syntax for annotations. Most of the examples in this book will have type hints because they help make the intent behind the example clear. Now that we've talked about how parameters and attributes are described with type hints, let's actually build some classes.

Creating Python classes

We don't have to write much Python code to realize that Python is a very *clean* language. When we want to do something, we can just do it, without having to set up a bunch of prerequisite code. The ubiquitous *hello world* in Python, as you've likely seen, is only one line.

Similarly, the simplest class in Python 3 looks like this:

```
class MyFirstClass:
    pass
```

There's our first object-oriented program! For more information on the syntax, see *section 9.3.1* (`https://docs.python.org/3/tutorial/classes.html#class-definition-syntax`) of the Python Tutorial.

The class name must follow standard Python variable naming rules: it must start with a letter or underscore, and can only be comprised of letters, underscores, or numbers. In addition, the Python style guide *PEP 8*: (`https://peps.python.org/pep-0008/`) recommends classes should be named using what PEP 8 calls **CapWords** notation: start with a capital letter; any subsequent words should also start with a capital. Also, in line with the style guide, use four spaces for indentation unless you have a compelling reason not to (such as fitting in with somebody else's code that uses tabs for indents).

Since our first class doesn't actually add any data or behaviors, we uses the `pass` statement on the second line. This is a placeholder to fill the requirement for a class body; it indicates that no further action needs to be taken.

We might think there isn't much we can do with this most basic class, but it does allow us to instantiate objects of that class. We can load the class into the Python 3 interpreter, so we can interactively play with it. To do this, save the class definition mentioned earlier in a file with a name such as `firs_class.py` and then run the following command:

```
% python -i src/first_class.py
```

The `-i` argument tells Python to *run the code and then drop to the interactive interpreter.* The following interpreter session demonstrates a basic interaction with this class:

```
>>> a = MyFirstClass()
>>> b = MyFirstClass()
>>> print(a)
<first_class.MyFirstClass object at ...>
>>> print(b)
<first_class.MyFirstClass object at ...>
```

This code instantiates two objects from the new class, assigning the object variable names `a` and `b`. Creating an instance of a class is a matter of typing the class name, followed by a pair of parentheses. It looks much like a function call; **calling** a class will create a new object. When printed, the two objects tell us which class they are and what memory address they live at. We've replaced the two memory addresses with `...` because they're always different. Generally, they're hexadecimal numbers such as `0xb7b7fbac`. Memory addresses aren't used much in Python code, but here, they demonstrate that there are two distinct objects involved, and they leave at two distinct memory addresses.

We can see they're distinct objects by using the `is` operator:

```
>>> a is b
False
```

This can help reduce confusion when we've created a bunch of objects and assigned different variable names to the objects.

Adding attributes

Now, we have a basic class, but it's fairly useless. It doesn't contain any data, and it doesn't do anything. What do we have to do to assign an attribute to a given object?

In fact, we don't have to do anything special in the class definition to be able to add attributes. We can set arbitrary attributes on an instantiated object using dot notation. Here's an example:

```
class Point:
    pass

p1 = Point()
p2 = Point()
p1.x = 5
p1.y = 4
p2.x = 3
p2.y = 6
print(p1.x, p1.y)
print(p2.x, p2.y)
```

If we run this code, the two `print` lines at the end tell us the new attribute values on the two objects:

```
5 4
3 6
```

This code created an empty `Point` class with no data or behaviors. Then, it created two instances of that class and assigns each of those instances `x` and `y` coordinates to identify a point in two dimensions. All we need to do to assign a value to an attribute on an object is use the `<object>.<attribute> = <value>` syntax. This is sometimes referred to as **dot notation**. The value can be anything: a Python primitive, a built-in data type, or another object. It can even be a function or another class!

Creating attributes like this can be confusing to people trying to read your code. It also confuses tools used to inspect code. There's a much, much better approach to attributes (and their type hints) that we'll examine in *Initializing the object*, later in this chapter. First, though, we'll add behaviors to our class definition.

Making it do something

Having objects with attributes is a great start. Object-oriented programming is about the interaction between objects. We're interested in invoking actions that cause things to happen to those attributes. We have data; now it's time to add behaviors to our classes.

Let's model a couple of actions on our `Point` class. We can start with a **method** called `reset`, which moves the point to the origin (the origin is the place where `x` and `y` are both zero). This is a good introductory action because it doesn't require any parameters:

```
class Point:
    def reset(self) -> None:
        self.x = 0
        self.y = 0

p = Point()
p.reset()
print(p.x, p.y)
```

This `print` statement shows us the two zero values of the attributes:

```
0 0
```

In Python, a method is formatted identically to a function. For more information on the syntax, see *section 9.3.4* (`https://docs.python.org/3/tutorial/classes.html#method-objects`) of the Python Tutorial.

The `self` parameter is essential. We'll discuss that `self` parameter, sometimes called the instance variable, in just a moment.

A function that doesn't return a value explicitly will implicitly return the `None` object. We formalized this feature of Python by providing `-> None` as part of the type annotations for the method.

Next, we'll look a little more at instance variables and how the `self` parameter works.

Talking about yourself

The one difference, syntactically, between methods of classes and functions outside classes is that methods have one required argument. This argument is conventionally named `self`; we've never seen a Python programmer use any other name for this variable (convention is a very powerful thing). There's nothing technically stopping you, however, from calling it `this` or even `Martha`, but it's best to acknowledge the social pressure of the Python community codified in PEP 8 and stick with `self`.

The `self` argument to a method is a reference to the object that the method is being invoked on. The object is an instance of a class, and that's why this is often called the instance variable.

We can access attributes and methods of that object via this variable. This is exactly what we do inside the `reset` method when we set the `x` and `y` attributes of the `self` object.

Pay attention to the difference between a **class** and an **object** in this discussion. We can think of the **method** as a function attached to a class. The `self` parameter refers to a specific instance of the class. When you call the method on two different objects, you are passing two different **objects** as the `self` parameter.

Notice that when we call the `p.reset()` method, we do not explicitly pass the `self` argument to it. Python automatically takes care of this part for us. It knows we're calling a method on the `p` object, so it automatically passes that object, `p`, to the method of the class, `Point`.

For some, it can help to think of a method as a function that happens to be part of a class. Instead of calling the method on the object, we could invoke the function as defined in the class, explicitly passing our object as the `self` argument:

```
>>> p = Point()
>>> Point.reset(p)  # Works, but...
>>> print(p.x, p.y)
0 0
```

The output is the same as in the previous example because, internally, the exact same process has occurred. We've flagged the line with a comment of `works, but...`; while this works, it's not the best practice. While it's useful in a few cases, we've empahsized it because it can help cement an understanding of the `self` argument.

What happens if we forget to include the `self` argument in our class definition? Python will bail with an error message, as follows:

```
>>> class Point:
...     def reset():
...         pass
...
>>> p = Point()
>>> p.reset()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Point.reset() takes 0 positional arguments but 1 was given
```

The error message is not as clear as it could be ("Hey, silly, you forgot to define the method with a `self` parameter" could be more informative). Just remember that when you see an error message that indicates missing arguments, the first thing to check is whether you forgot the `self` parameter in the method definition.

More arguments

How do we pass multiple arguments to a method? Let's add a new method that allows us to move a point to an arbitrary position, not just to the origin. We can also include a method that accepts another `Point` object as input and returns the distance between them:

```
from __future__ import annotations
import math


class Point:
    def move(self, x: float, y: float) -> None:
        self.x = x
        self.y = y

    def reset(self) -> None:
        self.move(0.0, 0.0)

    def calculate_distance(self, other: Point) -> float:
        return math.hypot(self.x - other.x, self.y - other.y)
```

We've defined a class with two attributes, `x` and `y`, and three separate methods, `move()`, `reset()`, and `calculate_distance()`.

The `move()` method accepts two arguments, `x` and `y`, and sets the relevant attributes of the `self` object. The `reset()` method calls the `move()` method, since a reset is just a move to a specific

known location.

The `calculate_distance()` method computes the Euclidean distance between two points. The distance computation is $\sqrt{(x_s - x_o)^2 + (y_s - y_o)^2}$, which is the `math.hypot()` function. In Python we'll use `self.x`, but mathematicians often prefer to write x_s.

The type name, `Point`, for the `other` parameter, is a reference to the class definition this method is a portion of. The class isn't completely defined, and the `Point` class definition isn't fully available inside any of the code inside the `Point` class. Prior to PEP 749, a string `"Point"` could be used to refer to a class that's not defined yet. With the `from __future__` import of the `annotations` package, we can include a reference to the `Point` class without the potentially confusing use of a string. Starting with Python 3.14, the `from __future__ import` will no longer be required to make kinds of references work as expected. If you haven't seen it before, the `__future__` package can offer some features before they become a standard part of the language. For now, we'll use it heavily.

Here's an example of using this class definition. This shows how to call a method with arguments: include the arguments inside the parentheses and use the same dot notation to access the method name within the instance. We just picked some random positions to test the methods. The test code calls each method and prints the results on the console:

```
>>> point1 = Point()
>>> point2 = Point()

>>> point1.reset()
>>> point2.move(5, 0)
>>> print(point2.calculate_distance(point1))
5.0
>>> assert point2.calculate_distance(point1) == point1.calculate_distance(
...     point2
... )
>>> point1.move(3, 4)
>>> print(point1.calculate_distance(point2))
4.47213595499958
>>> print(point1.calculate_distance(point1))
0.0
```

The `assert` statement is a marvelous test tool; the program will bail if the expression after `assert` evaluates to `False` (or zero, empty, or `None`). In this case, we use it to ensure that the distance is the

same regardless of which point called the other point's `calculate_distance()` method. We'll see a lot more use of `assert` in *Chapter 13*, where we'll write more rigorous tests.

Initializing the object

If we don't explicitly set the `x` and `y` positions on our `Point` object, either using `move` or by accessing them directly, we'll have a broken `Point` object with no real position. What will happen when we try to access it?

Well, let's just try it and see. *Try it and see* is an extremely useful tool for Python study. Open up your interactive interpreter and type away. (Using the interactive prompt is, after all, one of the tools we used to write this book.)

The following interactive session shows what happens if we try to access a missing attribute:

```
>>> point = Point()
>>> point.x = 5
>>> print(point.x)
5
>>> print(point.y)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Point' object has no attribute 'y'
```

Well, at least it threw a useful exception. We'll cover exceptions in detail in *Chapter 4*. You've probably seen them before (especially the ubiquitous `SyntaxError`, which means you typed something incorrectly!).

The output is useful for debugging. It tells us the error occurred at *line 1*, which is only partially true (in an interactive session, only one statement is executed at a time). If we were running a script in a file, it would tell us the exact line number, making it easy to find the offending code. In addition, it tells us that the error is an `AttributeError` error and gives a helpful message telling us what that error means.

We can catch and recover from this error, but in this case, it feels like we should have specified some sort of default value. Perhaps every new object should be `reset()` by default. Maybe it would be nice if we could force the user to tell us what those positions should be when they create the object.

Interestingly, annotation-checking tools such as **mypy** can't determine whether `y` was supposed to be an attribute of a `Point` object. Attributes are — by definition — dynamic, so there's no simple list

that's part of a class definition. However, Python has some widely followed conventions that can help name the expected set of attributes.

Most object-oriented programming languages have the concept of a **constructor**, a special method that creates and initializes the object when it is created. Python is a little different; it has an initializer. A constructor method, `__new__()`, is rarely used unless you're doing something very exotic. We'll look at some elements of this lower level of class design in *Chapter 6*. We'll start our discussion with the widely-used initialization method, `__init__()`. This is sometimes called a constructor for the instance, because it constructs the initial state of the object.

The Python initialization method is the same as any other method, except it has a special name, `__init__`. The leading and trailing double underscores mean this is a special method that the Python interpreter will treat as a special case.

Never name a method of your own with leading and trailing double underscores. It may mean nothing to Python today, but there's always the possibility that the designers of Python will add a function that has a special purpose with that name in the future. When they do, your code will break.

Let's add an initialization function on our `Point` class. This will require the user to supply `x` and `y` coordinates when the `Point` object is instantiated:

```
from __future__ import annotations
import math


class Point:
    def __init__(self, x: float, y: float) -> None:
        self.move(x, y)

    def move(self, x: float, y: float) -> None:
        self.x = x
        self.y = y

    def reset(self) -> None:
        self.move(0, 0)

    def calculate_distance(self, other: Point) -> float:
        return math.hypot(self.x - other.x, self.y - other.y)
```

Constructing a `Point` instance now looks like this:

```
>>> point = Point(3, 5)
>>> print(point.x, point.y)
3 5
```

Now, our `Point` object can never go without both `x` and `y` coordinates! If we try to construct a `Point` instance without including the proper initialization parameters, it will fail with a `TypeError` exception:

```
>>> point = Point()
Traceback (most recent call last):
...
TypeError: Point.__init__() missing 2 required positional arguments: 'x' and
'y'
```

Most of the time, we put our initialization statements in the `__init__()` function. It's very important to be sure that all of the attributes are initialized in the `__init__()` method. Doing this helps annotation-checking tools such as **mypy** by providing all of the attributes in one obvious place. More than that, it helps people reading your code; it saves them from having to read the whole application to find mysterious attributes set outside the class definition.

While they're optional, it's generally helpful to include type annotations on the method parameters and result values. After each parameter name, we've included the expected type of each value. At the end of the definition, we've included the two-character `->` operator and the type returned by the method.

Type hints and defaults

As we've noted a few times now, hints are optional. They don't do anything at runtime. There are tools, however, that can examine the hints to check for consistency. The **mypy** and **pyright** tools are widely used to check type hints.

If we don't want to make these two arguments required, we use the syntax Python functions use to provide default argument values. If the calling object does not provide this argument, then the default argument is used instead. The variables will still be available to the function, but they will have the values specified in the argument list. Here's an example:

```
class Point:
    def __init__(self, x: float = 0.0, y: float = 0.0) -> None:
        self.move(x, y)
```

The definitions when there are more than two or three parameters can get long, leading to very long lines of code. In some examples, you'll see this single logical line of code expanded to multiple physical lines. This relies on the way Python combines physical lines to match `()`s. We might write this when the line gets long:

```
class Point:
    def __init__(
        self,
        x: float = 0.0,
        y: float = 0.0
    ) -> None:
        self.move(x, y)
```

This style isn't used very often, but it's valid and keeps the lines shorter and easier to read.

The type hints and defaults are handy, but there's even more we can do to define a class that's easy to use and easy to extend when new requirements arise. We'll add documentation in the form of docstrings.

Explaining yourself with docstrings

Python can be an extremely easy-to-read programming language; some might say it is self-documenting. However, when carrying out object-oriented programming, it is important to write API documentation that clearly summarizes what each object and method does. Keeping documentation up to date is difficult; the best way to do it is to write it right into our code.

Python supports this through the use of **docstrings**. Each class, function, or method header can have a Python string as the first line of the indented suite of statements.

Often, docstrings are quite long and span multiple lines (the style guide suggests that the line length should not exceed 80 characters). This suggests using multi-line strings, enclosed in matching triple apostrophe (`'''`) or triple quote (`"""`) characters.

A docstring should summarize the purpose of the class or method it is describing. It should explain any parameters whose usage is not immediately obvious, and is also a good place to include short

examples of how to use the class. Any caveats or problems an unsuspecting user of the class should be aware of should also be noted. The interface to a class — the collection of methods and attributes — needs to be considered as carefully as the responsibilities of the class.

One of the best things to include in a docstring is a concrete example. Tools such as **doctest** can locate and confirm that these examples are correct. All the examples in this book are checked with the doctest tool.

To illustrate the use of docstrings, we will end this section with our completely documented `Point` class. We'll break it into two parts; here's the first:

```
class Point:
    """
    Represents a point in two-dimensional geometric coordinates

    >>> p_0 = Point()
    >>> p_1 = Point(3, 4)
    >>> p_0.calculate_distance(p_1)
    5.0
    """

    def __init__(self, x: float = 0.0, y: float = 0.0) -> None:
        """
        Initialize the position of a new point. The x and y
        coordinates can be specified. If they are not, the
        point defaults to the origin.

        :param x: float x-coordinate
        :param y: float x-coordinate
        """
        self.move(x, y)
```

Here's the rest of the definition:

```
    def move(self, x: float, y: float) -> None:
        """
        Move the point to a new location in 2D space.

        :param x: float x-coordinate
        :param y: float x-coordinate
        """
        self.x = x
```

```
        self.y = y

    def reset(self) -> None:
        """
        Reset the point back to the geometric origin: 0, 0
        """
        self.move(0.0, 0.0)

    def calculate_distance(self, other: Point) -> float:
        """
        Calculate the Euclidean distance from this point
        to a second point passed as a parameter.

        :param other: Point instance
        :return: float distance
        """
        return math.hypot(self.x - other.x, self.y - other.y)
```

Try typing or loading this file (using `python -i src/point_4.py`) into the interactive interpreter. Then, enter `help(Point)` at the Python prompt.

You should see nicely formatted documentation for the class, as shown in the following output. It's long, so we've broken into several parts to fit within a book.

Here's the first part of the help output:

```
>>> help(Point)
Help on class Point in module point_4:
<BLANKLINE>
class Point(builtins.object)
 |  Point(x: 'float' = 0.0, y: 'float' = 0.0) -> 'None'
 |
 |  Represents a point in two-dimensional geometric coordinates
 |
 |  >>> p_0 = Point()
 |  >>> p_1 = Point(3, 4)
 |  >>> p_0.calculate_distance(p_1)
 |  5.0
```

Here's the second part:

```
 |  Methods defined here:
 |
 |  __init__(self, x: 'float' = 0.0, y: 'float' = 0.0) -> 'None'
 |      Initialize the position of a new point. The x and y
 |      coordinates can be specified. If they are not, the
 |      point defaults to the origin.
 |
 |      :param x: float x-coordinate
 |      :param y: float x-coordinate
 |
 |  calculate_distance(self, other: 'Point') -> 'float'
 |      Calculate the Euclidean distance from this point
 |      to a second point passed as a parameter.
 |
 |      :param other: Point instance
 |      :return: float distance
 |
 |  move(self, x: 'float', y: 'float') -> 'None'
 |      Move the point to a new location in 2D space.
 |
 |      :param x: float x-coordinate
 |      :param y: float x-coordinate
 |
 |  reset(self) -> 'None'
 |      Reset the point back to the geometric origin: 0, 0
```

Here's the last bit:

```
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables
 |
 |  __weakref__
 |      list of weak references to the object
```

Not only is our documentation every bit as polished as the documentation for built-in functions, but we can run the following command to confirm the example shown in the docstring:

```
% python -m doctest src/point_4.py
```

Interestingly, if everything works, there's no output. By default, the **doctest** tool only produces output when a test fails. To see more verbose output, add the `-v` option.

Further, we can run a type-checking tool such as **mypy** to check the type hints. Using a command uch as the following in a terminal window will check all the files in the `src` folder:

```
% mypy src
```

(It's important to note that this is not a Python statement, used at the Python REPL prompt of `>>>`. This is an OS command entered in a terminal or command window. We try to use the `%` prompt consistently to show this.)

If you're using **uv** to manage your environment, then `uvx tool run mypy src` can be used to run **mypy**, leaving the details of downloading and installing the tool to **uv**.

The code repository for the book includes a `pyproject.toml` file that provides the **mypy** options to exclude two files that show some the known errors that are part of this chapter.

When there are no problems, the output is very short:

```
% mypy src
Success: no issues found in 22 source files
```

In the next sections, we'll turn to some additional details of object composition and then we'll turn to the idea of encapsulation.

Composition and decomposition

To see composition in action, we'll look at a few, isolated elements of the design of a chess game.

A *game* of chess is **played** between two *players*, using a chess set featuring a *board* containing 64 *positions* in an 8×8 grid. The board can have two sets of 16 *pieces* that can be **moved** in alternating *turns* by the two players in different ways. Each piece can **capture** other pieces. The board will be required to **draw** itself on the computer *screen* after each turn.

We've identified some of the possible objects in the description using *italics*, and a few key methods using **bold**. This is a common first step in turning an object-oriented analysis into a design. At this point, to emphasize composition, we'll focus on the board, without worrying too much about the players or the different types of pieces.

The chess set is composed of a board and 32 pieces. The board further comprises 64 positions. The positions are commonly identified by file (a-h) and rank (1-8). This means the white king commonly starts in position "e1". The black queen starts in position "d8".

Figure 2.2 is a class diagram showing the `Board` class as a composition of `Position` instances.

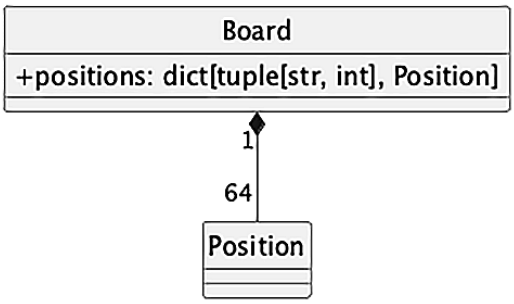


Figure 2.2: Class diagram for a chess board

We've omitted any details of the `Position` class for now.

The Python definitions for these classes might start like this:

```
class Position:
    def __init__(self, file: str, rank: str) -> None:
        self.file = file
        self.rank = rank

class Board:
    def __init__(self) -> None:
        self.positions: dict[tuple[str, str], Position] = {}
        for file in ('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'):
            for rank in ('1', '2', '3', '4', '5', '6', '7', '8'):
                self.positions[file, rank] = Position(file, rank)
```

The `__init__()` method created the 64 `Position` objects and assigned them to the `Board` object.

Note that we're using string values for the ranks, not numbers. This makes it very slightly easier to create a string describing a position on the board.

We have to observe an important shift of focus here. In a very technical sense, the `Board` is composed of one thing, a dictionary. The dictionary is composed of 64 `Position` objects. (And, don't forget, 64 two-tuples with the rank-file strings for the dictionary.) There's a lot of composition going on in

this seemingly simple composite object: dictionaries, strings, and `Position` instances.

We have two levels of detail:

- The implementation details involving `dict` and `str`
- The problem domain, which is a `Board` and `Position` instances

It's common to take a step back from the implementation details. We want to leave some room for the possibility of changing the implementation to use integer rank numbers. We might want have the board composed of a list of eight objects, one for each file. Each file would be a list of eight `Position` objects.

The idea that a `Board` is composed of `Position` instances is essential for clear communication about the software being designed. The implementation details of the dictionary or list-of-lists are less important. In some cases, we may need to provide both diagrams. Often, we'll omit the implementation details.

What happens when the `Board` object is no longer being used? What about the Position objects associated with the board? When the Board is no longer in use, all 64 `Position` objects are likewise no longer needed. This is the "a composite controls the composition" rule. There's an alternative to **composition**, called **aggregation**.

Figure 2.3 looks at an expanded version of the chess game. This diagram includes the pieces.

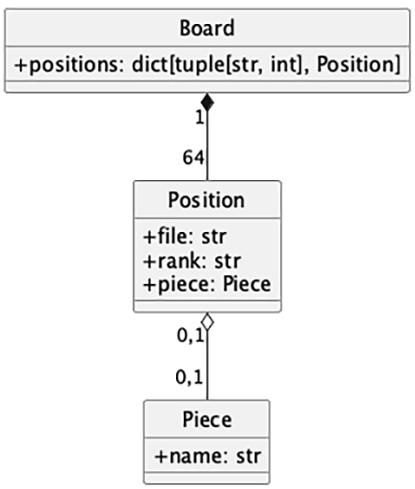


Figure 2.3: Class diagram for a chess board and pieces

Notice that we've introduced a new arrow-head style at the end of the association: an open diamond. The `Board` is composition of `Position` objects, with a filled-in diamond. A `Piece` is an aggregate object that may have a `Position` associated with it. (After a piece is captured, it will not have a position. Also, when a piece is moved, it leaves one position and arrives at another.)

The open diamond for an aggregation tells the reader that a `Piece` and `Position` can exist independently of each other. If we remove a `Piece` from play (because it's captured) the `Position` continues to exist. The `Position` remains as part of the `Board` composition.

The `Board`, on the other hand, must be composed of `Position` objects; a `Position` must be part of a `Board`. This relationship is essential.

In practice, the distinction between aggregation and composition is often irrelevant past the design stage. In a relational database, for example, there are explicit deletes as well as an explicit "cascading delete" relationship; the distinction between composition and aggregation becomes part of the SQL code. When implemented in Python, however, both composition and aggregation look the same. This is because object deletion happens automatically in Python.

An important part of object composition is how the objects collaborate. A very important part of this is having clear definitions of responsibility. This means keeping information encapsulated within a class, something we'll look at next.

Who can access my data?

Object-oriented programming languages have a concept of **access control**. This is related to the concept of encapsulation. Some languages have a spectrum of access controls including private, protected, public, and final.

Python doesn't do this. Instead, Python is kept very simple, and provides some guidelines and best practices. All methods and attributes on a class are publicly available. We often remind each other of this by saying "We're all adults here." There's no need to declare a variable as *private* or *protected* when we can all see the source code.

If we want to suggest that a method should not be used publicly, we really need to put a note in docstrings indicating that the method is meant for internal use only. Ideally, we include an explanation of how the public-facing API works. We often supplement this with examples copied and pasted from REPL interaction; examples that can be tested by the *doctest* module.

By convention, we prefix a non-public attribute or method with an underscore character, _. Python programmers will understand a leading underscore name to mean *this is an internal variable, think three times before accessing it directly.* But there is nothing inside the interpreter to stop them from accessing it if they think it is in their best interest to do so. It's a pretty clear warning sign to avoid using it. Most lint-checking tools will warn us when we use a non-public attribute or method.

There's another thing available to strongly suggest that outside objects don't access a property or method: prefix it with a double underscore, __. This will perform **name mangling** on the attribute in question. In essence, name mangling means that the method can still be called by outside objects if they really want to do so, but it requires extra work and is a strong indicator that you demand that your attribute remains **private**.

The mangling process isn't too complicated. The attribute's name is (secretly) prefixed with `_<classname>`. When methods inside the class internally access the attribute, the references are consistently mangled. When external classes wish to access this attribute, the attribute name isn't mangled, and the author of the external class has to mangle the name manually. Name mangling does not guarantee privacy; it only strongly recommends it. This is very rarely used, and is often a source of confusion when it is used.

Don't fret over other language's complicated rules for public, protected, private, and final attributes.

Don't use double-underscore names in your own code, it will only cause grief and heartache.

Use non-public (single-underscore) names to mark implementation details and features subject to change.

What's important is that encapsulation — as a design principle — assures that the methods of a class encapsulate the state changes for the attributes. Whether or not attributes (or methods) are private doesn't change the essential good design that flows from encapsulation.

The encapsulation principle applies to individual classes as well as a module with a bunch of classes. It also applies to a package with a bunch of modules. As designers of object-oriented Python,we're isolating responsibilities and clearly encapsulating features.

Classes are not the only kind of organization for our code. We also have modules and packages, something we'll look at next.

Modules and packages

Now we know how to create classes and instantiate objects. You don't need to write too many classes (or non-object-oriented code, for that matter) before you start to lose track of them. For small programs, we generally put all our classes into one file and add a little script at the end of the file to start them interacting. However, as our projects grow, it can become difficult to find the one class that needs to be edited among the many classes we've defined. This is where **modules** come in. Modules are Python files, nothing more. The single file in our small program is a module. Two Python files are two modules. If we have two files in the same folder, we can load a class from one module for use in the other module.

The Python module name is the file path's *stem*; the name without the `.py` suffix. A file with the name `model.py` is a module named `model`. Module files are found by searching paths that includes the local directory, the installed packages, and the standard library.

The `import` statement is used for importing modules or specific classes or functions from modules. We've already seen an example of this in our `Point` class in the previous section. We used the `import` statement to get Python's built-in `math` module and use its `hypot()` function in the `distance` calculation.

There are several variations on the `import` statement syntax that can be used to access parts of a module. One variant is to import the module as a whole:

```
import random

def dice1() -> tuple[int, int]:
    return (
        random.randint(1, 6), random.randint(1, 6)
```

This version imports the `random` module, creating a `random` namespace. All classes or functions in the `random` module are accessed using the `random.`*something* notation.

Alternatively, we can import just the one class we need using the `from...import` syntax:

```
from random import randint

def dice2() -> tuple[int, int]:
    return (
        randint(1, 6), randint(1, 6)
    )
```

This version imported only the `randint()` function from the `random` module. When we have a few items from a few modules, this can be a helpful simplification to avoid using longer, fully qualified names such as `random.randint`. When we import a number of items from a array of different modules, this can be a potential source of confusion because we can omit the qualifiers that show where a object originated.

If, for some reason, our application already has a class called `randint`, and we don't want the two names to be confused, we can rename the object during import:

```
from random import randint as rng

def dice3() -> tuple[int, int]:
    return (
        rng(1, 6), rng(1, 6)
    )
```

We can also import multiple items in one statement. If we need the `Random` class as well as the `seed()` function from the `random` module, we can import both objects using the following code:

```
from random import seed, Random
```

We can import all classes and functions from the `random` module using this syntax:

```
from random import *
```

While this works, **don't do this.** Most experienced Python programmers will tell you that you should never use this syntax (a few will tell you there are some very specific situations where it is useful, but we can disagree). Most tools that look for those lint-like bits of fuzz in your code that might catch fire (tools such as **ruff**) will warn you against this. One way to learn why to avoid this syntax is to use it and try to understand your code two years later. We can save some time — and two years of poorly written code — with a quick explanation now!

We've got several reasons for avoiding the `from X import *` syntax:

- When we explicitly import the `random` class at the top of our file using `from random import randint`, we can easily see where the `randint` function comes from. We might use `r = randint(1, 100)` 400 lines later in the file, and we can quickly look at the imports to see where that `randint` function came from. Then, if we need clarification as to how to use the

`randint` class, we can visit the original module. However, if we use the `from random import *` syntax, it takes a lot longer to find where that function was defined. Code maintenance becomes a nightmare.

- If there are conflicting names, we're doomed. Let's say we have two modules, both of which provide a class named `Random`. Using `from random import *` and `from my_module import *` means the second `import` statement can provide definitions that overwrite objects created by the first `import`. If we used `import random` and `import my_module`, we must then use the module names as qualifiers to disambiguate `random.Random` from `my_module.Random`.
- In addition, most code editors are able to provide extra functionality, such as reliable code completion, the ability to jump to the definition of a class, or inline documentation, if specific names are imported. However, the `import *` syntax can hamper their ability to do this reliably.
- Finally, using the `from X import *` syntax can bring unexpected objects into our local namespace. Sure, it will import all the classes and functions defined in the module being imported from, but unless a special `__all__` list is provided in the module, this `import` will also import any classes or modules that were themselves imported by that file! This surprise collection of other dependencies might be surprising because it's far from clear in our application's code.

Every name used in a module should come from a well-specified place, whether it is defined in that module, or explicitly imported from another module. There should be no magic variables that seem to come out of thin air. We should *always* be able to immediately identify where the names in our current namespace originated. We promise that if you use this evil syntax, you will one day have extremely frustrating moments of *where on earth can this class be coming from?*

For fun, try typing `import this` into your interactive interpreter. It prints a nice poem (with a couple of inside jokes) summarizing some of the idioms that Pythonistas tend to practice. Specific to this discussion, note the line "Explicit is better than implicit." Explicitly importing names into your namespace makes your code much easier to navigate than the implicit `from module import *` syntax.

While many, many applications are created with a flat collection of modules, this doesn't always work out. In the next few sections, we'll look at how to organize modules into packages.

Organizing modules

As a project grows into a collection of more and more modules, we may find that we want to add another level of abstraction, some kind of nested hierarchy on our modules. However, we can't put modules inside modules; one file can hold only one file after all, and modules are just files.

Files, however, can go in directories, and so can modules. A **package** is a collection of modules in a directory. The name of the package is the name of the folder. We need to tell Python that a folder is a package to distinguish it from other directories in the project. To do this, place a (normally empty) file in the folder named `__init__.py`. If we forget this file, we won't be able to import modules from that folder.

Let's start with a fresh example, an e-commerce application. This has a number of components, and it makes sense to put the modules into packages to permit expansion of the available features.

We'll put the overall `ecommerce` package in our project folder. This will also contain a `main.py` file to start the program. Let's additionally add another package inside the `ecommerce` package for various payment options.

We need to exercise some caution in creating deeply nested packages. The general advice in the Python community is "flat is better than nested." In this example, we need to create a nested package because there are some common features to all of the various payment alternatives.

The folder hierarchy will look like this, rooted under a directory in the project folder, commonly named `src`:

```
src/
 +-- main.py
 +-- ecommerce/
     +-- __init__.py
     +-- database.py
     +-- products.py
     +-- vendors.py
     +-- payments/
     |   +-- __init__.py
     |   +-- common.py
     |   +-- square.py
     |   +-- stripe.py
     +-- contact/
         +-- __init__.py
         +-- email.py
```

The `src` directory will be part of an overall project directory. In addition to `src`, the project will often have directories with names like `docs` and `tests`. It's common for the project parent directory to also have configuration files for tools such as **tox** and **pytest** among others. We'll return to this in *Chapter 13*.

In Python, there are two ways of importing modules: absolute imports and relative imports. If our packages have deeply-nested sub-packages, absolute imports can become lengthy. One common way to avoid this is to switch to a flatter design. We'll look at each import variant separately.

Absolute imports

Absolute imports specify a complete path to the module we want to import. If we need access to a `Product` class inside the `products` module inside the `ecommerce` package, we could use any of the following syntaxes to perform an absolute import:

```
import ecommerce.products

product_1 = ecommerce.products.Product("fore")
```

Or, we could specifically import a single class definition from the module within a package:

```
from ecommerce.products import Product

product_2 = Product("main")
```

Or, we could import an entire module from the containing package:

```
from ecommerce import products

product_3 = products.Product("mizzen")
```

The `import` statements use the period operator to separate packages or modules. A package is a **namespace** that contains module names, much in the way an object is a namespace containing attribute names.

These statements will work from any module. We could instantiate a `Product` class using this syntax in `main.py`, in the `database` module, or in either of the two payment modules. The other part of this is making sure the packages are available to Python. For example, the packages can also be installed in the Python `site-packages` directory; or the `PYTHONPATH` environment variable

could be set to tell Python which additional folders to search for packages and modules it is going to import.

With these choices, which syntax do we choose? It depends on your audience and the application at hand. If there are dozens of classes and functions inside the `products` module that we want to use, we'd generally import the module name using the `from ecommerce import products` syntax, and then access the individual classes using `products.Product`. If we only need one or two classes from the `products` module, we can import them directly using the `from ecommerce.products import Product` syntax. It's important to write whatever makes the code easiest for others to read and extend.

Relative imports

When working with related modules inside a deeply nested package, it seems kind of redundant to specify the full path. This is where **relative imports** come in. Relative imports identify a module as it is positioned relative to the current module. They only make sense inside modules, and, further, they only make sense where there's a complicated package structure.

For example, if we are working in the `products` module and we want to import the `Database` class from the `database` module next to it, we could use a relative import:

```
from .database import Database
```

The period before `database` says *use the database module inside the current package*. In this case, the current package is the package containing the `products.py` file we are currently editing, that is, the `ecommerce` package. The `products` and `database` modules are side-by-side peers.

If we were editing the `stripe` module inside the `ecommerce.payments` package, we would want, for example, to *use the database package inside the parent package* instead. This is easily done with two periods, as shown here:

```
from ..database import Database
```

We can use more periods to go further up the hierarchy, but at some point, we have to acknowledge that we have too many packages. Too much navigation suggests a flatter design would be more clear.

Of course, we can also go down one side and back up the other. The following would be a valid import from the `ecommerce.contact` package containing an `email` module if we wanted to import the `send_mail` function into our `payments.stripe` module:

```
from ..contact.email import send_mail
```

This import uses two periods indicating *the parent of the payments.stripe package*, and then uses the normal `package.module` syntax to go back down into the `contact` package to name the `email` module. It can be difficult to visualize the package structure required for this to work.

Relative imports aren't as useful as they might seem. As mentioned earlier, the *Zen of Python* (you can read it when you run `import this`) suggests "flat is better than nested". Python's standard library is relatively flat, with few packages and even fewer nested packages. If you're familiar with Java, the packages are deeply nested, something the Python community likes to avoid.

Packages as a whole

We can import code that appears to come directly from a package, as opposed to a module inside a package. As we'll see, there is a module involved,but it has a special name, so it's hidden. In this example, we have an `ecommerce` package containing two module files named `database.py` and `products.py`. The `database` module contains a `db` variable that is accessed from a lot of places. Wouldn't it be convenient if this could be imported as `from ecommerce import db` instead of `from ecommerce.database import db`?

Remember the `__init__.py` file that defines a directory as a package? This file can contain any variable or class declarations we like, and they will be available as part of the package. In our example, if the `ecommerce/__init__.py` file contained the following line:

```
from .database import db
```

We could then access the `db` attribute in a module such as `main.py` or any other file using the following import:

```
from ecommerce import db
```

It might help to think of the `ecommerce/__init__.py` file as if it were some kind `ecommerce.py` file. It lets us view the `ecommerce` package as having a module protocol as well as a package protocol.

This can also be useful if you put all your code in a single module and later decide to break it up into a package of modules. The `__init__.py` file for the new package can still be the main point of contact for other modules using it, but the code can be internally organized into several different modules or subpackages.

We recommend not putting much code in an `__init__.py` file, though. Programmers do not expect actual definitions to be placed in this file. It becomes like a `from x import *`: it can trip readers up if they are looking for the declaration of a particular piece of code and can't find it until they check `__init__.py`.

It's important to avoid complicated dependencies among modules. if module C requires some functions from module F and module F requires some class definitions from module C, there's a problem. Modules can't have a mutual (or circular) relationship. The dependencies must be direct; an `import` statement must be able to read — and execute the definitions — from the module it names. In some cases, it can help to keep the number of modules very small to avoid tangled dependencies.

After looking at modules in general, let's dive into what should be inside a module. The rules are flexible. Python gives you some freedom to bundle things in a way that's meaningful and informative.

Organizing our code in modules

The Python module is an important focus for organizing our code. Every application or library is at least one module. Even a seemingly "simple" Python script is a module. Inside any one module, we can specify all the relevant variables, classes, or functions. Some languages encourage (or require) one class per file; this isn't true in Python.

A module-level global can be a handy way to store state without namespace conflicts. For example, we have been importing the `Database` class into various modules and then instantiating it, but it might make more sense to have only one `database` object globally available from the `database` module. The `database` module might look like this:

```
class Database:
    """The Database Implementation"""
    def __init__(self, connection: str | None = None) -> None:
        """Create a connection to a database."""
        pass
```

```
database = Database("file:/path/to/database")
```

Then we can use any of the import methods we've discussed to access the `database` object. In the `products.py` module, for example, we might use this:

```
import ecommerce.database as database
```

A problem with the preceding definition of the `database` module is that a `database` object is created immediately when the module is first imported, which is usually when the program starts up. This isn't always ideal, since connecting to a database can take a while, slowing down startup. More important than that, the database connection information may not yet be available because we need to read a configuration file or parse command-line parameters or decode environment variables (or all three!) We could delay creating the database until it is actually needed by using an `initialize_database()` function to set the value of a module-level variable:

```
db: Database | None = None


def initialize_database(connection: str | None = None) -> None:
    global db
    if not db:
        db = Database(connection)
```

The `Database | None` type hint signals this may be `None` or it may have an instance of the `Database` class.

The `global` keyword tells Python that the database variable inside `initialize_database()` is the module-level variable; it exists outside the function. If we had not specified the variable as global, Python would have created a new local variable that would be discarded when the function exits, leaving the module-level value unchanged.

We need to make one additional change. We need to import the `database` module as a whole. In order to use this, we can't import the `db` object from inside the module and use it; it might not have been initialized. We need to be sure the `database.initialize_database()` function is called before the `db` will have a meaningful value. It's important that some class have responsibility for this so that the database connection is initialized properly. When we need direct access to the database object, we'd use `database.db` to be sure we're talking about the `db` variable in the `database` module.

A common alternative is a function that returns the current database object. We could import this function everywhere we needed access to the database:

```
def get_database(connection: str | None = None) -> Database:
    global db
    if not db:
        db = Database(connection)
    return db
```

We'd change our imports to fetch this `get_database()` function instead of the module-level `db` object. We can imagine a function as an API for a module. It's similar to the way a method is the API for a class. The difference is that there's only a single instance of a module, where many objects can be created that all have a common class.

As these examples illustrate, all module-level code is executed immediately at the time it is imported. Executing the `class` and `def` statements creates code objects to be used later. This is our general expectation, when importing a module: it will execute `class` and `def` statements.

We can get ourselves in trouble working with script files. Imagine, for a moment, that we once wrote a script or an application that does something useful. It runs via a command such as `python my_script.py`. Now, after using it for a while, we want to import one function or class from that module into a different program. However, as soon as Python executes a statement such as `import my_script` it, any code at the module level of the `my_script` script file is immediately executed. We can end up running the first program when we really only meant to access a couple of functions inside that module.

To solve this, we should always put the code for even the simplest script in a function (conventionally, called `main()`). Then, the module can be set up to only execute the `main()` function when we know we are running the module as a script. We can be sure the code will not run when our module is being imported. We can do this by **guarding** the call to the `main()` function inside a conditional statement, demonstrated as follows:

```
def main() -> None:
    """
    Does the useful work.

    >>> main()
    p1.calculate_distance(p2)=5.0
```

```
    """
    p1 = Point()
    p2 = Point(3, 4)
    print(f"{p1.calculate_distance(p2)=}")


if __name__ == "__main__":
    main()
```

We can import the contents of this module without any surprising processing happening. The `Point` class (and the `main()` function) can be reused without worry. At import time, the value of the `__name__` variable is the module name.

When run as a main program, the value of the `__name__` variable is `"__main__"`. This means the module will behave like a script and execute everything in the body of the `if` statement. In this example, it's the `main()` function.

Make it a policy to write all scripts as functions, with the function evaluated in an `if __name__ == "__main__":` suite of statements. This makes testing much easier. And, it makes reuse much easier.

So, methods go in classes, which go in modules, which go in packages. Is that all there is to it?

Actually, no. This is the typical order of things in a Python program, but it's not the only possible layout. Classes can be defined anywhere. They are typically defined at the module level, but they can also be defined inside a function or method, like this:

```
class Formatter:
    def format(self, string: str) -> str:
        return string


def format_string(string: str, formatter: Formatter | None = None) -> str:
    """
    Format a string using the formatter object, which
    is expected to have a format() method that accepts
    a string.
    """
```

```
    class DefaultFormatter(Formatter):
        """Format a string in title case."""

        def format(self, string: str) -> str:
            return str(string).title()

    if not formatter:
        formatter = DefaultFormatter()

    return formatter.format(string)
```

We've defined a `Formatter` class as an abstraction to explain what a formatter class needs to have. We haven't used the abstract base class (`abc`) definitions (we'll look at these in detail in *Chapter 6*). Instead, we've provided a method with no useful body. It has a full suite of type hints, to make sure type-checking tools have a formal definition of our intent.

Within the `format_string()` function, we created an internal class that is an extension of the `Formatter` class. This formalizes the expectation that our class inside the function has a specific set of methods. This connection between the definition of the `Formatter` class, the `formatter` parameter, and the concrete definition of the `DefaultFormatter` class assures us that we haven't accidentally forgotten something or added something.

We can execute this function like this:

```
>>> hello_string = "hello world, how are you today?"
>>> print(f" input: {hello_string}")
 input: hello world, how are you today?
>>> print(f"output: {format_string(hello_string)}")
output: Hello World, How Are You Today?
```

If no `Formatter` instance is supplied, the function creates a new `Formatter` subclass of its own as a local class and instantiates it. Since the `Formatter` subclass is created inside the scope of the function, this class cannot be accessed from anywhere outside of that function.

We can define a class in a function or a class. Similarly, functions can be defined inside a function, class, or a class method. In general, any Python statement can be executed at any time.

Common practice (enshrined in PEP-8) suggests all of the `import` statements be collected near the beginning of a module. They can be provided anywhere, but why have to search for them?

These inner classes and functions are occasionally useful for one-off items that don't require or deserve their own scope at the module level, or only make sense inside a single method. However, it is not common to see Python code that frequently uses this technique. We want to make our code clear and easy to follow; the general advice remains "flat is better than nested."

We've seen how to create classes and how to create modules. With these core techniques, we can start thinking about writing useful, helpful software to solve problems. When the application or service gets big, though, we often have boundary issues. We need to be sure that objects respect each other's privacy and avoid confusing entanglements that make complex software into a spaghetti bowl of interrelationships. We'd prefer each class to be a nicely encapsulated ravioli.

And, of course, we're using Python to solve problems. It turns out there's a huge standard library available to help us create useful software. The vast standard library is why we describe Python as a "batteries included" language. Right out of the box, you have almost everything you need, no running to the store to buy batteries.

Outside the standard library, there's an even larger universe of third-party packages. In the next section, we'll look at how we extend our Python installation with even more ready-made goodness.

Third-party libraries and virtual environments

Python ships with a lovely standard library, which is a collection of packages and modules that are available on every machine that runs Python. However, you'll soon find that it doesn't contain everything you need. When this happens, you have two options:

- Write a supporting library yourself
- Use somebody else's code, a third-party library

We won't be covering the details about turning your packages into libraries. If you have a problem you need to solve and you don't feel like coding it (the best programmers are extremely lazy and prefer to reuse existing, proven code, rather than write their own), you can probably find the library you want on the **Python Package Index** (**PyPI**) at `https://pypi.python.org/`. Once you've identified a package that you want to install, you can use a tool called `pip` to install it.

You can install packages using an operating system command such as the following:

```
% python -m pip install mypy
```

If you try this without making any preparation, you may get an error that you don't have permission

to update the Python installation you're using. This can happen in the cases where a Python interpreter is part of the operating system, or was installed by someone who as administrative privileges. Generally, we don't want to haphazardly install (and uninstall) packages. We generally need to take a disciplined approach and keep track of what packages we've added.

The common consensus in the Python community is to avoid any Python that's part of the OS. We should not tinker with these installations. Some Linux distributions can include Python. Older releases of macOS included Python. It's best to ignore these, and always install a fresh, new Python that's under our control.

We need to take one more step. We need to set up a **virtual environment** to manage the packages for a given project. (It's really unlikely that you'll only work on one — and only one — programming project.)

Python ships with a tool called `venv`, a utility that creates virtual environments. The environment, defined by the OS, is the collection of files and environment variables, set when you log in or open a terminal window. A virtual environment is an extension to the OS-defined environment. The idea is to activate and deactivate virtual environments to manage the mix of Python libraries.

When you activate a virtual environment, commands related to Python will work with the active virtual environment's Python. Changes to this environment are isolated from other virtual environments. Here's how to use it on most OSes:

```
% cd project_directory
% python -m venv env
% source env/bin/activate
```

This creates an environment with the unimaginative name of `env`. The `source` command updates the OS environment with the settings from the virtual environment, `env`, being activated.

Here's the slight variation for Windows:

```
> cd project_directory
> python -m venv env
> env/Scripts/activate
```

(For other OSes, see `https://docs.python.org/3/library/venv.html`, which has all the variations required to activate the environment.)

Once a virtual environment is activated, you are assured that `python -m pip` will install new

packages into the active virtual environment. There's a small (but very real) possibility that a command such as `pip` could be bound to a virtual environment that's not the active one. (This often happens when creating a new environment and not deactivating the old environment.) For this reason, we strongly encourage using `python -m pip` to focus on the active virtual environment.

On a home computer — where you have access to the privileged files — you can sometimes get away with installing and working with a single, centralized system-wide Python. In this case, there's one environment — the OS environment — which is the one-and-only environment. Experimentation with different libraries can be difficult. Multiple projects with different requirements will become annoying. Tracking changes to dependencies may become nearly impossible.

In an enterprise computing environment, where multiple people share a server, and system-wide directories require special privileges, a virtual environment is required. The alternative to a virtual environment would be one OS environment shared by all users. This means negotiating with the administrators (and all the other users) to upgrade a Python library.

Virtual environments are essential.

It's typical to create a distinct virtual environment for each Python project. You can store your virtual environments anywhere, but a good practice is to keep them in the same directory as the rest of the project files. When working with version control tools such as **Git**, a `.gitignore` file can make sure your virtual environments are not checked into the Git repository. Tools such as **uv** can help create additional files such as the `.gitignore` file.

As shown in the preceding code, when starting something new, we often create the directory, and then `cd` into that directory. Then, we'll run the `python -m venv env` utility to create a virtual environment, usually with a simple name such as `env`, and sometimes with a more complex name such as `CaseStudy5ed`.

Each time we do some work on a project, we can `cd` to the directory and execute the `source env/bin/activate` (or the Windows variant) command in the terminal window to activate the virtual environment. When switching projects, a `deactivate` command unwinds the environment setup.

Some tools look for external dependencies in files such as `requirements.txt`. Other tools, such as **uv** or **poetry**, track dependencies in the `pyproject.toml` file. What's essential is carefully logging

the dependencies so tools can track and update the modules on which your project depends.

Virtual environments are essential for keeping your third-party dependencies tidy and organized. Imagine having an old project using a package such as **Pydantic** version 1.10. A new project starts up, and the other people you're working with have decided to upgrade to Pydantic version 2.9. Using separate virtual environments makes it easy to work on either project in spite of the different dependencies. (At some point, you may rewrite the old project, and want to test the changes in yet another virtual environment.)

Virtual environment management

There are several add-on tools for managing virtual environments more effectively. For example `virtualenv`, can be used instead of the built-in `venv` package.

In some cases, even more support and automation is required. If you're working in a data science environment, you'll probably want to use `conda` so you can install the complex statistical and scientific packages. The **conda** tool works with the Anaconda libraries. For more information, see `https://docs.conda.io/en/latest/`.

Tools such as **uv** and **poetry** can help with installing packages, creating packages, and managing virtual environments. For more information, see `https://docs.astral.sh/uv/` and `https://python-poetry.org`, respectively.

When using a tool like **uv**, use the `uv init` command to initialize the project directory. The `-app` option sets up the common structure for building an application. The `-lib` option will prepare the kind of directory structure that's helpful for building a library. With no options, the assumption is that you're creating a script file.

The command to incorporate a library into a project will then be slightly different:

```
% uv add pydantic
```

This will update the project's dependencies. When needed, the virtual environment can be rebuilt with the following:

```
% uv sync
```

This command forces **uv** to examine all the packages you added to work out the compatible versions

of all of the dependencies, and install exactly what's needed. It keeps the environment tidy, and reproducible by others. The more projects we work on, the more environments we'll manage. Tools can help.

Recall

Some key points in this chapter are as follows:

- Python has optional type hints to help describe how data objects are related and what the parameters should be for methods and functions.
- We create Python classes with the `class` statement. We should initialize the attributes in the special `__init__()` method.
- Modules and packages are used as higher-level groupings of classes.
- We need to plan out the organization of module content. While the general advice is "flat is better than nested," there are a few cases where it can be helpful to have nested packages.
- Python has no notion of "private" data. We often say "we're all adults here"; we can see the source code, and private declarations aren't very helpful. This doesn't change our design; it simply removes the need for a handful of keywords.
- We can install third-party packages using PIP tools. We can create a virtual environment, for example, with `venv`.
- Some OO design techniques are as follows:
  - Encapsulating features into classes
  - Composition to build a class from component objects

Exercises

Write some object-oriented code. The goal is to use the principles and syntax you learned in this chapter to ensure you understand the topics we've covered. If you've been working on a Python project, go back over it and see whether there are some objects you can create and add properties or methods to. If your Python project is large, try dividing it into a few modules or even packages and play with the syntax. While a "simple" script may expand when refactored into classes, there's generally a gain in flexibility and extensibility.

If you don't have such a project, try starting a new one. It doesn't have to be something you intend

to finish; just stub out some basic design parts. You don't need to fully implement everything; often, just `print("this method will do something")` is all you need to get the overall design in place. This is called **top-down design**, in which you work out the different interactions and describe how they should work before actually implementing what they do. The converse, **bottom-up design**, implements details first and then ties them all together. Both patterns are useful at different times, but for understanding object-oriented principles, a top-down workflow is more suitable.

If you're having trouble coming up with ideas, try writing an application to maintain a to-do list. It can keep track of things you want to do each day. Items can have a state change from incomplete to completed. You might want to think about items that have an intermediate state of started, but not yet completed.

Another application is hinted at in *Chapter 1*. There, we have an example of a script that does a lot of processing. We describe some classes that seem to be part of the script processing. Take some time to think through some class definitions that might replace the complicated-looking code to navigate through nested dictionaries. Sketch out the classes that might be relevant.

Now try designing a bigger project. A collection of classes to model playing cards can be an interesting challenge. Cards have a few features, but there are many variations on the rules. A class for a hand of cards has interesting state changes as cards are added. Locate a game you like and create classes to model cards, hands, and play. (Don't tackle creating a winning strategy; that can be hard.)

A game like Cribbage has an interesting state change where two cards from each player's hand are used to create a kind of third hand, called the "crib." Make sure you experiment with the package and module-importing syntax. Add some functions in various modules and try importing them from other modules and packages. Use relative and absolute imports. See the difference, and try to imagine scenarios where you would want to use each one.

Summary

In this chapter, we learned how to create classes and assign properties and methods in Python. Unlike many languages, Python differentiates between a constructor and an initializer. It has a relaxed attitude toward access control. There are many different levels of scope, including packages, modules, classes, and functions. We understood the difference between relative and absolute imports, and how to manage third-party packages that don't come with Python.

In the next chapter, we'll learn more about sharing an implementation among classes using inheri-

tance.

Join our community Discord space

Join our Python Discord workspace to discuss and know more about the book: `https://packt.link/dHrHU`

3

When Objects Are Alike

In the programming world, duplicate code is considered evil. We should not have multiple copies of the same, or similar, code in different places. When we fix a bug (or add a feature) in one copy and fail to make the same change in another copy, we cause no end of problems for ourselves.

There are many ways to merge pieces of code or objects that have a similar functionality. In this chapter, we'll be covering the most famous object-oriented principle: inheritance. As discussed in *Chapter 1*, inheritance allows us to create "is-a" relationships between two or more classes, abstracting common logic into superclasses and extending the superclass with specific details in each subclass. In particular, we'll be covering the Python syntax and principles for the following:

- The inheritance relationship
- Using inheritance to write Python classes
- Using inheritance to extend built-in types
- Multiple inheritance
- Polymorphism and duck typing

We'll start by taking a close look at how inheritance works to factor out common features so we can avoid copy-and-paste programming.

The inheritance relationship

For example, there are 32 chess pieces in our chess set, but there are only six different types of pieces (pawns, rooks, bishops, knights, king, and queen). Each type behaves differently when it is moved. Each of these piece classes have properties, such as the color and the chess set they are part of, but they also have unique shapes when drawn on the chess board, and make different moves.

Figure 3.1 shows how the six types of pieces can inherit from a **Piece** class:

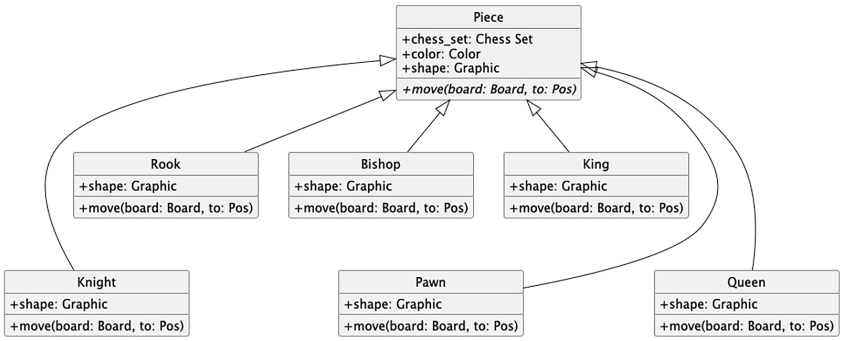


Figure 3.1: How chess pieces inherit from the Piece class

The hollow arrows indicate that the individual classes of pieces inherit from the `Piece` class. All the child classes automatically have a **chess_set** and **color** attribute inherited from the base class. Each piece provides a different shape property (to be drawn on the screen when rendering the board), and a different **move** method to move the piece to a new position on the board at each turn.

To be able to move, all subclasses of the **Piece** class need to have a **move** method; otherwise, when the board tries to move the piece, it will get confused. It is possible that we would want to create a new version of the game of chess that has one additional piece (maybe a wizard). Our current design will allow us to add this piece without giving it a **move** method. Attempting to use this piece without the required method will cause runtime problems.

We can fix this by creating a dummy `move()` method on the **Piece** class. The subclasses can then **override** this method with a more specific implementation. The default implementation might, for example, pop up an error message that says **That piece cannot be moved**.

Overriding methods in subclasses allows very powerful object-oriented systems to be developed. For example, if we wanted to implement a **Player** class with artificial intelligence, we might provide a **calculate_move** method that takes a **Board** object and decides which piece to move where.

A very basic class might randomly choose a piece and direction and move it accordingly. We could then override this method in a subclass with the *Deep Blue implementation*. (See `https://www.chess.com/terms/deep-blue-chess-computer`.) The first class would be suitable for play against a raw beginner; the latter would challenge a grand master. The important thing is that other methods in the class, such as the ones that inform the board as to which move was chosen, need not be changed; this implementation can be shared between the two classes.

In the case of chess pieces, it doesn't really make sense to provide a default implementation of the **move** method. All we need to do is specify that the **move** method is required in any subclasses. This can be done by making **Piece** an **abstract class** with the **move** method declared as **abstract**. Abstract methods basically say this:

> *"We demand this method exist in any non-abstract subclass, but we are declining to specify an implementation in this class."*

Indeed, it is possible to make an abstraction that does not implement any methods at all. Such a class would simply tell us what the class *should* do, but provides absolutely no help on how to do it. In Python, we often use the **abc** module to define these abstract base classes. We mark the abstract methods with an **@abstractmethod** decorator to act as a reminder that an implementation is required.

```
import abc

class Piece(abc.ABC):
    def __init__(self, set: ChessSet, color: Color, shape: Graphic) -> None:
        self.chess_set = set
        self.color = color
        self.shape = shape

    @abc.abstractmethod
    def move(self, board: Board, to: Position) -> None:
        ...
```

We make an abstract base class a subclass of `abc.ABC`. Doing this assures that an exception will be raised if the application attempts to create an instance of a subclass that doesn't provide concrete implementations for all of the abstract methods.

The `@abc.abstractmethod` decorator marks a definition as being only a specification that requires an override. And yes, the body really is `...`; this is valid Python and it will raise a runtime error

if — somehow — this method were executed. Rest assured, it's really hard to execute an abstract method. An application can't create an instance of the `Piece` class with this method missing; that's a really big obstacle. A `TypeError` exception will be raised when trying to create an object from a class that has abstract methods. Creating an instance of a subclass that provides an implementation for the abstract methods works as expected.

Using inheritance

The core idea behind inheritance is providing a way to avoid repeating code in multiple classes.

Technically, every class we create uses inheritance. All Python classes are subclasses of the built-in class named `object`. This class provides a little bit of metadata and a few built-in behaviors so Python can treat all objects consistently.

Inheritance requires a minimal amount of extra syntax over a basic class definition. Include the name of the parent class inside parentheses after the class name (and before the colon that follows.) This is all we have to do to tell Python that the new class should be derived from the given superclass. For more information on the syntax, see *section 9.5* (`https://docs.python.org/3/tutorial/classes.html#inheritance`) of the Python Tutorial.

How do we apply inheritance in practice? One use of inheritance is to add functionality to an existing class. Let's start with a contact manager that tracks the names and email addresses of several people. A `Contact` class can be responsible for maintaining a global list of all contacts ever seen in a class variable, and for initializing the name and address for an individual contact:

```
from __future__ import annotations
from typing import ClassVar


class Contact:
   all_contacts: ClassVar[list[Contact]] = []

   def __init__(self, name: str, email: str) -> None:
       self.name = name
       self.email = email
       Contact.all_contacts.append(self)

   def __repr__(self) -> str:
       return (
```

```
            f"{self.__class__.__name__}("
            f"{self.name!r}, {self.email!r}"
            f")"
        )
```

This example introduces us to two things, **class variables**, and the `__repr__()` method. The `all_contacts` list, because it is part of the class definition, is shared by all instances of this class. This means that there is only one `Contact.all_contacts` list. There are two modes of access for this: reading and writing.

- We can get the value `self.all_contacts` within any method on an instance of the `Contact` class. This works because any attribute that can't be found in the object (via `self`), will be searched for in the class.
- We can only write set value using the name `Contact.all_contacts`. (If you ever attempt to **set** a class variable using `self.all_contacts`, you will actually be creating a **new** instance variable associated just with that object.)

The `__repr__()` method is used by the built-in `repr()` function. This function is often used when printing an object in the interactive Python REPL environment. We often design this function to produce a line of Python that will recreate the object. We've used the special names `__class__` and `__name__` to extract some internal information from each object. This exposes the name for the object's class.

We can see how the class tracks data with the following example:

```
>>> c_1 = Contact("Dusty", "dusty@example.com")
>>> c_2 = Contact("Steve", "steve@itmaybeahack.com")
>>> Contact.all_contacts
[Contact('Dusty', 'dusty@example.com'), Contact('Steve',
'steve@itmaybeahack.com')]
```

We created two instances of the `Contact` class and assigned them to variables `c_1` and `c_2`. When we looked at the `Contact.all_contacts` class variable, we saw that the list has been updated to track the two objects.

This class allows us to track a couple of pieces of data about each contact. But what if some of our contacts are also suppliers that we need to order supplies from? We could add an `order` method to the `Contact` class, but that would allow people to accidentally order things from contacts who are

customers or family friends. Instead, let's create a new `Supplier` class that acts like our `Contact` class, but has an additional `order` method that accepts a yet-to-be-defined `Order` object:

```
class Supplier(Contact):
    def order(self, order: "Order") -> None:
        print(
            "If this were a real system we would send "
            f"'{order}' order to '{self.name}'"
        )
```

Now, if we test this class in our trusty interpreter, we see that all contacts, including suppliers, accept a name and email address in their `__init__()` method. But we can also see that only `Supplier` instances have an `order()` method:

```
>>> c = Contact("Some Body", "somebody@example.net")
>>> s = Supplier("Sue Plier", "supplier@example.net")
>>> print(c.name, c.email, s.name, s.email)
Some Body somebody@example.net Sue Plier supplier@example.net
>>> from pprint import pprint
>>> pprint(c.all_contacts)
[Contact('Dusty', 'dusty@example.com'),
 Contact('Steve', 'steve@itmaybeahack.com'),
 Contact('Some Body', 'somebody@example.net'),
 Supplier('Sue Plier', 'supplier@example.net')]
>>> c.order("I need pliers")
Traceback (most recent call last):
...
AttributeError: 'Contact' object has no attribute 'order'
>>> s.order("I need pliers")
If this were a real system we would send 'I need pliers' order to 'Sue
Plier'
```

Our `Supplier` class can do everything the `Contact` class can do (including adding itself to the list of `Contact.all_contacts`) and also does special things it needs to handle as a supplier. This code-sharing is the beauty of inheritance.

Because we used the object's `__class__` and `__name__` special attribute names, the `Contact` class and `Supplier` and subclass will both report the correct class name.

Also, note that `Contact.all_contacts` has collected every instance of the `Contact` class as well as the subclass, `Supplier`.

Extending built-ins

One interesting use of this kind of inheritance is adding functionality to built-in classes. In the `Contact` class seen earlier, we are adding contacts to a list of all contacts. What if we also wanted to search that list by name? Well, we could add a method on the `Contact` class to search it, but it feels like this method actually belongs to the list itself.

The following example shows how we can do this using inheritance from a built-in type. In this case, we're extending the built-in `list` type.

```
from __future__ import annotations

class ContactList(list["Contact"]):
    def search(self, name: str) -> list[Contact]:
        """All Contacts with name that contains the name parameter's
        value."""
        matching_contacts: list[Contact] = []
        for contact in self:
            if name in contact.name:
                matching_contacts.append(contact)
        return matching_contacts


class Contact:
    all_contacts = ContactList()

    def __init__(self, name: str, email: str) -> None:
        self.name = name
        self.email = email
        Contact.all_contacts.append(self)

    def __repr__(self) -> str:
        return (
            f"{self.__class__.__name__}("
            f"{self.name!r}, {self.email!r}"
            f")"
        )
```

Note that — in spite of using the `__future__` module — the type we're creating is written as `list["Contact"]` with quotation marks around the yet-to-be-defined `Contact` type. This kind of forward reference doesn't work in Python 3.13, so we're forced to refer to not-yet-defined types with string names. Or, we could change the order of the definitions.

Instead of instantiating the generic `list` class to create our `all_contacts` class variable, we create a new `ContactList` instance; this extends the built-in `list` data type, making a stronger claim about what types of objects will populate this list. We can test the new search functionality as follows:

```
>>> c1 = Contact("John A", "johna@example.net")
>>> c2 = Contact("John B", "johnb@sloop.net")
>>> c3 = Contact("Jenna C", "cutty@sark.io")
>>> Contact.all_contacts.search('John')
[Contact('John A', 'johna@example.net'), Contact('John B',
'johnb@sloop.net')]
```

Recall that Python has two ways to create generic `list` objects. Creating a list with `[]` is actually a shortcut for creating a list using `list()`; the two syntaxes behave identically:

```
>>> [] == list()
True
```

The `[]` is short and sweet. We can call it **syntactic sugar**. This is a call to the `list()` constructor, written with two characters instead of six.

Tools such as **mypy** can check the body of the `ContactList.search()` method to be sure it really will create a `list` instance populated with `Contact` objects. The extra detail in the type annotations can be a big help for preventing problems. The type-checking tool can alert us to code that may not do what we intended.

Because we provided the `Contact` class definition after the definition of the `ContactList` class, we had to provide the reference to a not-yet-defined class as a string, `list["Contact"]`. It's more common to provide the individual item class definition first, and the collection can then refer to the defined class by name without using a string. For this example, we provided them out of order.

The construction of a list of items can be simplified from this design pattern — using a `for` statement that appends to a list. This can be simplified to a "list comprehension." This is the subject of *Chapter 10*.

As a second example, we can extend the `dict` class, which is a collection of keys and their associated values. We can create instances of dictionaries using the syntax sugar. Here's an extended dictionary that tracks the longest key it has seen:

```
class LongNameDict(dict[str, int]):
    def longest_key(self) -> str | None:
        """In effect, max(self, key=len), but less obscure"""
        longest = None
        for key in self:
            if longest is None or len(key) > len(longest):
                longest = key
        return longest
```

The hint for the class narrowed the generic `dict` to a more specific `dict[str, int]`. This means the keys are of type `str` and the values are of type `int`. This helps tools (and people) reason about the `LongNameDict` class. (For more on generic types, see *Chapter 7*.) Since the keys are supposed to be `str`-type objects, the statement `for key in self:` will iterate over `str` objects. The result will be `str`, or possibly `None`. That's why the result is described as `str | None`.

(This raises questions, also. Is `None` appropriate? Perhaps not. Perhaps a `ValueError` exception is a better idea than returning some value that stands in for an error. Further design will have to wait until *Chapter 4*.)

This class is going to be working with strings and integer values. Perhaps the strings are usernames, and the integer values are the number of articles they've read on a website. In addition to the core username and reading history, we also need to know the longest name so we can format a table of scores with the right size display box. This is easy to test in the interactive interpreter:

```
>>> articles_read = LongNameDict()
>>> articles_read['lucy'] = 42
>>> articles_read['c_c_phillips'] = 6
>>> articles_read['steve'] = 7
>>> articles_read.longest_key()
'c_c_phillips'
```

What if we wanted a more generic dictionary? Say, with either strings or integers as the values? We'd need a slightly more expansive type hint. We might use `dict[str, str | int]` to describe a dictionary mapping strings to a union of either strings or integers. For more on these kinds of type unions, we'll wait for *Chapter 7*.

In the next section, we'll look more deeply at the benefits of inheritance and how we can selectively leverage features of the superclass in our subclass.

Overriding and super()

Inheritance is great for *adding* new behavior to existing classes, but what about *changing* behavior? Our `Contact` class allows only a name and an email address. This may be sufficient for most contacts, but what if we want to add a phone number for our close friends?

As we saw in *Chapter 2*, we can do this by setting a `phone` attribute on the contact after it is constructed. But if we want to make this additional attribute part of initialization, we have to override the `__init__()` method. Overriding means altering or replacing a method of the superclass with a new method (with the same name) in the subclass. No special syntax is needed to do this; the subclass's newly created method is automatically called instead of the superclass's method, as shown in the following code:

```
class Friend(Contact):
    def __init__(self, name: str, email: str, phone: str) -> None:
        self.name = name
        self.email = email
        self.phone = phone
```

Any method can be overridden, not just `__init__()`. Before we go on, however, we need to address some problems in this example. Specifically, the `Contact` and `Friend` classes have duplicate code to set up the `name` and `email` properties. This can make code maintenance complicated, as we have to update the code in two places. More alarmingly, our `Friend` class is neglecting to add itself to the `all_contacts` list we have created on the `Contact` class. Finally, looking forward, if we add a feature to the `Contact` class, we'd like it to also be part of the `Friend` class, and that can't happen with the code as shown.

What we need is a way to execute the original `__init__()` method of the `Contact` class from inside our new class. This is what the built-in `super()` function does: it returns the object as if it was actually an instance of the parent class, allowing us to call the parent method directly. It looks like this:

```
class Friend_S(Contact):
    def __init__(self, name: str, email: str, phone: str) -> None:
        super().__init__(name, email)
        self.phone = phone
```

(We've given this variant an atypical name to make it clearly distinct from the previous example. The name doesn't fit Python naming conventions. We use it so we can put the examples in a single file for side-by-side comparison purposes.)

This example first binds the instance to the parent class using `super()` and calls `__init__()` on that object, passing in the expected arguments. The `Friend_S` class then does its own initialization, namely, setting the `phone` attribute, processing that is unique to the `Friend_S` class.

The `Contact` class provided a definition for the `__repr__()` method to produce a string representation. Our subclass did not override the `__repr__()` method inherited from the superclass. Here's the consequence of that:

```
>>> f = Friend("Dusty", "Dusty@private.com", "555-1212")
>>> f
Friend('Dusty', 'Dusty@private.com')
```

The details shown for a `Friend_S` instance don't include the new `phone` attribute. It's easy to overlook the special method definitions, such as `__repr__()`, when thinking about class design.

This requires adding the method, like the following example:

```
    def __repr__(self) -> str:
        return (
            f"{self.__class__.__name__}("
            f"{self.name!r}, {self.email!r}, {self.phone!r}"
            f")"
        )
```

A `super()` call can be made inside any method. Therefore, all methods can be extended via overriding and using calls to `super()`. The call to `super()` can also be made at any point in the method; we don't have to make the call as the first line. For example, we may need to manipulate or validate incoming parameters before forwarding them to the superclass.

Composition as an alternative to inheritance

When looking at the chess piece design in section *The inheritance relationship*, we used inheritance to isolate common features of distinct varieties of pieces. This isn't the only tool we have for isolating common code. We could — just as easily — define a number of classes that have the different kinds of `move()` methods. This would include a class for pawns, rooks, knights, bishops, kings, and queens,

since each has a distinct move implementation.

Each piece would be a composition of `Piece` methods that show the position and the display icon for the piece — and other things such as relative value when considering a capture — and an instance of one of the `Move` classes.

At this point, one common question is "Which is better?". This is a difficult question, because a word like *better* isn't bound to any specific measurable feature of the software. The choice between composition or inheritance is always a difficult one to make.

Generally, we like to characterize an inheritance relationship with the name "is-a." We try to characterize the various kinds of composition and aggregation as "has-a." In the case of chess pieces, it's clear that "A pawn is a chess piece" is clear and helpful for software designers.

In some cases, it helps to enumerate all of the "is-a" statements from a UML diagram to see if they're really sensible. We might find some places where an "is-a" statement seems wrong, or pushes against the real-world things we're modeling.

Imagine we have an e-commerce application where an email address is modeled as a subclass of URL's. Perhaps these classes have some common methods; someone defined a common abstract class that encompassed both URL strings and email address strings. It helps think of how to defend a statement such as "An email address is a URL" to a skeptical colleague. (Or, maybe, think of yourself in the future when you haven't looked at this software in a year.) It seems clear that an email isn't a URL, but has some features common with a URL. In this case, using a composition of features might be smarter than inheritance.

Often, choosing between "is-a" and "has-a" is obvious. One of the variants makes perfect sense.

The two design techniques are peers: they accomplish the same goals with about the same intellectual costs and runtime overheads. We can consider the **SOLID** design principles when looking for some guidance. Much of the time, we have to pause and reflect on what the design seems to mean about the world.

Multiple inheritance

Multiple inheritance is a touchy subject. In principle, it's simple: a subclass that inherits from more than one parent class can access functionality from both of them. In practice, it requires some care to be sure any method overrides are fully understood.

As a humorous rule of thumb, if you think you need multiple inheritance, you're probably wrong,

but if you know you need it, you might be right. This is often taken to be a dire warning: don't attempt multiple inheritance. We don't think this dire warning is appropriate if a few patterns are followed.

The simplest and most useful form of multiple inheritance follows a design pattern called the **mixin**. A mixin class definition is not intended to exist on its own, but is meant to be inherited by some other class to provide extra functionality. For example, let's say we wanted to add functionality to our `Contact` class that allows sending an email to `self.email`.

Sending email is a common task that we might want to use on many other classes. So, we can write a mixin class to add the email feature:

```
from typing import Protocol

class Emailable(Protocol):
    email: str


class MailSender(Emailable):
    def send_mail(self, message: str) -> None:
        print(f"Sending mail to {self.email=}")
        # Add e-mail logic here
```

For brevity, we didn't include the actual email logic here; if you're interested in studying how it's done, see the `smtplib` module in the Python standard library.

It's a common practice to use an adjective, such as *Emailable*, for the mixin class name. This helps readers to understand a mixin as a way of encapsulating a specific action or focused group of actions. The idea is to use a mixin to add an ability to a base class.

The `MailSender` class can barely function as a standalone class. This mixin doesn't have any `__init__()` special method to set the instance variables. The `email` attribute here is a type hint. This isn't an instance variable; it's a suggestion that the base class will provide this as an instance variable.

This attribute defines a protocol that the mix of classes must satisfy. In this case, the protocol is only a single attribute and type. It defines an aspect that **must** be present in a concrete class; tools such as **mypy** will object if the aspect is not present. The name `self.email` can be resolved as either an instance variable, or a class-level variable, or a property. (For more details on what properties are, see *Chapter 5*.) If the parent class doesn't properly provide this attribute, the mixin subclass

methods can't possibly work.

Mixin-style design can require some care to make sure the various mixin classes have clearly segregated interface protocols. The Interface Segregation Principle is helpful when considering this kind of design. Think of a class with actions that have multiple implementation choices. Maybe one mixin choice uses less memory while another mixin alternative runs faster. One can then choose appropriate mixins based on an application's unique optimization goals.

We can use the `MailSender` mixin with any class that has an `email` attribute defined. Doing this lets us define a new class that describes both `Contact` and `MailSender`, using multiple inheritance:

```
class EmailableContact(Contact, MailSender):
    pass
```

The syntax for multiple inheritance looks like a parameter list in the class definition. Instead of including one base class inside the parentheses, we include two (or more), separated by a comma. It's possible to build classes that have no unique features of their own. A class can be defined as a combination of mixins. In this case, the body of the class definition is often nothing more than the `pass` placeholder statement.

We can test this new hybrid to see the mixin at work:

```
>>> e = EmailableContact("John B", "johnb@sloop.net")
>>> Contact.all_contacts
[EmailableContact('John B', 'johnb@sloop.net')]
>>> e.send_mail("Hello, test e-mail here")
Sending mail to self.email='johnb@sloop.net'
```

The `Contact` initializer is still adding the new contact to the `all_contacts` list, and the mixin is able to send mail to `self.email`, so we know that everything is working.

This wasn't so hard, and you're probably wondering why the "don't attempt multiple inheritance" dire warnings are repeated so often. We'll get into the complexities in a minute, but let's consider some other options we had for this example, rather than using a mixin:

- We could have used single inheritance and added the `send_mail` function to a subclass of `Contact`. The disadvantage here is that the email functionality then has to be duplicated for any unrelated classes that need an email. For example, if we had email information in the payments part of our application, unrelated to these contacts, and we wanted a `send_mail()`

method, we'd have to duplicate the code.

- We can create a standalone Python function for sending an email, and call that function with the correct email address supplied as a parameter when the email needs to be sent. This is a very popular choice in Python. Because the function is not part of a class, it's harder to be sure that proper encapsulation is being used, particularly when there are other state changes such as counting emails.
- We could use composition instead of inheritance. For example, `EmailableContact` could have a `MailSender` object as a property instead of inheriting from it. This leads to a more complex `MailSender` class because it now has to stand alone. It also leads to a more complex `EmailableContact` class because it has to associate a `MailSender` instance with each `Contact`.
- We could try to "monkey patch" the `Contact` class to have a `send_mail` method after the class has been created. (We'll briefly cover monkey patching in *Chapter 13*.) This is done by defining a function that accepts the `self` argument, and setting this function as an attribute on an existing class. While acceptable for creating a unit test fixture, this is terrible for the application itself. Future you will never figure out how the method got added to the class.

Multiple inheritance works best when we're mixing methods from different classes, but it can be messy when we have to call methods on the superclass. When there are multiple superclasses, how do we know which one's methods to call? What is the rule for selecting the appropriate superclass method?

Let's explore these questions by adding a home address to our `Friend` class. There are a few approaches we might take:

- An address is a collection of strings representing the street, city, country, and other related details of the contact. We could pass each of these strings as a parameter into the `Friend` class's `__init__()` method. We could also store these strings in a generic tuple or dictionary. These options work well when the address information doesn't need new methods. Adding methods suggests we should encapsulate this as a separate class.
- Another option would be to create our own `Address` class to hold those strings together, and then pass an instance of this class into the `__init__()` method in our `Friend` class. The advantage of this solution is that we can add behavior (say, a method to give directions or to print a map) to the data instead of just storing it statically. This is an example of composition, as we discussed in *Chapter 1*. The "has-a" relationship of composition is a perfectly viable solution to this problem and allows us to reuse `Address` classes in other entities, such as

buildings, businesses, or organizations.

- A third course of action is a collaborative multiple inheritance design. We'll look at how to use mixin classes to create this more sophisticated definition. Follow along as we step through an implementation that doesn't work out before landing on a design that seems to do everything we need.

The objective here is to add a mixin class to hold an address. We'll call this new class `AddressHolder` instead of `Address` because inheritance defines an "is-a" relationship. It is not correct to say a `Friend` is an `Address`. It seems much better to say `Friend` has an `Address`; we can also argue that a `Friend` class is an `AddressHolder` class. Later, we could create other entities (companies, buildings) that also hold addresses. (Convoluted naming and nuanced questions about "is-a" are indications we should be sticking with composition, rather than inheritance.)

Why "AddressHolder"? Why not "Addressable"? Frequently, mixins have names that are adjectives and reflect an action that's defined by the mixin. The English verb *to address* seems like it applies here, but it also seems a little misleading: it seems more like this mixin holds some additional attributes. Calling it a *holder* seems a little nicer than calling it a *holdable*. This reinforces the common observation that one of the two hardest problems in computing is naming things. (The others are cache invalidation and off-by-one errors.)

Here's a naïve `AddressHolder` class. We're calling it naïve because it doesn't account for multiple inheritance well:

```
class AddressHolder:
    def __init__(
            self,
            street: str,
            city: str,
            state: str,
            code: str
    ) -> None:
        self.street = street
        self.city = city
        self.state = state
        self.code = code
```

We take all the data and toss the argument values into instance variables upon initialization. We'll look at the consequences of this, and then show a better design.

The diamond problem

We can use multiple inheritance to add this new class as a parent of our existing `Friend` class. The tricky part is we now have two parent `__init__()` methods, both of which need to be called. And they need to be called with different arguments. How do we do this?

Well, we could start with a naïve approach for the `Friend` class, also:

```
class Friend_A(Contact, AddressHolder):
    def __init__(
        self,
        name: str,
        email: str,
        phone: str,
        street: str,
        city: str,
        state: str,
        code: str,
    ) -> None:
        Contact.__init__(self, name, email)
        AddressHolder.__init__(self, street, city, state, code)
        self.phone = phone
```

(We've given it an odd-looking name of `Friend_A` because — spoiler alert — it won't work out very well.)

In this example, we directly call the `__init__()` function on each of the superclasses and explicitly pass the `self` argument. This example technically works; we can access the different variables directly in the class. But there are a few problems.

First, consider someone adding yet another mixin. That new superclass could remain uninitialized if we neglect to explicitly call a new initializer. We would get a lot of `AttributeError` exceptions stemming from a class where there's clearly an `__init__()` method. It's rarely obvious that the `__init__()` method wasn't actually used.

A more insidious possibility is a superclass being called multiple times because of the organization of the class hierarchy. We call this the *Diamond Problem*.

Figure 3.2 shows this complicated inheritance problem:

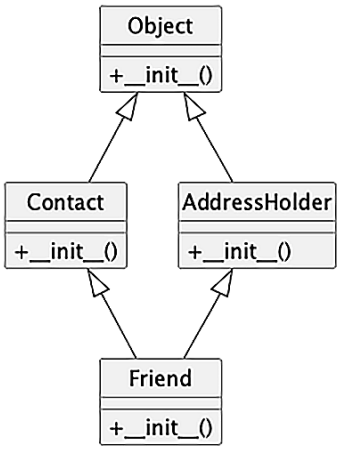


Figure 3.2: Inheritance diagram for the Friend class

The `__init__()` method from the `Friend` class first calls `__init__()` on the `Contact` class, which implicitly initializes the `object` superclass (remember, all classes derive from `object`). Then, after doing this, the `Friend` class calls `__init__()` on `AddressHolder`, which implicitly initializes the `object` superclass *again*. This means the parent class has been set up twice. With the `object` class, that's relatively harmless, but in some situations, it could spell disaster. Imagine trying to connect to a database twice for every request!

The base class should only be called once. Once, yes, but when? Do we call `Friend`, then `Contact`, then `Object`, and then `AddressHolder`? Or `Friend`, then `Contact`, then `AddressHolder`, and then `Object`?

Let's switch from methods called implicitly, to one we need to call explicitly. This will let us see the problem more clearly. Here, we have a base class, `BaseClass`, that has a method named `call_me()`. Two subclasses, `LeftSubclass` and `RightSubclass`, extend the `BaseClass` class, and each overrides the `call_me()` method with different implementations.

Then, *another* subclass extends both of these using multiple inheritance with a fourth, distinct implementation of the `call_me()` method.

Figure 3.3 shows why this is called **diamond inheritance**, note the diamond shape of the class diagram:

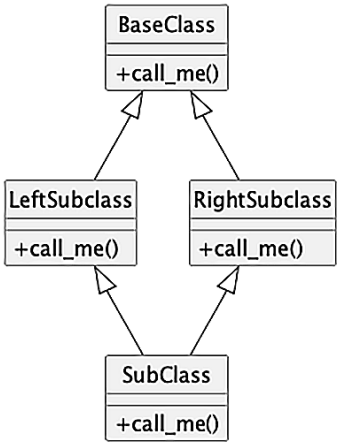


Figure 3.3: Diamond inheritance

Let's convert this diagram into code. This example shows when the methods are called:

```
class BaseClass:
    num_base_calls = 0

    def call_me(self) -> None:
        print("Calling method on BaseClass")
        self.num_base_calls += 1


class LeftSubclass(BaseClass):
    num_left_calls = 0

    def call_me(self) -> None:
        BaseClass.call_me(self)
        print("Calling method on LeftSubclass")
        self.num_left_calls += 1


class RightSubclass(BaseClass):
    num_right_calls = 0

    def call_me(self) -> None:
        BaseClass.call_me(self)
        print("Calling method on RightSubclass")
        self.num_right_calls += 1
```

```
class Subclass(LeftSubclass, RightSubclass):
    num_sub_calls = 0

    def call_me(self) -> None:
        LeftSubclass.call_me(self)
        RightSubclass.call_me(self)
        print("Calling method on Subclass")
        self.num_sub_calls += 1
```

This example ensures that each overridden `call_me()` method directly calls the parent method with the same name. It lets us know each time a method is called. It also creates a distinct instance variable to show how many times it has been called.

This relies on the way Python search starts with the instance, then the class for an object. The very first time it looks for `self.num_sub_calls`, the attribute is not part of the instance. It is, however, part of the class, and this value is fetched. The assignment statement will then create a new instance variable. Ever after, the instance variable will be found first.

If we instantiate one `Subclass` object and call the `call_me()` method on it once, we get the following output:

```
>>> s = Subclass()
>>> s.call_me()
Calling method on BaseClass
Calling method on LeftSubclass
Calling method on BaseClass
Calling method on RightSubclass
Calling method on Subclass
>>> print(
... s.num_sub_calls,
... s.num_left_calls,
... s.num_right_calls,
... s.num_base_calls)
1 1 1 2
```

We can see the base class's `call_me()` method being called twice. This could lead to some pernicious bugs if that method is doing actual work, such as depositing into a bank account, twice.

Python's **Method Resolution Order** (**MRO**) algorithm transforms the diamond into a flat, linear tuple. We can see the results of this in the `__mro__` attribute of a class. The linear version of this diamond is the sequence `Subclass`, `LeftSubclass`, `RightSubClass`, `BaseClass`, `object`. What's important here is that `Subclass` lists `LeftSubclass` before `RightSubClass`; this imposes a strict ordering on the classes in the multiple-inheritance diamond.

The thing to keep in mind with multiple inheritance is that we often want to call the next method in the MRO sequence, not necessarily a method of the parent class. The `super()` function locates the name in the MRO sequence.

Here is the same code written using `super()`. We've renamed some of the classes, adding an `_S` to make it clear this is the version using `super()`:

```
class LeftSubclass_S(BaseClass):
    num_left_calls = 0

    def call_me(self) -> None:
        super().call_me()
        print("Calling method on LeftSubclass_S")
        self.num_left_calls += 1


class RightSubclass_S(BaseClass):
    num_right_calls = 0

    def call_me(self) -> None:
        super().call_me()
        print("Calling method on RightSubclass_S")
        self.num_right_calls += 1


class Subclass_S(LeftSubclass_S, RightSubclass_S):
    num_sub_calls = 0

    def call_me(self) -> None:
        super().call_me()
        print("Calling method on Subclass_S")
        self.num_sub_calls += 1
```

The change is pretty minor; we only replaced the naïve direct calls with calls to `super()`. The definition for `BaseClass` didn't change at all. The `Subclass_S` class, at the bottom of the diamond,

only calls `super()` once rather than having to make the calls for both the left and right. The change is easy enough, but look at the difference when we execute it:

```
>>> ss = Subclass_S()
>>> ss.call_me()
Calling method on BaseClass
Calling method on RightSubclass_S
Calling method on LeftSubclass_S
Calling method on Subclass_S
>>> print(
... ss.num_sub_calls,
... ss.num_left_calls,
... ss.num_right_calls,
... ss.num_base_calls)
1 1 1 1
```

This output looks good: our base method is only being called once. We can see how this works by looking at the `__mro__` attribute of the class:

```
>>> from pprint import pprint
>>> pprint(Subclass_S.__mro__)
(<class 'diamond.Subclass_S'>,
 <class 'diamond.LeftSubclass_S'>,
 <class 'diamond.RightSubclass_S'>,
 <class 'diamond.BaseClass'>,
 <class 'object'>)
```

The order of the classes shows what order `super()` will use. The last class in the tuple is generally the built-in `object` class. As noted earlier in this chapter, it's the implicit superclass of all classes.

This shows what `super()` is actually doing. Since the `print` statements are executed after the `super()` calls, the printed output is in the order each method is actually executed. Let's look at the output from back to front to see who is calling what:

1. We start with the `Subclass_S.call_me()` method. This evaluates `super().call_me()`. The MRO shows `LeftSubclass_S` as next.
2. This begins with evaluation of the `LeftSubclass_S.call_me()` method. This evaluates `super().call_me()`. The MRO puts `RightSubclass_S` as next. This is not a superclass; it's adjacent in the class diamond.

3. Then, evaluation proceeds to the `RightSubclass_S.call_me()` method, `super().call_me()`. This leads to `BaseClass`.
4. Then, the `BaseClass.call_me()` method finishes its processing: printing a message and setting an instance variable, `self.num_base_calls`, to `BaseClass.num_base_calls + 1`.
5. Then, the `RightSubclass_S.call_me()` method can finish, printing a message and setting an instance variable, `self.num_right_calls`.
6. Then, the `LeftSubclass_S.call_me()` method will finish by printing a message and setting an instance variable, `self.num_left_calls`.
7. This serves to set the stage for `Subclass_S` to finish its `call_me()` method processing. It writes a message, sets an instance variable, and rests, happy and successful.

The `super()` call is *not* calling the method on the superclass of `LeftSubclass_S` (which is `BaseClass`). Rather, it is calling `RightSubclass_S`, even though it is not a direct parent of `LeftSubclass_S`. This is the *next* class in the MRO, not the parent class. `RightSubclass_S` then calls `BaseClass` and the `super()` calls have ensured each method in the class hierarchy is executed once.

This feature of `super()` solves the diamond problem elegantly, allowing multiple inheritance to have well-defined behaviors.

Different sets of arguments

Things can get complicated as we return to our `Friend` example. In the `__init__()` method for the `Friend` class, we were originally delegating initialization to the `__init__()` methods of both parent classes, *with different sets of arguments*:

```
leContact('John B', 'johnb@sloop.net')]
nd_mail("Hello, test e-mail here")
```

How can we manage different sets of arguments when using `super()`? We only really have access to the next class in the MRO sequence. Because of this, we need a way to pass the **extra** arguments through the constructors so that subsequent calls to `super()`, from other mixin classes, receive the right arguments.

It works like this. The first call to `super()` provides arguments to the first class of the MRO, passing

the `name` and `email` arguments to `Contact.__init__()`. Then, when `Contact.__init__()` calls `super()`, it needs to be able to pass the address-related arguments to the method of the next class in the MRO, which is `AddressHolder.__init__()`.

This problem often manifests itself any time we want to call a superclass method with the same name, but with different sets of arguments. Collisions often arise around the special method names. Of these, the most common example is having a different set of arguments to various `__init__()` methods, as we're doing here.

There's no magical Python feature to handle cooperation among classes with methods that have divergent parameters. Indeed, this is often flagged a violation of the Liskov Substitution Principle: the subclass methods don't properly match the superclass methods. Consequently, this requires some care to design our class parameter lists. One multiple inheritance approach is to accept keyword arguments for any parameters that are not required by every subclass implementation. A method must pass the unexpected arguments on to its `super()` call, in case they are necessary to later methods in the MRO sequence of classes.

While this works and works well, it's difficult to describe with type hints. Instead, we have to silence **mypy** in a few key places.

Python's function parameter syntax provides a tool we can use to do this, but it makes the overall code look cumbersome. Have a look at a version of the `Friend` multiple inheritance code:

```
class Contact:
    all_contacts = ContactList()

    def __init__(self, /, name: str = "", email: str = "", **kwargs: Any) ->
    None:
        super().__init__(**kwargs)  # type: ignore [call-arg]
        self.name = name
        self.email = email
        self.all_contacts.append(self)

    def __repr__(self) -> str:
        return f"Contact(" f"{self.name!r}, {self.email!r}" f")"
```

```
class AddressHolder:
    def __init__(
        self,
```

```
        /,
        street: str = "",
        city: str = "",
        state: str = "",
        code: str = "",
        **kwargs: Any,
    ) -> None:
        super().__init__(**kwargs)  # type: ignore [call-arg]
        self.street = street
        self.city = city
        self.state = state
        self.code = code
```

```
class Friend(Contact, AddressHolder):
    def __init__(self, /, phone: str = "", **kwargs: Any) -> None:
        super().__init__(**kwargs)
        self.phone = phone
```

We've added the `**kwargs` parameter, which collects all additional keyword argument values into a dictionary. When called with `Contact(name="this", email="that", street="something")`, the `street` argument is put into the `kwargs` dictionary; these extra parameters are passed up to the next class with the `super()` call. The special parameter `/` separates parameters that could be provided by position in the call from parameters that require a keyword to associate them with an argument value. We've given all string parameters an empty string as a default value, also.

If you aren't familiar with the `**kwargs` syntax, it will be set to a dictionary built from all keyword arguments passed into the method not explicitly listed in the parameter list. When we call a method, for example, `super().__init__()`,with `**kwargs` as an argument value, it unpacks the dictionary and passes the results to the method as keyword arguments. We'll look at this in more depth in *Chapter 9*.

We've introduced two comments that are addressed to tools like **mypy** (and any person scrutinizing the code). The `# type: ignore` comments provide a specific error code, `call-arg`, on a specific line to be ignored. In this case, we need to ignore the `super().__init__(**kwargs)` calls because it isn't obvious to **mypy** what the MRO will be at runtime. As someone reading the code, we can look at the `Friend` class and see the order: `Contact` and `AddressHolder`. This order means that inside the `Contact` class, the `super()` function will locate the next class, `AddressHolder`.

The **mypy** tool, however, doesn't look this deeply; it goes by the explicit list of parent classes in the `class` statement. Since there's no parent class named, **mypy** is only able to infer the `object` class will be located by `super()`. Since `object.__init__()` does not take any arguments, the `super().__init__(**kwargs)` in both `Contact` and `AddressHolder` appear incorrect. Practically, the chain of classes in the MRO will consume all the various parameters and there will be nothing left over for the `AddressHolder` class's `__init__()` method.

If we want to be **sure** that all the names from `kwargs` are consumed, we need to add an error check. The next-to-last class in the `__mro__` chain should not receive any additional values in `kwargs`. (The last class is always `object`.) If the `AddressHolder` class gets any `kwargs`, it's something that was not consumed by any of the previous classes; raising a `TypeError` exception for this situation is appropriate. The `object` class will do this.

The previous example does what it is supposed to do. But it's supremely difficult to answer the question: *What arguments do we need to provide* to the `Friend.__init__()` method? This is the foremost question for anyone planning to use the class, so a docstring should be added to the method to explain the entire list of parameters from all the parent classes.

The error message in the event of a misspelled or extraneous parameter can be confusing, also. The message `TypeError: object.__init__() takes exactly one argument (the instance to initialize)` isn't too informative on how an extra parameter came to be provided to `object.__init__()`.

We have covered many of the caveats involved with cooperative multiple inheritance in Python. Multiple inheritance following the **mixin** pattern often works out very nicely. The idea is to have additional methods defined in mixin classes, avoid overlapping method names, and keep all the attributes centralized in a single class hierarchy. This can avoid the complexity of cooperative initialization.

Design using composition also often works better than complex multiple inheritance. Many of the design patterns we'll be covering in *Chapter 11* and *Chapter 12*, are examples of composition-based design.

The inheritance paradigm depends on a clear "is-a" relationship between classes. Multiple inheritance folds in other relationships that aren't as clear. We can say that an "Email is a kind of Contact," for example. This seems perfectly sensible. But it doesn't seem as clear that we can say "A Customer is an Email." It seems better to say "A Customer has an Email address" or "A Customer is contacted via Email," using a composition-like "has-an" or "is contacted by" relationship instead of a direct

"is-a" relationship.

Polymorphism

Polymorphism is a showy name describing a simple concept: different behaviors happen depending on which subclass is being used, without having to explicitly know what the subclass actually is. It is characterized by the Liskov Substitution Principle. The design principle reminds us we should be able to substitute any subclass for its superclass.

As an example, imagine a program that plays audio files. A media player might need to load an `AudioFile` object and then play it. We can put a `play()` method on the object, which is responsible for decompressing or extracting the audio and routing it to the sound card and speakers. The act of playing an `AudioFile` object could feasibly be as simple as this:

```
from pathlib import Path

audio_file = SomePlayer(Path("/path/to/file"))
audio_file.play()
```

However, the process of decompressing and extracting an audio file is very different for different types of files. While `.wav` files are stored uncompressed, `.mp3`, `.wma`, and `.ogg` files each use totally different compression algorithms.

We can use inheritance with polymorphism to simplify the design. Each type of file can be represented by a different subclass of `AudioFile`, for example, `WavFile` and `MP3File`. Each of these would have a `play()` method that would be implemented differently for each file to ensure that the correct extraction procedure is followed. The media player object would never need to know which subclass of `AudioFile` it is referring to; it just calls `play()` and polymorphically lets the object take care of the actual details of playing. Let's look at a quick skeleton showing how this might work. We'll start with an abstract base class to define an interface that all subclasses will share:

```
from pathlib import Path
import abc


class AudioFile(abc.ABC):
    ext: str
```

```
    def __init__(self, filepath: Path) -> None:
        if not filepath.suffix == self.ext:
            raise ValueError(f"invalid file format {filepath.suffix!r}")
        self.filepath = filepath

    @abc.abstractmethod
    def play(self) -> None:
        ...
```

Here are some concrete subclasses:

```
class MP3File(AudioFile):
    ext = ".mp3"

    def play(self) -> None:
        print(f"playing {self.filepath} as mp3")


class WavFile(AudioFile):
    ext = ".wav"

    def play(self) -> None:
        print(f"playing {self.filepath} as wav")


class OggFile(AudioFile):
    ext = ".ogg"

    def play(self) -> None:
        print(f"playing {self.filepath} as ogg")
```

All audio files check to ensure that a valid extension was given upon initialization. If the filename doesn't end with a recognized correct name, it raises an exception (exceptions will be covered in detail in *Chapter 4*).

But did you notice how the `__init__()` method in the base class, `AudioFile`, is able to access the `ext` class variable from any of the subclasses? That's polymorphism at work. The `AudioFile` parent class merely has a type hint explaining to tools such as **mypy** that there will be an attribute named `ext`. It doesn't actually store a reference to the `ext` attribute. When the inherited method is used by a subclass, then the subclass' definition of the `ext` attribute is used. The type hint help

annotation-checking tools warn us about a class missing the attribute assignment.

In addition, each subclass of `AudioFile` implements `play()` in a different way (this example doesn't actually play the music; audio compression algorithms really deserve a separate book!). This is also polymorphism in action. The media player can use the exact same code to play a file, no matter what type it is; it doesn't care what subclass of `AudioFile` it is looking at. The details of decompressing the audio file are *encapsulated*. If we test this example, it works as we would hope:

```
>>> p_1 = MP3File(Path("Heart of the Sunrise.mp3"))
>>> p_1.play()
playing Heart of the Sunrise.mp3 as mp3
>>> p_2 = WavFile(Path("Roundabout.wav"))
>>> p_2.play()
playing Roundabout.wav as wav
>>> p_3 = OggFile(Path("Heart of the Sunrise.ogg"))
>>> p_3.play()
playing Heart of the Sunrise.ogg as ogg
>>> error = MP3File(Path("The Fish.mov"))
Traceback (most recent call last):
...
ValueError: invalid file format '.mov'
```

See how `AudioFile.__init__()` can check the file type without actually knowing which subclass it is referring to?

Polymorphism is actually one of the coolest things about object-oriented programming, and it makes some programming designs obvious that weren't possible in earlier paradigms.

Python makes polymorphism even easier because of a technique called "duck typing." Duck typing in Python allows us to use *any* object that provides the required behavior without forcing it to be a subclass of a common parent class. The dynamic nature of Python makes this trivial. The following example class does not extend `AudioFile`, but it can be interacted with in Python using the same interface:

```
class FlacFile:
    def __init__(self, filepath: Path) -> None:
        if not filepath.suffix == ".flac":
            raise ValueError("Not a .flac file")
        self.filepath = filepath
```

```
    def play(self) -> None:
        print(f"playing {self.filepath} as flac")
```

Our media player can play objects of the `FlacFile` class just as easily as objects of classes that extend `AudioFile`. They have the same API, but don't share any common base class (except for `object`).

Polymorphism is one of the most important reasons to use inheritance in many object-oriented contexts. Because any objects that supply the correct interface can be used interchangeably in Python, then duck typing reduces the need for polymorphic common superclasses. Inheritance is useful for sharing code, but if all that is being shared is the public interface, duck typing is all that is required.

In some cases, we can formalize this kind of duck typing using a `typing.Protocol` type definition. To make tools such as **mypy** aware of the expectations, we can define a number of functions using a formal `Protocol` type.

For example, we could define a common features between the `FlacFile` class and the `AudioFile` class hierarchy:

```
from typing import Protocol

class Playable(Protocol):
    def play(self) -> None:
        ...
```

Of course, just because an object satisfies a particular protocol (by providing required methods or attributes) this does not mean it will simply work in all situations. It also has to fulfill that interface in a way that makes sense in the overall system. It's possible that the `Cribbage` class also has a `play()` method, but doesn't work with a media player.

An alternative is to define a new type that's a union of the various types that are available. We might do the following, for example:

```
type Playable = AudioFile | FlacFile
```

This defines a new type, `Playable`, that summarizes the available types with a name that can then be used to define additional classes and objects. The `|` operator is the like the mathematical ∪

operator used for sets: the `Playable` type is the union of the two types. This puts a lightweight wrapper around the duck typing feature of Python.

Another useful feature of duck typing is that classes only need to provide those methods and attributes that are actually being accessed. For example, if we needed to create a fake file object from which we can read data, we could create a new object that has a `read()` method. We don't have to override the `write()` method if the code that is going to interact with the fake object will not be writing. Duck typing doesn't require implementing the entire interface of an object; it only needs to fulfill the protocol that is actually used.

Recall

Here are some key points from this chapter:

- A central object-oriented design principle is inheritance: a subclass can inherit aspects of a superclass, saving copy-and-paste programming. A subclass can extend the superclass to add features or specialize the superclass in other ways.
- Multiple inheritance is a feature of Python. The most common form is a host class with mixin class definitions. We can combine multiple classes leveraging the method resolution order to handle common features such as initialization.
- Polymorphism lets us create multiple classes that provide alternative implementations for fulfilling a contract. Because of Python's duck typing rules, any classes that have the right methods can substitute for each other.

Exercises

Look around you at some of the physical objects in your workspace and see if you can describe them in an inheritance hierarchy. Humans have been dividing the world into taxonomies like this for centuries, so it shouldn't be difficult. Are there any non-obvious inheritance relationships between classes of objects? If you were to model these objects in a computer application, what properties and methods would they share? Which ones would have to be polymorphically overridden? What properties would be completely different between them?

Now write some code. No, not for the physical hierarchy; that's difficult because physical items have more properties than methods. Think about a pet programming project you've wanted to tackle in the past year, but never gotten around to. For whatever problem you want to solve, try

to think of the inheritance relationships and then implement them. Make sure that you also pay attention to the sorts of relationships that you actually don't need to use inheritance for. Are there any places where you might want to use multiple inheritance? Are you sure? Can you see any place where you would want to use a mixin? Try to knock together a quick prototype. It doesn't have to be useful or even partially working. You've seen how you can test code using `python -i` already; just write some code and test it in the interactive interpreter. If it works, write some more. If it doesn't, fix it!

For an example application, refer to *Chapter 1*. The "*Reading a big script*" section shows a script that does a lot of processing. We describe some classes that seem to be part of the script processing. How can inheritance simplify these classes? What's similar about the various sections of the JSON-formatted documents? What's distinct about them?

Summary

We've gone from simple inheritance, one of the most useful tools in the object-oriented programmer's toolbox, all the way through to multiple inheritance — one of the more complicated. Inheritance can be used to add functionality to existing classes and built-in generics. Abstracting similar code into a parent class can help increase maintainability. Methods on parent classes can be called using the `super()` function, and argument lists must be formatted safely for these calls to work when using multiple inheritance.

In the next chapter, we'll cover the subtle art of handling exceptional circumstances.

4

Expecting the Unexpected

Software is rugged; computer systems, however, can be fragile. While the software is highly predictable, the runtime context can provide unexpected inputs and situations. Devices fail, networks are unreliable, and mere anarchy is let loose on our application. We need to have a way to handle the spectrum of failures that plague complicated systems such as modern computers.

There are two broad approaches to dealing with the unforeseen. One approach is to return a recognizable error-signaling value from a function. A value, such as `None`, could be used. Other library functions can then be used by an application to retrieve details of the erroneous condition. A variation on this theme is to pair the return value from an OS request with a success or failure indicator. The other approach is to interrupt the normal, sequential execution of statements and divert to statements designed to handle the exceptional situation. This second approach is what Python does: it eliminates the need to check return values for errors.

In this chapter, we will study **exceptions**, special error objects raised when a normal response is impossible. In particular, we will cover the following:

- How to cause an exception to occur
- How to recover when an exception has occurred
- How to handle different exception types in different ways

- Cleaning up when an exception has occurred
- Creating new types of exception
- Using the exception syntax for flow control

We'll start by looking at Python's concept of an `Exception`, and how exceptions are raised and handled.

Raising exceptions

Python's normal behavior is to execute statements in the order they are found, either in a file or interactively at the `>>>` prompt. A few statements, specifically `if`, `while`, and `for`, alter the simple top-to-bottom sequence of statement execution. Additionally, an exception can break the sequential flow of execution. When an exception is raised, it interrupts the sequential execution of statements.

In Python, the exception that's raised is also an object. There are many different exception classes available, and we can easily define more of our own. The one thing they all have in common is that they inherit from a built-in class called `BaseException`. (As a practical matter, we're far more interested in exceptions based on the `Exception` class.)

When an exception is raised, everything that was supposed to happen is preempted. Instead, exception handling replaces normal processing.

The easiest way to cause an exception to occur is to do something silly. Chances are you've done this already and seen the exception output. For example, any time Python encounters a line in your program that it can't understand, it raises the `SyntaxError` exception. Here's a common one:

```
>>> print "hello world"
Traceback (most recent call last):
...
    print "hello world"
    ^^^^^^^^^^^^^^^^^^
SyntaxError: Missing parentheses in call to 'print'. Did you mean
print(...)?
```

The `print()` function requires the arguments to be enclosed in parentheses. Note that we've used `...` in place of some details of the tracback message that aren't — for now — as interesting as the final exception.

In addition to `SyntaxError`, some other common exceptions are shown in the following example:

```
>>> x = 5 / 0
Traceback (most recent call last):
...
    x = 5 / 0
        ~~^~~
ZeroDivisionError: division by zero

>>> lst = [1,2,3]
>>> print(lst[3])
Traceback (most recent call last):
...
    print(lst[3])
          ~~~^^^
IndexError: list index out of range
>>> lst + 2
Traceback (most recent call last):
...
    lst + 2
    ~~~~^~~
TypeError: can only concatenate list (not "int") to list
>>> lst.add
Traceback (most recent call last):
...
    lst.add
AttributeError: 'list' object has no attribute 'add'

>>> d = {'a': 'hello'}
>>> d['b']
Traceback (most recent call last):
...
    d['b']
    ~^^^^^
KeyError: 'b'

>>> print(this_is_not_a_var)
Traceback (most recent call last):
...
    print(this_is_not_a_var)
          ^^^^^^^^^^^^^^^^
NameError: name 'this_is_not_a_var' is not defined
```

With a little hand-waving, we can partition these exceptions into roughly three categories:

- Some exceptions are indicators of something being clearly wrong with the way we're writing our program. Exceptions such as `SyntaxError` and `NameError` mean we need to find the indicated line number and fix the problem. An `ImportError`, for example, is either a misspelled name or a library that hasn't been added to the virtual environment.
- A few exceptions are indicators of something wrong in the Python runtime. As one example, there's a vague `SystemError` exception that can get raised when there are internal problems. Sometimes a reboot of the computer will resolve the problem. Other times, it might mean that it's time to download and install a newer Python.
- Most exceptions are design problems. We may fail to account for an edge case properly and sometimes try to compute an average of an empty list. This will result in `ZeroDivisionError`. When we find these, again, we'll have to go to the indicated line number. But once we've found the resulting exception, we'll need to work backward from there to find out what caused the problem that raised the exception. Somewhere there will be an object in an unexpected, not-designed-for state.

The bulk of these exceptions tend to arise near our program's interfaces. Any user input or OS request, including file operations, can encounter problems with the resources outside our program, leading to exceptions. We can subdivide these design problems further into two subgroups:

- External objects in an unusual or unanticipated state. This is common with files that aren't found because the path was spelled incorrectly, or directories that already exist because our application crashed earlier and we restarted it. These will often be some kind of `OSError` with a reasonably clear root cause. File-related problems will be visible as an (IOError) or one of its many subclasses. We need to expand our design to cover these cases, also.
- And there's also the (relatively small) category of hard-to-pin-down chaos. In the final analysis, a computer system is a lot of interconnected devices and any one of the components could behave badly. These are hard to anticipate, and it's harder still to plan a recovery strategy. For example, when working with a small IoT computer, there are few parts, but it may be installed in a challenging physical environment. When working with an enterprise server farm with thousands of components, a 0.1% failure rate means something is always broken, and a root cause may be hardware instead of our programming. We may need to expand our logging and error display to expose the underlying problem.

You may have noticed that most of Python's built-in exceptions end with the name `Error`. In Python, the words **error** and **exception** are used almost interchangeably. Errors are sometimes considered

more dire than exceptions, but they are dealt with in the same way.

All error classes in the preceding example have `Exception` as their base class. The `SystemExit` exception is an example of the few exceptions based on `BaseException`, instead of `Exception`. There's no sensible way for any code we write to handle this exception; this exception is used internally to make the Python runtime stop.

Raising an exception

We'll get to responding to such exceptions in a minute. First, we need to discover what we should do if we're writing a program that needs to inform the user or a calling function of some exception. An example is invalid input values. We can use the same mechanism for notification that Python uses. Here's a simple class that adds items to a list only if they are even-numbered integers:

```
class EvenOnly(list[int]):
    def append(self, value: int) -> None:
        match value:
            case int():
                if value % 2 != 0:
                    raise ValueError("Only even numbers can be added")
            case _:
                raise TypeError("Only integers can be added")
        super().append(value)
```

This class extends the built-in `list` type, as we discussed in *Chapter 2*. The type hint suggests our intent to create a list of integer objects only. To do the appending, we've overridden the `append` method to check two conditions that ensure each item is an even integer. We first check whether the input is an instance of the `int` type. (This can also be done with the `isinstance()` function.) Then we use the modulo operator to ensure it is divisible by two. If either of the two conditions is not met, the `raise` statement causes an exception to occur.

The `raise` keyword is followed by the object being raised as an exception. In the preceding example, two objects are constructed from the built-in `TypeError` and `ValueError` classes. The raised object could just as easily be an instance of a new `Exception` class we create ourselves (we'll see how shortly), an exception that was defined elsewhere, or even an existing `Exception` object that has been previously raised and handled.

If we test this class in the Python interpreter, we can see that it is outputting useful error information

when exceptions occur:

```
>>> e = EvenOnly()
>>> e.append("a string")
Traceback (most recent call last):
...
TypeError: Only integers can be added

>>> e.append(3)
Traceback (most recent call last):
...
ValueError: Only even numbers can be added
>>> e.append(2)
```

While this class is effective for demonstrating exceptions in action, it isn't very good at its job. It is still possible to get other values into the list using index notation, slice notation, or the `insert()` method. These additional behaviors can be avoided by overriding other appropriate methods, some of which are magic double-underscore methods. To be really complete, we'd need to override methods such as `extend()`, `insert()`, `__setittem__()`, and even `__init__()` to be sure things start off correctly.

The effects of an exception

When an exception is raised, it appears to stop program execution immediately. Any lines that were supposed to run after the exception is raised are not executed, and unless the exception is handled by an `except` clause, the program will exit with an error message. We'll examine unhandled exceptions first, and then take a close look at handling exceptions.

Take a look at this basic function:

```
from typing import NoReturn


def never_returns() -> NoReturn:
    print("I am about to raise an exception")
    raise Exception("This is always raised")
    print("This line will never execute")
    return "I won't be returned"
```

We've included the `NoReturn` type hint for this function. This helps ease **mypy**'s worry that there's

no way for this function to reach the end and return a string value. The type hint states, formally, that the function isn't expected to reach the `return` statement at the end.

If we execute this function, we see that the first `print()` call is executed and then the exception is raised. The second `print()` function call is never executed, nor is the `return` statement. Here's what it looks like:

```
>>> never_returns()
Traceback (most recent call last):
...
    never_returns()
...
    raise Exception("This is always raised")
Exception: This is always raised
```

Furthermore, if we have a function that calls another function that raises an exception, nothing is executed in the first function after the point where the second function's exception was raised. Raising an exception stops all execution right up through the function call stack until it is either handled or forces the interpreter to exit. To demonstrate, let's add a second function that calls the `never_returns()` function:

```
def call_exceptor() -> None:
    print("call_exceptor starts here...")
    never_returns()
    print("an exception was raised...")
    print("...so these lines don't run")
```

When we call this function, we see that the first `print` statement executes, as well as the first line in the `never_returns()` function. But once the exception is raised, nothing else executes:

```
>>> call_exceptor()
Traceback (most recent call last):
...
    call_exceptor()
...
    never_returns()
...
    raise Exception("This is always raised")
Exception: This is always raised
```

Note that some clever reasoning is required to recognize what the call to `never_returns()` does to the processing in the `call_exceptor()` function. Based on previous examples, it may seem like `call_exceptor()` is better described as a `NoReturn` function. When we try this, we can get a warning from **mypy**. It's far from obvious to a software tool that `never_returns()` always raises an exception. The amount of careful reasoning about the code would be extraordinary to discover an aspect that's a symptom of bad design.

We can control how exceptions propagate from the initial `raise` statement. We can react to and deal with the exception inside either of the functions in the call stack. To understand this idea, look at the output from the preceding unhandled exception, called a **traceback**. The list of functions shows the call stack. The command line ("`<module>`" is the name used when there's no input file) called the `call_exceptor()` function. Then the `call_exceptor()` function called the `never_returns()` function. Inside the `never_returns()` function, the exception is raised.

An unhandled exception propagates up through the call stack. The exception started in the `never_returns()` function, and was ignored. It wound up in the `call_exceptor()` function, where it was also ignored. From there, it went up one more level to the main interpreter, which, not knowing what else to do with it, gave up and printed the traceback object.

Interactive Python returns to the `>>>` prompt. When running a script, the interpreter stops.

Handling exceptions

Now let's look at the tails side of the exception coin. When an exception is raised, how should our code react to or recover from it? We handle exceptions by wrapping any code that might throw an exception inside a `try...except` clause. The most basic syntax looks like this:

```
def handler() -> None:
    try:
        never_returns()
        print("Never executed")
    except Exception as ex:
        print(f"I caught an exception: {ex!r}")
    print("Executed after the exception")
```

If we run this simple script using our existing `never_returns()` function — which, as we know very well, always throws an exception — we get this output:

```
>>> handler()
I am about to raise an exception
I caught an exception: Exception('This is always raised')
Executed after the exception
```

The `never_returns()` function happily informs us that it is about to raise an exception and raises it. The `handler()` function's `except` clause catches the exception. Once caught, we are able to clean up after ourselves (in this case, by outputting that we are handling the situation), and continue on our way. The remainder of the code in the `never_returns()` function remains unexecuted, but the code in the `handler()` function after the `try:` statement is able to recover and continue.

Note the indentation around `try` and `except`. The `try` clause wraps any code that might throw an exception we want to handle. The `except` clause is then back on the same indentation level as the `try` line. Any code to handle the exception is indented inside the `except` clause. Then normal code resumes at the original indentation level.

The preceding code uses the overly broad `Exception` class to match any type of exception. What if we were writing some code that could raise either `TypeError` or `ZeroDivisionError`? We might need to catch `ZeroDivisionError` because it reflects a known object state, but let any other exceptions propagate to the console because they reflect bugs we need to catch and kill. Can you guess the syntax?

Here's a rather silly function that does just that:

```
def funny_division(divisor: float) -> str | float:
    try:
        return 100 / divisor
    except ZeroDivisionError:
        return "Zero is not a good idea!"
```

This function does a simple computation. We've provided the type hint of `float` for the `divisor` parameter. At runtime, the function using this can also provide an integer, and ordinary Python type conversions will work. The type-checking tools are aware of how integers are promoted to floats when needed, and how the true division operator `/` returns a float result.

We do, however, have to be very clear about the return types. If an exception is not raised, we'll compute and return a floating result. If there is a `ZeroDivisionError` exception, it will be handled, and we'll return a string result. This is a union of type types, a subject we'll return to in *Chapter 7*.

What about other types of exceptions? Let's try it and see:

```
>>> print(funny_division(0))
Zero is not a good idea!
>>> print(funny_division(50.0))
2.0
>>> print(funny_division("hello"))
Traceback (most recent call last):
...
    print(funny_division("hello"))
          ^^^^^^^^^^^^^^^^^^^^^^
...
    return 100 / divisor
           ~~~~^~~~~~~~~
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

The first line of output shows that if we enter `0`, we get properly mocked. If we call with a valid number, it operates correctly. Yet if we enter a string (you were wondering how to get a `TypeError`, weren't you?), it fails with an unhandled exception.

Starting with 3.14, Python will issue warnings about the "bare except" syntax. Using `except:` with no exception class to match will be deprecated because it tends to prevent an application from simply crashing when it should. For more information on the removal of this feature, see *PEP 760*, (`https://peps.python.org/pep-0760/`).

The bare `except:` syntax is actually the same as using `except BaseException:`, which attempts to handle system-level exceptions that are often impossible to recover from. Indeed, matching this exception base class can make it impossible to crash your application when it's misbehaving. It's possible, for example, to catch the **Ctrl + C** keyboard sequence that normally stops a program from running; doing this is a very bad idea.

We can even name two or more different exceptions and handle them with the same code. Here's an example that raises three different types of exceptions. It handles `TypeError` and `ZeroDivisionError` with the same exception handler, but it may also raise a `ValueError` error if you supply the number `13`:

```
def funnier_division(divisor: int) -> str | float:
    try:
        if divisor == 13:
            raise ValueError("13 is an unlucky number")
        return 100 / divisor
    except (ZeroDivisionError, TypeError):
        return "Enter a number other than zero"
```

:

We've included multiple exception classes in the `except` clause. This lets us handle a variety of conditions with a common handler. Here's how we can test this with a bunch of different values:

```
>>> for val in (0, "hello", 50.0):
...     print(f"Testing {val!r}:", end=" ")
...     print(funnier_division(val))
...
Testing 0: Enter a number other than zero
Testing 'hello': Enter a number other than zero
Testing 50.0: 2.0
>>> val = 13
>>> print(funnier_division(val))
Traceback (most recent call last):
...
    print(funnier_division(val))
          ^^^^^^^^^^^^^^^^^^^^
...
    raise ValueError("13 is an unlucky number")
ValueError: 13 is an unlucky number
```

The `for` statement iterates over several test inputs and prints the results. If you're wondering about that `end` parameter in the `print` function, it just turns the default trailing newline into a space so that it's joined with the output from the next line.

The number `0` and the string are both caught by the `except` clause, and a suitable error message is printed. The exception from the number `13` is not caught because it is a `ValueError`, which was not included in the types of exceptions being handled. How do we catch different exceptions and do different things with them?

We don't need any new syntax to deal with the multiple-handler case. It's possible to stack the `except` clauses, and only the first match will be executed.

There is one other thing we can do in a handler: we can re-raise the current exception. This lets us do a partial intervention — maybe write some helpful debugging context — before allowing the exception to bubble up to a handler. The `raise` keyword, with no arguments, will re-raise the current exception. Observe the following code:

```
def funniest_division(divisor: int) -> str | float:
    try:
        if divisor == 13:
            raise ValueError("13 is an unlucky number")
        return 100 / divisor
    except ZeroDivisionError:
        return "Enter a number other than zero"
    except TypeError:
        return "Enter a numerical value"
    except ValueError:
        print("No, No, not 13!")
        raise
```

The last line re-raises the `ValueError` error. After outputting `No, No, not 13!`, it will raise the exception again. Because it's unhandled, we'll still get the original stack trace on the console.

If we stack exception clauses like we did in the preceding example, only the first matching clause will be run, even if more than one of them fits. How can more than one clause match? Remember that exceptions are objects, and can therefore be subclassed. As we'll see in the next section, most exceptions extend the `Exception` class. If we have an `except` clause to match `Exception` **before** we have a clause to match `TypeError`, then only the `Exception` handler will be executed. We need to stack them from most specific to most generic.

This can come in handy in cases where we want to handle some exceptions specifically, and then handle all remaining exceptions as a more general case. We can list `Exception` in its own clause after catching all the specific exceptions and handle the general case there.

Often, when we catch an exception, we need a reference to the `Exception` object itself. This most often happens when we define our own exceptions with custom arguments, but can also be relevant with standard exceptions. Most exception classes accept a set of arguments in their constructor, and we might want to access those attributes in the exception handler. If we define our own `Exception` class, we can even call custom methods on it when we catch it. The syntax for capturing an exception as a variable uses the `as` keyword:

```
>>> try:
...     raise ValueError("This is an argument")
... except ValueError as e:
...     print(f"The exception arguments were {e.args}")
...
The exception arguments were ('This is an argument',)
```

When we run this snippet, it prints out the string argument that we passed into `ValueError` upon initialization.

We've seen several variations on the syntax for handling exceptions. We have another design feature: execute code whether or not an exception has occurred. We also haven't seen how to specify code that should be executed **only** if an exception does **not** occur. Two more keywords, `finally` and `else`, provide some additional execution paths. Neither one takes any extra arguments.

We'll show an example with the `finally` clause. For the most part, we often use context managers instead of exception blocks as a cleaner way to implement a finalization that occurs whether or not an exception interrupted processing. The idea is to encapsulate responsibility for finalization in the context manager.

The following example iterates through a number of exception classes, raising an instance of each. Then some not-so-complicated exception-handling code runs that illustrates the newly introduced syntax:

```
some_exceptions = [ValueError, TypeError, IndexError, None]

for choice in some_exceptions:
    try:
        print(f"\nRaising {choice}")
        if choice:
            raise choice("An error")
        else:
            print("no exception raised")
    except ValueError:
        print("Caught a ValueError")
    except TypeError:
        print("Caught a TypeError")
    except Exception as e:
        print(f"Caught some other error: {e.__class__.__name__}")
    else:
```

```
        print("This code called if there is no exception")
    finally:
        print("This cleanup code is always called")
```

If we run this example, which illustrates almost every conceivable exception-handling scenario, we'll see the following output:

```
[pycon]
Raising <class 'ValueError'>
Caught a ValueError
This cleanup code is always called

Raising <class 'TypeError'>
Caught a TypeError
This cleanup code is always called

Raising <class 'IndexError'>
Caught some other error: IndexError
This cleanup code is always called

Raising None
no exception raised
This code called if there is no exception
This cleanup code is always called
```

Note how the `print()` function in the `finally` clause is executed no matter what happens. This is one way to perform certain tasks after our code has finished running (even if an exception has occurred). Some common examples include the following:

- Cleaning up an open database connection
- Closing an open file
- Sending a closing handshake over the network

All of these are more commonly handled with context managers, one of the topics of *Chapter 9*.

While obscure, the `finally` clause is executed after the `return` statement inside a `try` clause. While this can be exploited for post-`return` processing, it can also be confusing to folks reading the code.

Also, pay attention to the output when no exception is raised: both the `else` and the `finally` clauses are executed. The `else` clause will not be executed if an exception is caught and handled.

The `finally` clause is always executed. We'll see more on this when we discuss using exceptions as flow control later.

Clearly, the `else`, and `finally` clauses are optional. It turns out you can have a `finally` clause without any other clauses to specify some processing that will happen even if there's an unhandled exception. As noted earlier, a context manager is a less tricky way to do this kind of thing.

Make sure the order of the `except` clauses has classes start with the most specific subclasses and end with the most generic superclasses.

The exception hierarchy

We've already seen several of the most common built-in exceptions. As we noticed earlier, most exceptions are subclasses of the `Exception` class. But this is not true of all exceptions. The `Exception` class actually extends a class called `BaseException`.

There are three key built-in exception classes, `SystemExit`, and `KeyboardInterrupt`, `GeneratorExit`, that derive directly from the `BaseException` class instead of the `Exception` class. A fourth class, `BaseExceptionGroup`, is also defined here; handling this isn't fraught with complications like the other three direct subclassess of `BaseException`.

The `SystemExit` exception is raised whenever the program exits naturally. This is typically raised by the `sys.exit()` function; for example, when the user selected an exit menu item, clicked the **Close** button on a window, or entered a command to shut down a server, or the OS sent a signal to the application to terminate. This exception is designed to simply ignore the processing that's going on and exit.

If we try to handle the `SystemExit` exception, we must re-raise it, since silencing it could stop the program from exiting. Imagine a web service with a bug that is holding database locks and can't be stopped without rebooting the server.

The `KeyboardInterrupt` exception is common in command-line programs. It is raised when the user explicitly interrupts program execution with an OS-dependent key combination (normally, **Ctrl + C**). For Linux and macOS users, the `kill -2 <pid>` command will also work. This is a standard way for the user to deliberately interrupt a running program and, like the `SystemExit` exception, it should almost always respond by terminating the program. Silencing the exception may make a badly behaving program unstoppable without a reboot.

There are more OS signals available. The `SIGINT` signal sent by **Ctrl + C** raises an exception by default. Several other signals are ignored by default. The `signal` module provides ways to respond to the other OS signals.

As with the previous two, the `GeneratorExit` exception isn't really an error. It's a signal that a generator or coroutine is finishing. Generators and coroutines can't return a special value, so they raise an exception.

Figure 4.1 illustrates the exception hierarchy:

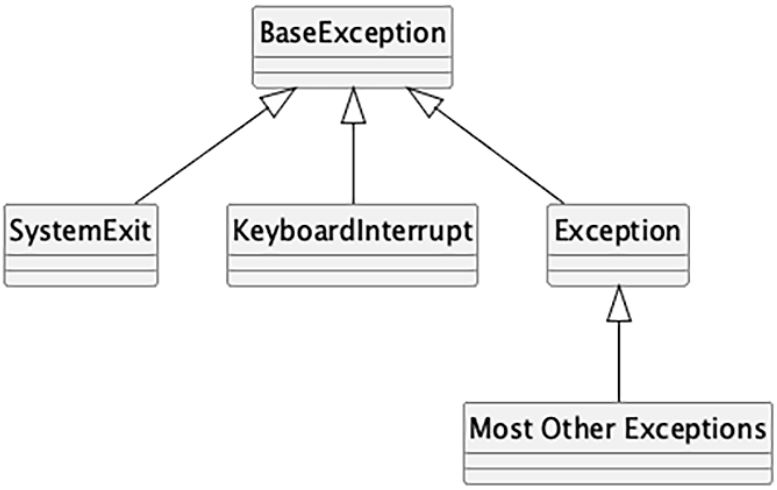


Figure 4.1: Exception hierarchy

Almost universally, the most general kind of exception is the `Exception` class. Avoid the `BaseException` class, since it's more likely to cause problems than solve them.

Defining our own exceptions

Occasionally, when we want to raise an exception, we find that none of the built-in exceptions are suitable. The distinction is often focused on how applications must handle the exception; when we introduce a new exception it must be because there will be distinct processing in a handler.

There's no good reason to define an exception that's handled exactly like `ValueError`; we can use `ValueError`. Luckily, it's trivial to define new exception subclasses of our own. The name of the class is usually designed to communicate what went wrong, and we can provide arbitrary arguments in the initializer to include additional information.

All we have to do is inherit from the `Exception` class or one of the existing exceptions that's semantically similar. We don't even have to add any content to the class!

Here's a simple exception we might use in a banking application:

```
class InvalidWithdrawal_1(ValueError):
    pass
```

Here's how it looks when we raise this exception with a string argument value:

```
>>> raise InvalidWithdrawal_1("You don't have $50 in your account")
Traceback (most recent call last):
...
banking.InvalidWithdrawal_1: You don't have $50 in your account
```

We are able to pass an arbitrary number of arguments into the exception. Often a string message is used, but any object that might be useful in a later exception handler can be stored. The `Exception.__init__()` method is designed to accept any arguments and store them as a tuple in an attribute named `args`. This makes exceptions easier to define without needing to override `__init__()`.

Of course, if we do want to customize the initializer, we are free to do so. Here's a revision to the above exception whose initializer accepts the current balance and the amount the user wants to withdraw. In addition, it adds a method to calculate how overdrawn the request is:

```
from decimal import Decimal


class InvalidWithdrawal(ValueError):
    def __init__(self, balance: Decimal, amount: Decimal) -> None:
        super().__init__(f"account doesn't have ${amount}")
        self.amount = amount
        self.balance = balance

    def overage(self) -> Decimal:
        return self.amount - self.balance
```

Since we're working with currency, we've imported the `Decimal` class of numbers. We can't use Python's default `int`or `float` types for money where there are a fixed number of decimal places

and exquisitely complex rounding rules that assume exact decimal arithmetic.

(Also note that an account number or any other **personally identifiable information**, or **PII**, is not part of the exception. Bankers frown on using PII in a way that could be exposed in a log or a traceback message.)

Here's an example of creating an instance of this exception:

```
>>> raise InvalidWithdrawal(Decimal('28.63'), Decimal('42.00'))
Traceback (most recent call last):
...
banking.InvalidWithdrawal: account doesn't have $42.00
```

Let's assume there's a `do_transfer()` function that either computes the new balance or raises an exception if the transfer is invalid. Here's how we would handle an `InvalidWithdrawal` exception if one was raised:

```
>>> balance = Decimal('28.63')
>>> transfer = Decimal('42.00')
>>> try:
...     new_balance = do_transfer(balance, transfer)
... except InvalidWithdrawal as ex:
...     print("I'm sorry, but your withdrawal is "
...           "more than your balance by "
...           f"${ex.overage()}")
...
I'm sorry, but your withdrawal is more than your balance by $13.37
```

Here we see the `as` keyword to save the exception in a local variable, `ex`. By convention, most Python coders assign the exception a variable such as `ex`, `exc`, or `exception`; although, as usual, you are free to call it `the_exception_raised_above` or `dogs_breakfast` if you prefer.

There are many reasons for defining our own exceptions. It is often useful to add information to the exception or log it in some way. But the utility of custom exceptions truly comes to light when creating a framework, library, or API that is intended for access by other programmers. In that case, be careful to ensure your code is raising exceptions that make sense to the client programmer. Here are some criteria:

- They should clearly describe what went on. The `KeyError` exception, for example, provides the key that could not be found.

- The client programmer should be able to easily see how to fix the error (if it reflects a bug in their code) or handle the exception (if it's a situation they need to be made aware of).
- The handling should be distinct from other exceptions. If the handling is the same as an existing exception, reusing the existing exception is best.

Now that we've looked at raising exceptions and defining new exceptions, we can look at some of the design considerations that surround exceptional data and responding to problems. There are a number of alternative design choices, and we'll start with the idea that exceptions, in Python, can be used for a number of things that aren't — strictly speaking — erroneous.

Exceptions aren't exceptional

Novice programmers tend to think of exceptions as only useful for really profoundly earth-shattering, program-ending circumstances. However, the definition of exceptional circumstances can be subject to interpretation. Consider the following two functions:

```
def divide_with_exception(dividend: int, divisor: int) -> None:
    try:
        print(f"{dividend / divisor=}")
    except ZeroDivisionError:
        print("You can't divide by zero")


def divide_with_if(dividend: int, divisor: int) -> None:
    if divisor == 0:
        print("You can't divide by zero")
    else:
        print(f"{dividend / divisor=}")
```

These two functions behave identically. If `divisor` is 0, an error message is printed; otherwise, a message printing the result of the division is displayed. In the second function, `divide_with_if()`, the test for a valid division is a relatively simple-looking expression, `divisor == 0`. In some cases, the test for valid data can be rather complex. In some cases, it may involve computing some intermediate results. In the worst cases, the test for "will this work?" involves using a number of other methods of a class to — in effect — dry-run the operation to see whether there would be an error along the way. This amounts to doing the work twice: once to see whether it's feasible, then doing it with intent to save the results.

Python programmers prefer to avoid doing the work twice. They tend to follow a model summarized

by "**it's easier to ask for forgiveness than permission**," sometimes abbreviated EAFP. The point is to execute code and then deal with anything that goes wrong. The alternative is described as "**look before you leap**," often abbreviated LBYL. This is generally less popular. The main reason for this is it shouldn't be necessary to burn CPU cycles looking for an unusual situation that is rare. Python classes generally check for bad data; writing a redundant check is clearly wasteful.

This is why Python programmers use exceptions for all kinds of exceptional circumstances, even if those circumstances are only a little bit exceptional. Taking this argument further, exception syntax can be effective for flow control. A `try` statement is somewhat like an `if` statement, with exceptions used for decision-making. It's common practice to simply retrieve an item from a dictionary or a list without checking first to see whether the key exists or the list index is valid.

For an extended example, imagine an inventory application. When a customer makes a purchase, the item can either be available or be out of stock. A method will update the inventory counts when the item is available. Being out of stock is a perfectly normal thing to happen in an inventory application. It is certainly not an erroneous circumstance. But what does a function do when an item is out of stock? Return a string saying "out of stock"? Return a special error code? These uses of code strings or numeric values can be messy. Further, the collaborating functions must check for these coded return values.

Instead, we can raise an `OutOfStock` exception and use the `try` statement to direct program flow control.

In addition, we want to make sure we don't sell the same item to two different customers, or sell an item that isn't in stock yet. One way to facilitate this is to lock each type of item to ensure only one person can update it at a time. The user must lock the item, manipulate the item (purchase, add stock, count items left...), and then unlock the item. (This is, in effect, a context manager, one subject of *Chapter 9*.)

Here's an incomplete `Inventory` example. We'll start with two exceptions, then show a class with with docstrings that describe what some of the methods should do:

```
class OutOfStock(Exception):
    pass


class InvalidItemType(Exception):
    pass
```

```
class Inventory:
    def __init__(self, stock: list[ItemType]) -> None:
        pass

    def lock(self, item_type: ItemType) -> None:
        """Context Entry.
        Lock the item type so nobody else can manipulate the
        inventory while we're working."""
        pass

    def unlock(self, item_type: ItemType) -> None:
        """Context Exit.
        Unlock the item type."""
        pass

    def purchase(self, item_type: ItemType) -> int:
        """If the item is not locked, raise an
        ValueError because someting went wrong.
        If the item_type does not exist,
          raise InvalidItemType.
        If the item is currently out of stock,
          raise OutOfStock.
        If the item is available,
          subtract one item; return the number of items left.
        """
        # Mocked results.
        if item_type.name == "Widget":
            raise OutOfStock(item_type)
        elif item_type.name == "Gadget":
            return 42
        else:
            raise InvalidItemType(item_type)
```

We could hand this object prototype to a developer and have them implement the methods to do exactly as they say while we work on the code needed to make a purchase. We'll use Python's robust exception handling to consider different processing paths, depending on how the purchase was made. We can even write a test case to be sure that there's no question about how this class should work.

Here's a definition of `ItemType`, just to round out the example:

```
class ItemType:
    def __init__(self, name: str) -> None:
        self.name = name
        self.on_hand = 0
```

Here's an interactive session using this `Inventory` class:

```
>>> widget = ItemType("Widget")
>>> gadget = ItemType("Gadget")
>>> inv = Inventory([widget, gadget])

>>> item_to_buy = widget
>>> inv.lock(item_to_buy)
>>> try:
...     num_left = inv.purchase(item_to_buy)
... except InvalidItemType:
...     print(f"Sorry, we don't sell {item_to_buy.name}")
... except OutOfStock:
...     print("Sorry, that item is out of stock.")
... else:
...     print(f"Purchase complete. There are {num_left} {item_to_buy.name}s
left")
... finally:
...     inv.unlock(item_to_buy)
...
Sorry, that item is out of stock.
```

All the possible exception-handling clauses are used to ensure that the correct actions happen at the correct time. Even though the `OutOfStock` exception is not a terribly exceptional circumstance, we are able to use an exception to handle it suitably. This same code could be written with an `if...elif...else` structure, but it wouldn't be as easy to read or maintain.

We can also use exceptions to pass messages between different methods. For example, if we wanted to inform the customer as to what date the item is expected to be in stock again, we could ensure our `OutOfStock` class requires a `back_in_stock` parameter when it is constructed. Then, when we handle the exception, we can check that value and provide additional information to the customer. The information attached to the object can be easily passed between two different parts of the program. The exception could even provide a method that instructs the inventory object to reorder or backorder an item.

Using exceptions for flow control can make for some handy program designs. The important thing to take from this discussion is that exceptions are not a bad thing that we should try to avoid. Having an exception occur does not mean that you should have prevented this exceptional circumstance from happening. Rather, it is a powerful way to communicate information between two sections of code that may not be directly calling each other.

Recall

Some key points in this chapter are the following:

- Raising an exception happens when something goes wrong. We looked at division by zero as an example. Exceptions can also be raised with the `raise` statement.
- The effects of an exception are to interrupt the normal sequential execution of statements. It saves us from having to write a lot of `if` statements to check to see whether things can possibly work or whether something actually failed.
- Handling exceptions is done with the `try:` statement, which has an `except:` clause for each kind of exception we want to handle.
- The exception hierarchy follows object-oriented design patterns to define a number of subclasses of the `Exception` class we can work with. Some additional exceptions, `SystemExit` and `KeyboardInterrupt`, are not subclasses of the `Exception` class; handling these exceptions can introduce risks and doesn't solve very many problems, so we generally ignore them.
- Defining our own exceptions is a matter of extending the `Exception` class. This makes it possible to define exceptions with very specific semantics.

Exercises

If you've never dealt with exceptions before, the first thing you need to do is look at any old Python code you've written and notice if there are places you should have been handling exceptions. How would you handle them? Do you need to handle them at all? Sometimes, letting the exception propagate to the console is the best way to communicate to the user, especially if the user is also the script's coder. Sometimes, you can recover from the error and allow the program to continue. Sometimes, you can only reformat the error into something the user can understand and display it to them.

Some common places to look are file I/O (is it possible your code will try to read a file that doesn't exist?), mathematical expressions (is it possible that a value you are dividing by is zero?), list indices (is the list empty?), and dictionaries (does the key exist?).

Ask yourself whether you should ignore the problem, handle it by checking values first, or handle it with an exception. Pay special attention to areas where you might have used `finally` and `else` to ensure the correct code is executed under all conditions.

Try to think of places in your code where you can raise exceptions. It can be in code you've written or are working on, or you can write a new project as an exercise. You'll probably have the best luck designing a small framework or API that is meant to be used by other people; exceptions are a terrific communication tool between your code and someone else's. Remember to design and document any self-raised exceptions as part of the API, or they won't know whether or how to handle them!

For an example application, refer to *Chapter 1*. The "*Reading a big script*" section shows a script that does a lot of processing. What kinds of things can go wrong in this processing? Certainly, files can be unreadable. What about missing keys because of unexpected JSON structures? What about unexpected values for the outcomes of various tests? What kinds of exceptions can be handled, and what exceptions should stop the processing?

Summary

In this chapter, we went into the gritty details of raising, handling, defining, and manipulating exceptions. Exceptions are a powerful way to communicate unusual circumstances or error conditions without requiring a calling function to explicitly check return values. There are many built-in exceptions, and raising them is trivially easy. There are several different syntaxes for handling different exception events.

In the next chapter, everything we've studied so far will come together as we discuss how object-oriented programming principles and structures should best be applied in Python applications.

Join our community Discord space

Join our Python Discord workspace to discuss and know more about the book: `https://packt.link/dHrHU`

5

When to Use Object-Oriented Programming

In previous chapters, we've covered many of the defining features of object-oriented programming. We now know some principles and paradigms of object-oriented design, and we've covered the syntax of object-oriented programming in Python.

Yet, we don't know exactly how and, especially, when to utilize these principles and syntax in practice. In this chapter, we'll discuss some useful applications of the knowledge we've gained, looking at some new topics along the way:

- How to recognize objects
- Data and behaviors, once again
- Wrapping data behaviors using properties
- The Don't Repeat Yourself principle and avoiding repetition

We'll start this chapter with a close look at the nature of objects and their internal state. There are cases when there's no state change, and a class definition isn't desirable.

Treat objects as objects

This may seem obvious: you should generally give separate objects in your problem domain a special class in your code. We've seen examples of this in the examples in previous chapters: first, we identify objects in the problem, and then model their data and behaviors.

Identifying objects is a very important task in object-oriented analysis and programming. But it isn't always as easy as counting the nouns in short paragraphs that, frankly, the authors have constructed explicitly for that purpose. Remember, objects are things that have both data and behavior. If we are working only with data, we are often better off storing it in a list, set, dictionary, or other Python data structure (which we'll be covering thoroughly in *Chapter 8*). On the other hand, if we are working only with behavior, but no stored data, a simple function is more suitable.

An object, however, has both data and behavior. Proficient Python programmers use built-in data structures unless (or until) there is an obvious need to define a class. There is no reason to add an extra level of complexity if it doesn't help organize our code. On the other hand, the need is not always self-evident.

We can often start our Python programs by storing data in a few variables. As the program expands, we will later find that we are passing the same set of related variables to a set of functions. This is the time to think about grouping both variables and functions into a class.

For example, if we are designing a program to model polygons in two-dimensional space, we might start with each polygon represented as a list of points. The points would be modeled as two tuples (x, y) describing where that point is located. This is all data, stored in a set of nested data structures (specifically, a list of tuples). We can (and often do) start hacking at the command prompt:

```
>>> square = [(1,1), (1,2), (2,2), (2,1)]
```

Now, if we want to calculate the distance around the perimeter of the polygon, we need to sum the distances between each point. To do this, we need a function to calculate the distance between two points. Here are two such functions:

```
>>> from math import hypot

>>> def distance(p_1, p_2):
...     return hypot(p_1[0] - p_2[0], p_1[1] - p_2[1])

>>> def perimeter(polygon):
...     pairs = zip(polygon, polygon[1:] + polygon[:1])
...     return sum(
...         distance(p1, p2) for p1, p2 in pairs
...     )
```

We can exercise the functions to check our work:

```
>>> perimeter(square)
4.0
```

This is a start, but it's not completely descriptive of the problem domain. With some study, we can work out what type the value of `polygon` might be. We need to read the entire batch of code to see how the two functions work together.

We can add type hints to help clarify the intent behind each function. It helps to switch to writing a module based on the results of experiments at the >>> REPL prompt. The result looks like this:

```
from math import hypot

type Point = tuple[float, float]


def distance(p_1: Point, p_2: Point) -> float:
    return hypot(p_1[0] - p_2[0], p_1[1] - p_2[1])


type Polygon = list[Point]


def perimeter(polygon: Polygon) -> float:
    pairs = zip(polygon, polygon[1:] + polygon[:1])
    return sum(distance(p1, p2) for p1, p2 in pairs)
```

We've added two type definitions, `Point` and `Polygon`, to help clarify our intentions. The definition of `Point` shows how we'll use the built-in `tuple` class to contain two floating-point values. The

definition of `Polygon` shows how the built-in `list` class builds on the `Point` class. (Both of these hints are examples of generic types, one of the topics of *Chapter 7*.)

Now, as object-oriented programmers, we clearly recognize that a `polygon` class could encapsulate the list of points (data) and the `perimeter` function (behavior). Further, a `Point` class, such as we defined in *Chapter 2*, might encapsulate the `x` and `y` coordinates and the `distance` method. The question is: is it valuable to do this?

For the previous code, maybe yes, maybe no. With our recent experience in object-oriented principles, we can write an object-oriented version in record time. Let's compare them as follows:

```
from math import hypot


class Point:
    def __init__(self, x: float, y: float) -> None:
        self.x = x
        self.y = y

    def distance(self, other: "Point") -> float:
        return hypot(self.x - other.x, self.y - other.y)


class Polygon:
    def __init__(self) -> None:
        self.vertices: list[Point] = []

    def add_point(self, point: Point) -> None:
        self.vertices.append(point)

    def perimeter(self) -> float:
        pairs = zip(self.vertices, self.vertices[1:] + self.vertices[:1])
        return sum(p1.distance(p2) for p1, p2 in pairs)
```

There seems to be almost twice as much code here as there was in our earlier version, although we could argue that the `add_point()` method is not strictly necessary. We could also try to insist on using `_vertices` to discourage the use of the attribute, but the use of leading _ variable names doesn't seem to really solve any problems we might have.

Now, to understand the differences between the two classes a little better, let's compare the two APIs in use. Here's how to calculate the perimeter of a square using the object-oriented code:

```
>>> square = Polygon()
>>> square.add_point(Point(1,1))
>>> square.add_point(Point(1,2))
>>> square.add_point(Point(2,2))
>>> square.add_point(Point(2,1))
>>> square.perimeter()
4.0
```

That's fairly succinct and easy to read, you might think, but let's compare it to the function-based code:

```
>>> square = [(1,1), (1,2), (2,2), (2,1)]
>>> perimeter(square)
4.0
```

Hmm, maybe the object-oriented API isn't so compact! Our first, hacked-in version is the shortest. How do we know what the list of tuples is supposed to represent? How do we remember what kind of object we're supposed to pass into the `perimeter` function? We needed some documentation to explain how the first set of functions should be used.

We haven't added docstrings, but you can see how the volume of code grows as we strive for clarity.

Code length is not a good indicator of code complexity. Some programmers get hung up on complicated *one-liners* that do an incredible amount of work in one line of code. This can be a fun exercise, but the result is often unreadable, even to the original author the following day. Minimizing the amount of code can make a program easier to read, but there are limits.

No one wins at code golf.

Minimizing the volume of code is rarely desirable.

Luckily, this trade-off isn't necessary. We can make the object-oriented `Polygon` API as easy to use as the functional implementation. All we have to do is alter our `Polygon` class so that it can be constructed with multiple points.

Let's give it an initializer that accepts a list of `Point` objects:

```
from collections.abc import Iterable


class Polygon_2:
    def __init__(self, vertices: Iterable[Point] | None = None) -> None:
        self.vertices = list(vertices) if vertices else []

    def perimeter(self) -> float:
        pairs = zip(self.vertices, self.vertices[1:] + self.vertices[:1])
        return sum(p1.distance(p2) for p1, p2 in pairs)
```

This seems to improve things considerably. We can now use this class like the original function definitions:

```
>>> square = Polygon_2(
... [Point(1,1), Point(1,2), Point(2,2), Point(2,1)]
... )
>>> square.perimeter()
4.0
```

It's handy to have the details of the individual method definitions. We've built an API that's close to the original, succinct set of definitions. We've added enough formality to be confident the code is likely to work before we even start putting test cases together.

Let's take one more step. Let's allow it to accept tuples too, and we can construct the `Point` objects ourselves, if needed:

```
type Pair = tuple[float, float]
type Point_or_Tuple = Point | Pair


class Polygon_3:
    def __init__(
            self,
            vertices: Iterable[Point_or_Tuple] | None = None
    ) -> None:
        self.vertices: list[Point] = []
        for pt_tup in vertices or []:
            self.vertices.append(
                self.make_point(pt_tup)
```

```
            )

    @staticmethod
    def make_point(item: Point_or_Tuple) -> Point:
        match item:
            case Point() as pt:
                return pt
            case (float() | int(), float() | int()) as tup:
                return Point(*tup)
            case _:
                raise TypeError(
                    f"unexpected {type(item)}: {item!r}"
                )
```

This initializer goes through the list of items (either `Point` or `tuple`) and ensures that any pairs of float values are converted to `Point` instances.

The `match` statement uses structural pattern matching to identify the structure of the types presented. When the argument value is a `Point` object, nothing needs to be done, it's the desired type. When the argument value is a `tuple` object, then the internals are examined to see whether it contains two objects; each of these must be either a `float` value or an `int` value. We could try to write this using the built-in `isinstance()` function, but it would be a painfully long expression.

If you are experimenting with the preceding code, you should also define these variant class designs by creating subclasses of `Polygon` and overriding the `__init__()` method. Extending a class with dramatically different method signatures can raise error flags from tools such as **Pyright**.

For an example this small, there's no clear winner among the implementation choices. They all do the same small thing. If we have new functions that accept a `Polygon` argument, such as `area(polygon)` or `point_in_polygon(polygon, x, y)`, the benefits of the object-oriented code become increasingly obvious. Likewise, if we add other attributes to the `Polygon` class definition, such as `color` or `texture`, it makes more and more sense to encapsulate that data into a single class.

The more important a set of data is, the more likely it is to have multiple functions specific to that data, and the more useful it is to use a class with attributes and methods that encapsulate the data. We often suggest that two functions with the same data structures are fine, three pushes the envelope. Writing that fourth function that *also* uses the same `list[Point]` data structure is when a class is a good idea.

When making this design decision, it also pays to consider how the class will be used. If we're only trying to calculate the perimeter of one polygon in the context of a much greater problem, using a function will probably be quickest to code and easier to use *one time only*. On the other hand, if our program needs to manipulate numerous polygons in a wide variety of ways (calculating the perimeter, area, and intersection with other polygons, moving or scaling them, and so on), we have almost certainly identified a class of related objects. The class definition becomes more important as the number of instances increases past one.

Additionally, pay attention to the interaction between objects. Look for inheritance relationships; inheritance is difficult to model elegantly without classes. (Functions tend to rely on composition to avoid repetition.) Look for the other types of relationships we discussed in *Chapter 2*, section *Composition and decomposition*.

One size does not fit all. The built-in, generic collections and functions work well for a large number of simple cases. A class definition works well for a large number of more complex cases. The boundary is hazy at best.

The interface to a class needs to offer helpful features. One thing Python lets us do is define a method that behaves as if it were a simple attribute. This can simplify someone's understanding of what a class does.

Adding behaviors to class data with properties

Throughout this book, we've focused on the encapsulation of behavior and data. Often, there is a clear set of largely passive data objects, and generally active methods. In Python, the distinction between data and behavior is uncannily blurry.

It may not help to use platitudes such as *think outside the box*. Rather, the number of possibilities available suggests we need to stop thinking about the box.

Before we get into the details, let's discuss some bad object-oriented design principles. In some languages, we're advised to never access attributes directly. They insist that we write attribute access like this:

```
class Color:
    def __init__(self, rgb_value: int, name: str) -> None:
```

```
        self._rgb_value = rgb_value
        self._name = name

    def set_name(self, name: str) -> None:
        self._name = name

    def get_name(self) -> str:
        return self._name

    def set_rgb_value(self, rgb_value: int) -> None:
        self._rgb_value = rgb_value

    def get_rgb_value(self) -> int:
        return self._rgb_value
```

The instance variables are prefixed with an underscore to suggest that they are non-public (other languages would actually force them to be private). Then, the `get` and `set` methods provide access to each instance variable.

This style is sometimes called "writing C++ with a Python accent." It's more like C++ than it is like Python.

This class would be used in practice as follows:

```
>>> c = Color(0xff0000, "bright red")
>>> c.get_name()
'bright red'
>>> c.set_name("red")
>>> c.get_name()
'red'
```

The preceding example is not nearly as readable as the direct access version that Python favors:

```
class Color_Py:
    def __init__(self, rgb_value: int, name: str) -> None:
        self.rgb_value = rgb_value
        self.name = name
```

Here's how this class works. It's slightly simpler:

```
>>> c = Color_Py(0xff0000, "bright red")
>>> c.name
'bright red'
>>> c.name = "red"
>>> c.name
'red'
```

Why would anyone insist upon purely method-based syntax?

Historically, using getters and setters could make the separate compilation of machine-specific binaries work out in a tidy way. Using a method for every access created a consistent binary interface to an object's attributes. This consideration doesn't apply very well to Python.

One ongoing justification for getters and setters is that, someday, we may want to add extra code when a value is set or retrieved. For example, we could decide to cache a value to avoid complex computations,or we might want to validate that a given value is a suitable input.

For example, we could decide to change the `set_name()` method as follows:

```
class Color_V:
    def __init__(self, rgb_value: int, name: str) -> None:
        self._rgb_value = rgb_value
        if not name:
            raise ValueError(f"Invalid name {name!r}")
        self._name = name

    def set_name(self, name: str) -> None:
        if not name:
            raise ValueError(f"Invalid name {name!r}")
        self._name = name

    # etc.
```

If we had started a project by writing the original code for direct attribute access, and then later changed the class to require using a `set_name()` method, we'd have created a problem: everyone who had written code to access the attribute directly would now have to change their code to use the new method. This leads to unpleasantness in a project team trying to collaborate. Time is spent refactoring working code to keep it working.

The mantra of "make all attributes private, accessible through methods," doesn't make much sense in Python. The Python language lacks any real concept of private members! We can see the source; we often say "We're all adults here."

What can we do? Instead of insisting on methods, we can make the syntax distinction between attribute and method invisible.

Python gives us the `property()` function to make methods that *look* like attributes. We can therefore write our code to use direct member access, and if we ever unexpectedly need to alter the implementation to do some calculation when getting or setting that attribute's value, we can do so without changing the interface.

Let's see how it looks:

```
class Color_VP:
    def __init__(self, rgb_value: int, name: str) -> None:
        self._rgb_value = rgb_value
        if not name:
            raise ValueError(f"Invalid name {name!r}")
        self._name = name

    def _set_name(self, name: str) -> None:
        if not name:
            raise ValueError(f"Invalid name {name!r}")
        self._name = name

    def _get_name(self) -> str:
        return self._name

    name = property(_get_name, _set_name)
```

Compared to the earlier class, we first change the public `name` attribute into a non-public `_name` attribute. Then, we add two more non-public methods to get and set that variable, performing our validation when we set it.

Finally, we have the `property()` construction at the bottom. This is the Python magic. It creates a new attribute on the `Color` class called `name`. It sets this attribute to be a **property** descriptor. Under the hood, a `property` attribute delegates the real work to the two methods we just created. When used in an access context (on the right side of the `=` or `:=` symbol), the first method, `_get_name()`, is evaluated. When used in an update context (on the left side of the `=` or `:=` symbol), the second method, `_set_name()`, is evaluated with the given argument value.

This new version of the `Color` class can be used in exactly the same way as the earlier version, yet it now performs validation when we set the `name` attribute:

```
>>> c = Color_VP(0x0000ff, "bright red")
>>> c.name
'bright red'
>>> c.name = "red"
>>> c.name
'red'
>>> c.name = ""
Traceback (most recent call last):
...
  File "src/colors.py", line 85, in _set_name
    raise ValueError(f"Invalid name {name!r}")
ValueError: Invalid name ''
```

So, if we'd previously written code to access the `name` attribute, and then changed the class so the attribute is now a property, the previous code would still work.

Bear in mind that, even with the `name` property, the previous code is not 100% safe. People can still access the `_name` attribute directly and set it to an empty string if they want to. But if they access a variable we've explicitly marked with an underscore to suggest it is not part of the public interface, they're the ones that have to deal with the consequences, not us. We established a formal contract, and if they elect to break the contract, they own the resulting problems.

Properties in detail

Think of the `property()` function as returning an object that proxies any requests to get or set (or delete) the attribute value through the method names we provide. The `property()` built-in is a constructor for such an object, and that object is set as the public-facing member for the given attribute.

This `property` constructor can actually accept two additional arguments, a `delete` function and a docstring for the property. The `delete` function is evaluated in response to the rarely used `del` statement. A property here can be useful for logging the fact that a value has been deleted, or coping with particularly complicated data structures. The docstring is the string describing what the property does, no different from the docstrings we discussed in *Chapter 2*. If we do not supply this parameter, the docstring will instead be copied from the docstring for the first argument: the `getter` method.

Here is a silly example that states whenever any of the methods are called:

```
class NorwegianBlue:
    def __init__(self, name: str) -> None:
        self._name = name
        self._state: str

    def _get_state(self) -> str:
        print(f"Getting {self._name}'s State")
        return self._state

    def _set_state(self, state: str) -> None:
        print(f"Setting {self._name}'s State to {state!r}")
        self._state = state

    def _del_state(self) -> None:
        print(f"{self._name} is pushing up daisies!")
        del self._state

    silly = property(_get_state, _set_state, _del_state, "This is a silly
    property")
```

Note that the `state` attribute has a type hint, `str`, but no initial value. It can be deleted, and only exists for *part* of the life of a `NorwegianBlue` object. We need to provide a hint to help a tool such as **mypy** understand what the type should be. But we don't assign a default value because that's the job of the `setter` method.

If we actually use an instance of this class, it does indeed print out the correct strings when we exercise the methods:

```
>>> p = NorwegianBlue("Polly")
>>> p.silly = "Pining for the fjords"
Setting Polly's State to 'Pining for the fjords'
>>> p.silly
Getting Polly's State
'Pining for the fjords'
>>> del p.silly
Polly is pushing up daisies!
```

Further, if we look at the help text for the `Silly` class (by issuing `help(Silly)` at the interpreter prompt), it shows us the custom docstring for our `silly` attribute:

```
class NorwegianBlue(builtins.object)
 |  NorwegianBlue(name: str) -> None
 |
 |  Methods defined here:
 |
 |  __init__(self, name: str) -> None
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables
 |
 |  __weakref__
 |      list of weak references to the object
 |
 |  silly
 |      This is a silly property
```

Once again, everything is working as we planned. In practice, properties are generally defined with the first two parameters: the `getter` and `setter` functions. If we want to supply a docstring for a property, we can define it on the `getter` function; the property proxy will copy it into its own docstring. The `delete` function is often left empty because object attributes rarely require explicit deletion.

Decorators—another way to create properties

We can create properties using decorators. This syntax feature makes the definitions easier to read. Decorators are used widely in Python, with a variety of purposes. For the most part, decorators modify the function (or class) definition that they precede. We'll look at the decorator design pattern more broadly in *Chapter 11*.

The `property` function can be used with the decorator syntax to turn a `get` method into a `property` attribute, as follows:

```
class NorwegianBlue_P:
    def __init__(self, name: str) -> None:
        self._name = name
        self._state: str
```

```
    @property
    def silly(self) -> str:
        """This is a silly property"""
        print(f"Getting {self._name}'s State")
        return self._state
```

This applies the `property()` function as a decorator to the method definition that follows, `silly()`. This alternative is precisely equivalent to the previous `silly = property(_get_state)` syntax. The difference is readability: we get to mark the `silly` method as a property at the top of the method, instead of after it is defined, where it can be easily overlooked. It also means we don't have to create a non-public method with underscore prefixes just to define a property.

Going one step further, we can specify a `setter` function for the new property as follows:

```
class NorwegianBlue_P:
    def __init__(self, name: str) -> None:
        self._name = name
        self._state: str

    @property
    def silly(self) -> str:
        """This is a silly property"""
        print(f"Getting {self._name}'s State")
        return self._state

    @silly.setter
    def silly(self, state: str) -> None:
        print(f"Setting {self._name}'s State to {state!r}")
        self._state = state
```

This syntax, `@silly.setter`, looks odd compared with `@property`, although the intent should be clear. First, we decorate the `silly()` method as a getter. Then, we decorate a second method — which must have the same name — by applying the `setter` attribute of the originally decorated `silly()` method! This works because the `property` function returns an object; this object also has its own `setter()` method, which can then be applied as a decorator to other methods. Using the same name for the get and set methods helps to group together the multiple methods that access one common attribute.

We can also specify a `delete` function with `@silly.deleter`. Here's what it looks like:

```
    @silly.deleter
    def silly(self) -> None:
        print(f"{self._name} is pushing up daisies!")
        del self._state
```

We cannot specify a docstring using `property` decorators, so we need to rely on the decorator copying the docstring from the initial getter method. This class operates *exactly* as our earlier version acted, including the help text. You'll see the decorator syntax in widespread use. The function syntax is how it actually works under the hood.

Deciding when to use properties

With the built-in `property` blurring the division between behavior and data, it can be confusing to know when to choose an attribute, a method, or a property. In the `Color_VP` class example we saw earlier, we added argument value validation to setting an attribute. In the `NorwegianBlue` class example, we wrote detailed log entries when attributes were set and deleted.

There are other factors to take into account when deciding to use a property. In Python, data, properties, and methods are all attributes of a class. The fact that a method is also a callable object does not distinguish it from other types of attributes; indeed, we'll see in *Chapter 9*, that it is possible to create objects that can be called like functions. We'll also discover that functions and methods are themselves ordinary objects.

The fact that methods are callable attributes, and properties are also attributes, can help us make this decision. We suggest the following approach:

- Use methods to represent actions: things that can be done to, or performed by, the object. When you call a method, even with only one argument, it should *do* something. Method names are generally verbs.
- Use attributes or properties to represent the state of the object. These are the nouns, adjectives, and prepositions that describe an object.

  There are some further considerations:

  - Default to ordinary (non-property) attributes, initialized in the `__init__()` method. These must be computed eagerly, which is a good starting point for any design.
  - Use properties for attributes when there's a computation involved with setting, getting, or deleting an attribute. Examples include data validation, logging, and access controls.

> We'll look at cache management in a moment. Properties are ideal for creating lazy attributes, where we want to defer the computation when it's costly and rarely needed.

Let's look at a more realistic example. A common need for custom behavior is caching a value that is difficult to calculate or expensive to look up (requiring, for example, a network request or database query). The goal is to store the value locally to avoid repeated calls to the expensive calculation.

We can optimize access with a custom getter on the property. The first time the value is retrieved, we perform the lookup or calculation. Then, we can locally cache the value as a non-public attribute on our object (or in dedicated caching software), and the next time the value is requested, we return the stored data. Here's how we might cache a web page:

```
from urllib.request import urlopen


class WebPage:
    def __init__(self, url: str) -> None:
        self.url = url
        self._content: bytes | None = None

    @property
    def content(self) -> bytes:
        if self._content is None:
            print("Retrieving New Page...")
            with urlopen(self.url) as response:
                self._content = response.read()
        return self._content or b''
```

We'll only read the website content once, when `self._content` has the initial value of `None`. After that, we'll return the value most recently read for the site. We can test this code to see that the page is only retrieved once:

```
import time

webpage = WebPage("http://ccphillips.net/")

now = time.perf_counter()
content1 = webpage.content
first_fetch = time.perf_counter() - now
```

```
now = time.perf_counter()
content2 = webpage.content
second_fetch = time.perf_counter() - now

assert content2 == content1, "Problem: Pages were different"
print(f"Initial Request     {first_fetch:.6f}")
print(f"Subsequent Requests {second_fetch:.6f}")
```

The output?

```
% python src/colors.py
Retrieving New Page...
Initial Request     0.506753
Subsequent Requests 0.000004
```

It took about 0.5 seconds to retrieve a page from the `ccphilips.net` web host. The second fetch — from a laptop's RAM — takes 0.004 milliseconds! This is sometimes written as 4μs, 4 microseconds. Since this is the last digit, we can suspect it's subject to rounding, and the time may be even smaller, perhaps as little as 3μs.

Custom getters are also useful for attributes that need to be calculated on the fly, based on other object attributes. For example, we might want to calculate the average for a list of integers:

```
s AverageList(list[int]):
@property
def average(self) -> float:
    return sum(self) / len(self)
```

This small class inherits from `list`, so we get list-like behavior for free. We added a property to the class, and — hey, presto! — our list can have an average as follows:

```
>>> a = AverageList([10, 8, 13, 9, 11, 14, 6, 4, 12, 7, 5])
>>> a.average
9.0
```


```
>>> a = AverageList([10, 8, 13, 9, 11, 14, 6, 4, 12, 7, 5])
>>> a.average
9.0
```

Of course, we could have made this a method instead. The implication of a method is that some action was involved, but there's no state change and nothing happens to any other objects. A property called `average` is more suitable to a derived value, and is both easier to type and easier to read.

We can imagine a number of similar reductions, including minimum, maximum, standard deviation, median, and mode, all being properties of a collection of numbers. This can simplify a more complex analysis by encapsulating these summaries into the collection of data values as properties, computed as needed.

Custom setters are useful for validation, as we've already seen, but they can also be used to proxy a value to another location. For example, we could add a content setter to the `WebPage` class that automatically logs into our web server and uploads a new page whenever the value is set.

Scripts to functions to classes

Back in *Chapter 1*, we showed a long script. This script consumed a number of files in JSON notation, and summarized some details. The script works. From one perspective, if it works, what more is needed?

The answer is two-fold:

- We need reason to believe the script works
- We need to be able to adapt and maintain the script

How do we know software works? The "try it and see" approach can be expensive and frustrating.

When dealing with something safe and relatively inexpensive, such as a new bookmark, we can put it between pages in a book and confirm that it works. We do have to inspect it, though. Using a slice of freshly-cooked bacon as a bookmark might damage our favorite copy of *Moby Dick, or The Whale*. Using a piece of duct tape, similarly, requires some care before trying it.

Note that even something as safe-seeming as a bookmark requires some inspection. Consider something more dangerous. We don't want to try out software that could delete precious files or corrupt the operating system without some careful inspection.

For software, the inspection process involves looking at a number of artifacts:

- Documentation such as a `README` file, or a `docs` folder. For more sophisticated software, we expect to find a website, often something hosted by `https://about.readthedocs.com/?ref=readthedocs.org`, or `https://github.com`.

- Results of type-checking and lint-checking tools.
- The results of test runs.
- The code itself.

One of the value propositions of Python (and open source software in general) is being able to inspect the code and gain confidence that it does what we expect. A well-crafted script file may have some documentation. Often, lint-checking tools can examine the source code for potential programming problems. A script file is notoriously hard to test. Further, a long script can be baffling to read because so many distinct classes of objects can be juxtaposed.

What can we do? It's difficult to leap from a long sequence of statements into class definitions. There are several small steps that can help move toward class definitions:

1. Refactor big script files to make at least one function, to set a baseline of testability. It's very difficult to make meaningful changes without a test suite, and it's very difficult to test a script file.
2. Decompose a long functions into several functions, organized around common data elements. Rename the functions to have a common name prefix to reflect the common data. Provide some notes on the common data and how the functions interact with this data. Test these group of functions as a separate unit.
3. Collect groups of functions with related data into a single class definition.

None of these is particularly difficult. What's essential is starting from working code and evolving toward tested, documented, working code. The documentation and the test cases provide evidence that the code is trustworthy and does what it's supposed to do. The class-oriented structure lets new readers inspect the code and come to an understanding of the elements in isolation. From the isolated bits, the overall emergent behavior of the application will become evident.

This lets us address second topic, under "what more is needed?", specifically, the need to adapt and maintain the script. There are several common kinds of adaptations:

- New formats for the input or output
- New processing features
- New supporting libraries or frameworks

Generally, a script is structured around the inputs being consumed. A small change to the input breaks the script. This kind of problem is avoided by refactoring a script to encapsulate the input

processing from the computations and output formatting.

The changes to the output format are often pervasive throughout a script: the results are built in separate parts of the script. This kind of problem is avoided by refactoring a script to encapsulate the output processing from the computations and input parsing.

Changes to the processing, similarly, might be scattered throughout a script. This, too, needs to be encapsulated.

Change to libraries and frameworks is part of modern open source software. Everyone has a good idea, and the good ideas evolve rapidly. This requires the careful design of interfaces to allow a supporting package to be gracefully excised and replaced when a better idea comes along.

Software that's popular and useful will be adapted and changed.

Since there will be constant refactoring, strive for clear, expressive class definitions. This can help facilitate the changes, the documentation, and the testing.

We don't want to say that object-oriented programming is inevitable. We do want to encourage active decomposition of big scripts to smaller functions.

Recall

Here are some of the key points in this chapter:

- When we have both data and behavior, this is the sweet spot for object-oriented design. We can leverage Python's generic collections and ordinary functions for many things. When it becomes complex enough that we need to be sure that pieces are all defined together, then we need to start using classes.
- When an attribute value is a reference to another object, the Pythonic approach is to allow direct access to the attribute; we don't write elaborate setter and getter functions.
- When an attribute value is computed, we have two choices: we can compute it eagerly or lazily. Setting it during the `__init__()` processing computes it eagerly. A property lets us be lazy and do the computation only if it is needed.

Exercises

We've looked at various ways that objects, data, and methods can interact with each other in an object-oriented Python program. As usual, your first thoughts should be how you can apply these

principles to your own work. Do you have any messy scripts lying around that could be rewritten using an object-oriented manager? Look through some of your old code and look for methods that are not actions. If the name isn't a verb, try rewriting it as a property.

Think about code you've written in any language. Does it break any of the widely-used design principles? Is there any duplicate code? Did you copy and paste code? Did you write two versions of similar pieces of code because you didn't feel like understanding the original code? Go back over some of your recent code now and see whether you can refactor the duplicate code using inheritance or composition. Try to pick a project you're still interested in maintaining, not code so old that you never want to touch again. That will help to keep you interested when you do the improvements!

For an example application, refer to *Chapter 1*. The "*Reading a big script*" section shows a script that does a lot of processing. The script is a lot of statements. Rewriting it into a collection of class definitions may increase the side of the code. But can it simplify all of those decision-making if statements? Can it unwind the deeply nested for statements? Does this help make it more understandable?

In your daily coding, pay attention to the copy and paste commands. Every time you use them in your editor, consider whether it would be a good idea to improve your program's organization so that you only have one version of the code you are about to copy.

Summary

In this chapter, we focused on identifying objects, especially objects that are not immediately apparent: objects that manage and control. Objects should have both data and behaviors, but properties can be used to blur the distinction between the two. The DRY principle is an important indicator of code quality, and inheritance and composition can be applied to reduce code duplication.

In the next chapter, we'll look at Python's methods for defining abstract base classes. This lets us define a class that's a kind of template; it must be extended with subclasses that add narrowly-defined implementation features. This lets us build families of related classes, confident that they will work together properly.

Join our community Discord space

Join our Python Discord workspace to discuss and know more about the book: `https://packt.link/dHrHU`

6

Abstract Base Classes and Operator Overloading

We often need to make a distinction between concrete classes that have a complete set of attributes and methods and an abstract class that is missing some details. This parallels the philosophical idea of abstraction as a way to summarize complexities. We might say that a sailboat and an airplane have a common, abstract relationship of being vehicles, but the details of how they move are distinct.

In Python, we have two approaches to defining similar things:

- **Duck typing**: When two class definitions have the same attributes and methods, then instances of the two classes have the same protocol and can be used interchangeably. We often say, "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."
- **Inheritance**: When two class definitions have common aspects, a subclass can share common features of a superclass. The implementation details of the two classes may vary, but the classes should be interchangeable when we use the common features defined by the superclass.

We can take inheritance one step further. We can have superclass definitions that are abstract: this means they aren't directly usable by themselves, but can be used through inheritance to create concrete classes.

We have to acknowledge a terminology problem with *base class* and *superclass*. This is confusing because they're synonyms. There are two definitions here, and we flip back and forth between them. Sometimes, we'll use the "base class is a foundation" definition, where another class builds on it via inheritance. Other times, we'll use the "concrete class extends a superclass" definition. The "super" class is superior to the concrete class; it's typically drawn above it on a UML class diagram, and it needs to be defined first.

Figure 6.1 shows an abstract base class at the top of the diagram:

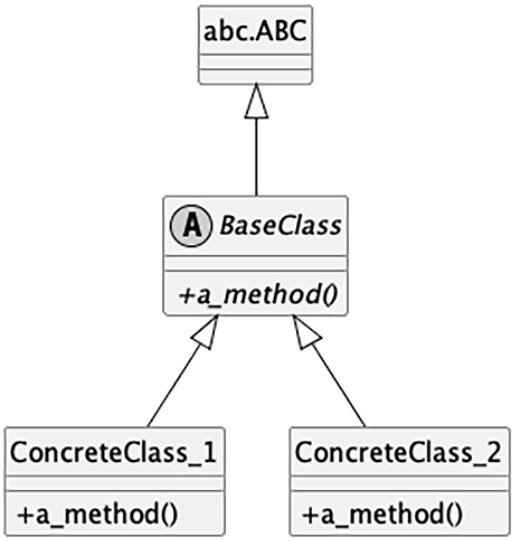


Figure 6.1: Abstract base class

Our base class, named `BaseClass` here, has a special class, `abc.ABC`, as a parent class. This provides some metaclass features that help make sure the concrete classes have replaced the abstractions. In this diagram, we have added a big "A" circle to mark the class as abstract. This bit of decoration is optional, and often unhelpful, so we won't use it in other diagrams. The slanted font is another clue that the class is abstract.

The diagram shows an abstract method, `a_method()`, which doesn't have a defined body. A subclass must provide the missing method. Again, a slanted font is used for the method name to provide a

suggestion that it's abstract. The two concrete subclasses provide distinct implementations of this missing method.

In this chapter, we'll cover the following topics:

- Creating an abstract base class
- ABCs and type hints
- The `collections.abc` module
- Creating your own abstract base class
- Demystifying the magic — looking under the hood at the implementation of an ABC
- Operator overloading
- Extending built-ins
- Metaclasses

We'll start by looking at how we use an abstract class and create a concrete class from it.

Creating an abstract base class

Imagine we are creating a media player that can work with third-party plugins. It is advisable to create an abstract base class (ABC) in this case to document what API the third-party plugins should provide. Clearly stating the contract between the player and plugin is one of the stronger use cases for ABCs.

The general design is to have a common feature, such as `play()`, that applies to a number of classes. We don't want to pick some particular media format to use as a superclass; it seems somehow wrong to claim that some format is foundational, and all others are derived from it.

We'd prefer to define the media player as an *abstraction*. Each unique kind of media file format can provide a *concrete* implementation of the abstraction.

The `abc` module provides the tools to do this. Here's an abstract class that requires a subclass to provide a concrete method and a concrete property to be useful:

```
import abc


class MediaLoader(abc.ABC):
```

```
    ext: str

    @abc.abstractmethod
    def play(self) -> None:
        ...
```

The `abc.ABC` class uses a **metaclass** — a class used to build the concrete class definitions. Python's default metaclass is named `type`. The `type` metaclass doesn't check for abstract methods when we try to create an instance; it allows construction to proceed. The `abc.ABC` class includes an extension to the `type` metaclass to prevent us from creating instances of classes that are not fully defined.

There are two decorators used to describe the placeholders in the abstraction. The example shows an `@abc.abstractmethod` decorator on the `play()` definition. Python uses decorators widely to make modifications to the general nature of the method or function. In this case, the decorator provides additional details used by the metaclass that was included by the `ABC` class. Because we marked a method or property as abstract, any subclass of this class must implement that method or property in order to be a useful, concrete implementation. An attempt to create an instance of an abstract class will raise an exception.

The body of the method is actually `...`; with a three-dot token, the *ellipsis*, which really is valid Python syntax. It reminds everyone that a useful body needs to be written in order to create a working, concrete subclass.

Note the type hint for the `ext` property. This looks like a class-level variable, but doesn't have any value. The intent is for a subclass to provide a class-level variable with a string literal value. This hint helps tools such as **mypy** to check the code for consistent use of types.

One of the consequences of the metaclass and the decorator is the class now has a new special attribute, `__abstractmethods__`. This attribute lists all of the names that need to be filled in to create a concrete class:

```
>>> MediaLoader.__abstractmethods__
frozenset({'play'})
```

Let's see what happens if we don't supply concrete implementations for the abstractions. The first example is a class that's incomplete, and can't be concrete:

```
>>> class Wav(MediaLoader):
...     pass
...
>>> x = Wav()
Traceback (most recent call last):
...
    x = Wav()
TypeError: Can't instantiate abstract class Wav without an implementation
for abstract method 'play'
```

The second example is a class that has the required methods:

```
>>> class Ogg(MediaLoader):
...     ext = '.ogg'
...     def play(self) -> None:
...         pass
...
>>> o = Ogg()
```

The definition of the `Wav` subclass fails to implement the abstract `play()` method. Because this subclass of `MediaLoader` is still abstract, it is not possible to instantiate the class; an exception is raised. The class is still a potentially useful abstract class, but you'd have to subclass it and fill in the abstract placeholder before it can actually do anything.

The `Ogg` subclass supplies both attributes, so it — at least — can instantiate cleanly. It's true, the body of the `play()` method doesn't do very much. What's important is that all of the placeholders were filled, making `Ogg` a concrete subclass of the abstract `MediaLoader` class.

There's a subtle issue with using a class-level variable for the preferred media file extension. Because the `ext` attribute is a variable, it can be updated. Using `o.ext = '.xyz'` is not expressly prohibited. Python doesn't have an easy, obvious way to create read-only attributes. We often rely on documentation to explain the consequences of changing the value of the `ext` attribute. In some cases, we rely on composition to create an immutable wrapper that contains a mutable object. If it were a problem with the community building on this class, we could make it a property, which can be read-only.

Abstract classes have clear advantages when creating a complex application. The use of abstraction like this makes it very easy for tools such as **mypy** to include that a class does (or does not) have the required methods and attributes.

This also mandates a certain amount of fussy importing to be sure that the module has access to the necessary abstract base classes for an application. One of the advantages of duck typing is the ability to avoid complex imports and still create a useful class that can act polymorphically with peer classes. This advantage is often outweighed by the ability of the `abc.ABC` class definition to support type checking via tools such as **mypy**, and to also do a runtime check for completeness of a subclass definition. The `abc.ABC` class also provides far more useful error messages when something is wrong.

One important use case for ABCs is the `collections` module. This module defines the built-in generic collections using a sophisticated set of base classes and mixins.

The ABCs of collections

A really comprehensive use of the ABCs in the Python standard library lives in the `collections` module. The collections we use are extensions of the `Collection` abstract class. `Collection` is an extension of an even more fundamental abstraction, `Container`.

Since the foundation is the `Container` class, let's inspect it in the Python interpreter to see what methods this class requires:

```
>>> from collections.abc import Container
>>> Container.__abstractmethods__
frozenset({'__contains__'})
```

The `Container` class has exactly one abstract method that needs to be implemented, `__contains__()`. You can use `help(Container.__contains__)` to see what the function signature should look like:

```
>>> help(Container.__contains__)
Help on function __contains__ in module collections.abc:

__contains__(self, x)
```

We can see that `__contains__()` needs to take a single argument. Unfortunately, the help text doesn't tell us much about what that argument should be. From the name of the ABC, it seems clear that the argument is the value the user is checking to see whether the container holds.

This `__contains__()` special method implements the Python `in` operator. This method is also implemented by `set`, `list`, `str`, `tuple`, and `dict`. However, we can also define a silly container that

tells us whether a given value is in the set of odd integers:

```
class OddIntegers:
    def __contains__(self, x: int) -> bool:
        return x % 2 != 0
```

We've used the modulo test for oddity. If the remainder of `x` divided by 2 is 0, then `x` was even; otherwise, `x` was odd.

Here's the interesting part: we can instantiate an `OddContainer` object and determine that, even though we did not extend `Container`, the class behaves as a `Container` object:

```
>>> from collections.abc import Container

>>> odd = OddIntegers()
>>> isinstance(odd, Container)
True
>>> issubclass(OddIntegers, Container)
True
```

This example shows why duck typing is an awesome addition to classical polymorphism. We can create is-a relationships without the overhead of writing the code to set up inheritance (or worse, multiple inheritance).

It's important to be sure that methods are defined properly, so a static type-checking tool is important. Use a tool such as **mypy** or **pyright** to provide additional assurance that the definitions are correct.

One cool thing about the `Container` ABC is that any class that implements it gets to use the `in` keyword for free. In fact, `in` is just syntax sugar that delegates to the `__contains__()` method. Any class that has a `__contains__()` method is a `Container` and can therefore be queried by the `in` keyword.

For example:

```
>>> odd = OddIntegers()
>>> 1 in odd
True
>>> 2 in odd
```

```
False
>>> 3 in odd
True
```

The real value here is the ability to create new kinds of collections that are completely compatible with Python's built-in generic collections. We could, for example, create a dictionary that uses a binary tree structure to retain keys instead of a hashed lookup. We'd start with the `Mapping` ABC definitions, but change the algorithms that support methods such as `__getitem__()`, `__setitem__()`, and `__delitem__()`.

Python's duck typing works (in part) because the search for a method is a simple examination of the class definitions in method resolution order. There's little real magic to this.

Abstract base classes and protocols

A concept that's adjacent to abstract classes is the **protocol**. A protocol captures the essence of duck typing: when two classes have the same batch of methods, they both adhere to a common protocol. We see this frequently with classes with similar methods, where there's a common protocol. Consider how the various numeric types, `int`, `float`, `complex`, and others, all support the comparison operators, such as `<`, `<=`, `>`, and `>=`. Comparison is one of many protocols that are part of Python.

The essence of creating ABCs is defined in the `abc` module. We'll look at how this works later. We can make use of protocols without using the `abc` module. In the next section, we want to change direction slightly and make formal use of abstract classes; that means using the definitions in the `collections.abc` module.

The collections.abc module

One prominent use of ABCs is in the `collections.abc` module. This module provides the ABC definitions for Python's built-in collections. This is how `list`, `set`, and `dict` (and a few others) can be built from individual component definitions.

We can use the definitions to build our own unique data structures in ways that overlap with built-in structures. We can also use the definitions when we want to write a type hint for a specific feature of a data structure, without being overly specific about alternative implementations that might also be acceptable.

The definitions in `collections.abc` don't — trivially — include `list`, `set`, or `dict`. Instead, the module provides definitions such as `MutableSequence`,`MutableMapping`, and `MutableSet`, which are the ABCs for which the `list`, `dict`, or `set` classes we use are the concrete implementations.

Let's follow the various aspects of the definition of `Mapping` back to its origins. Python's `dict` class is a concrete implementation of `MutableMapping`. The abstraction comes from the idea of mapping a key to a value. The `MutableMapping` class depends on the `Mapping` definition, an immutable, frozen dictionary, potentially optimized for lookups. Let's follow the relationships among these abstractions.

Figure 6.2 shows the path we want to follow:

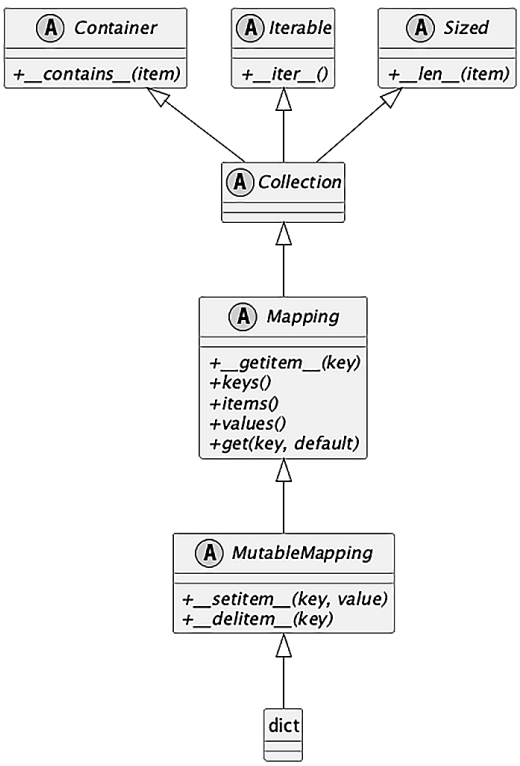


Figure 6.2: The mapping abstractions

Starting in the middle, we can see that the `Mapping` definition depends on the `Collection` class definition. The definition of the `Collection` abstract class, in turn, depends on three other ABCs: `Sized`,`Iterable`, and `Container`. Each of these abstractions demands specific methods.

If we're going to create a lookup-only dictionary — a concrete `Mapping` implementation — we'll need to implement at least the following methods:

- The `Sized` abstraction requires an implementation of the `__len__()` method. This lets an instance of our class respond to the `len()` function with a useful answer.
- The `Iterable` abstraction requires an implementation of the `__iter__()` method. This lets an object work with the `for` statement and the `iter()` function. In *Chapter 10, The Iterator Pattern*, we'll revisit this topic.
- The `Container` abstraction requires an implementation of the `__contains__()` method. This permits the `in` and `not in` operators to work.
- The `Collection` abstraction combines `Sized`, `Iterable`, and `Container` without introducing additional abstract methods.
- The `Mapping` abstraction, based on `Collection`, requires, among other things, `__getitem__()`, `__iter__()`, and `__len__()`. The class has a default definition for `__contains__()`, based on whatever `__iter__()` method we provide. The `Mapping` definition will provide a few other methods, also.

This list of methods comes directly from the abstract relationships in the base classes. By building our new dictionary-like immutable class from these abstractions, we can be sure that our class will collaborate seamlessly with other Python generic classes.

If we look at the documentation at `https://docs.python.org/3/library/collections.abc.html`, we see the page is dominated by a table showing abstract class definitions and the definitions they depend on. There's a lattice of dependencies showing overlap among the class definitions. It's this overlap that allows us to use a `for` statement to iterate through every kind of collection that implements the `Iterable` ABC.

Let's define our own immutable `Mapping` object implementation by extending the abstract classes. The goal is to be able to load our dictionary-like mapping once with keys and values, and then use it to map the keys to their values. Since we aren't going to allow any updates, we can apply a variety of algorithms to make it very fast as well as very compact.

This means we'll start from an abstract base class of `Mapping`. We're not going start from the more general `dict`, because that class offers too many features. The `Mapping` type is both an abstract class as well as a generic type annotation.

Almost all Python code is generic with respect to type. A generic type annotation describes common Python functionality, with one extra bonus. We can provide a missing type to make the generic feature more specific. The general-purpose `dict` class is an implementation of `MutableMapping[Hashable, Any]`. We want to use `Mapping` as the base, to make the structure immutable. And we'd like to narrow the types even further. (We'll look at generic types in *Chapter 7*, also.)

If the key is a type that can be sorted into order, we can do rapid lookups without computing a hash. The value will be an object of any possible type; we won't do anything special here. Changing the type of mapping and the key will lead to a slightly different generic type. The subclass definition starts with the following line of code:

```
class Lookup(Mapping[Comparable, Any]):
```

We've defined the key with the type `Comparable` because we want to be able to compare the keys and sort them into order. Searching through a list in order is often more efficient than searching a list that's not in order. We've left the value to be `Any`.

We'll look at the core of a `Lookup` class definition first. We'll return to the `Comparable` class definition after solidifying the essentials of a new kind of mapping from keys to values.

When we look at ways we can construct a dictionary, we see that a dictionary can be built from two different kinds of data structures. Our new mapping has to have this same flexibility. The two structures are exemplified by the following:

```
>>> x = dict({"a": 42, "b": 7, "c": 6})
>>> y = dict([("a", 42), ("b", 7), ("c", 6)])
>>> x == y
True
```

We can build a mapping from an existing mapping, or we can build a mapping from a sequence of two tuples with keys and values. This means there are two separate definitions for `__init__()`:

- `def __init__ (self, source: Mapping[Comparable, Any]) -> None`
- `def __init__ (self, source: Iterable[tuple[Comparable, Any]]) -> None`

These two definitions have distinct type hints. To make it clear to tools such as **mypy**, we need to provide **overloaded** method definitions. This is done with the `@overload` decorator from the `typing` module. We'll provide two method definitions with the two alternative type hints. After

these, we'll provide the actual implementation method to do the useful work. (The two overloaded type definitions are optional, of course.)

Here's the first part of the `Lookup` class definition. We'll break this into pieces because the `__init__()` method needs to cover these two cases defined by the alternative overloads:

```
from collections.abc import Iterator, Iterable, Mapping
```

```
class Lookup(Mapping[Comparable, Any]):
    @overload
    def __init__(self, source: Iterable[tuple[Comparable, Any]]) -> None:
        ...

    @overload
    def __init__(self, source: "Mapping[Comparable, Any]") -> None:
        ...

    def __init__(
        self,
        source: Any = None,
    ) -> None:
        sorted_pairs: list[tuple[Comparable, Any]]
        match source:
            case Iterable() as an_iter:
                # Assume it's pairs.
                sorted_pairs = sorted(
                    cast(Iterable[tuple[Comparable, Any]], an_iter)
                )
            case Mapping() as a_map:
                sorted_pairs = sorted(a_map.items())
            case _:
                sorted_pairs = []
        self.key_list: list[Comparable] = [p[0] for p in sorted_pairs]
        self.value_list: list[Any] = [p[1] for p in sorted_pairs]
```

The `__init__()` method needs to handle three cases for loading a mapping: a sequence of pairs, another mapping, or nothing. The implementation must sort the keys, and then separate the keys from the values and put them into two parallel lists. A sorted list of keys can be rapidly searched to find a match. The `bisect` module handles this elegantly. A single item from the sorted list of values is returned when we get a key's value from the mapping.

Here are the imports needed:

```
import bisect
from collections.abc import Iterator, Iterable, Mapping
from typing import Protocol, Any, overload, cast
```

Here are the other abstract methods that are defined by the `@abstractmethod` decorator. We provide the following concrete implementations:

```
    def __len__(self) -> int:
        return len(self.key_list)

    def __iter__(self) -> Iterator[Comparable]:
        return iter(self.key_list)

    def __contains__(self, key: object) -> bool:
        index = bisect.bisect_left(
            self.key_list,
            cast(Comparable, key)
        )
        return key == self.key_list[index]

    def __getitem__(self, key: Comparable) -> Any:
        index = bisect.bisect_left(self.key_list, key)
        if key == self.key_list[index]:
            return self.value_list[index]
        raise KeyError(key)
```

The `__len__()`, `__iter__()`, and `__contains__()` methods are required by the `Sized`, `Iterable`, and `Container` abstract classes. The `Collection` abstract class combines the other three without introducing any new abstract methods.

The `__getitem__()` method is required to be a `Mapping`. Without it, a collaborating object can't retrieve an individual value for a given key.

The use of the `bisect` module is one way to find a specific value rapidly in a sorted list of keys. The `bisect.bisect_left()` function finds the spot where a key **should be** in a sorted list. If the key is in the expected position, we can return the value to which it maps. If the key is where it was expected, we can raise the `KeyError` exception.

Note that the `__contains__()` definition has the `object` class as the type hint. This is required because Python's `in` operation needs to support any kind of object, even ones that don't obviously

support the `Comparable` protocol.

Here's how it looks when we use our shiny new `Lookup` class:

```
>>> x = Lookup(
...     [
...         ("z", "Zillah"),
...         ("a", "Amy"),
...         ("c", "Clara"),
...         ("b", "Basil"),
...     ]
... )

>>> x["c"]
'Clara'
```

This collection, generally, behaves a bit like a dictionary. There are a number of dict-like aspects we can't use, though, because we chose an ABC that didn't describe the full set of methods for the `dict` class.

If we try to update the mapping, it looks like this:

```
>>> x["m"] = "Maud"
Traceback (most recent call last):
  ...
  File "<doctest lookup_mapping.__test__.test_not_a_dict[1]>", line 1, in
  <module>
    x["m"] = "Maud"
TypeError: 'Lookup' object does not support item assignment
```

This exception is consistent with the rest of our design. An update to this object means inserting an item at the correct position to maintain a sorted order. Shuffling a large list around gets expensive; if we need to update the lookup collection, we should consider other data structures, such as a red-black tree. But, for the pure search operation using the `bisect` algorithm, this performs nicely.

We also need to look at the definition of the `Comparable` class. This defines the minimum set of features — the protocol — for the keys. It's a way to formalize the comparison rules required to keep the keys for the mapping in order. This helps **mypy** confirm that objects we try to use as keys really can be compared:

```
class Comparable(Protocol):
    def __eq__(self, other: Any) -> bool:
        ...

    def __ne__(self, other: Any) -> bool:
        ...

    def __le__(self, other: Any) -> bool:
        ...

    def __lt__(self, other: Any) -> bool:
        ...

    def __ge__(self, other: Any) -> bool:
        ...

    def __gt__(self, other: Any) -> bool:
        ...
```

There's no implementation for a protocol; it is a very pure specification of a contract a class must adhere to. This protocol definition introduces a new type hint. Existing class definitions such as `str` and `int` implement this protocol, and can be used with our new class. Any other class that also provides the needed methods can be used. This is the beauty of duck typing: any class that offers these methods can be used.

Note that we don't rely on items having a hash code. This is distinct from the built-in `dict` class, which requires the keys to be hashable.

The general approach to using abstract classes is this:

1. Find a class that does most of what you need.
2. Identify the methods in the `collections.abc` definitions that are marked as *abstract*. The documentation often gives a lot of information, but you'll also have to look at the source.
3. Subclass the abstract class, filling in the missing methods.
4. While it can help to make a checklist of the methods, there are tools to help with this. Creating a unit test (we'll cover testing in *Chapter 13*) means you need to create an instance of your new class. If you haven't defined all the abstract methods, this will raise an exception. Using **mypy** will also help with spotting abstract methods that aren't properly defined in the concrete subclass.

This is a powerful way to reuse code when we choose the abstractions well; a person can form a mental model of the class without knowing all of the details. It's also a powerful way to create closely related classes that can easily be examined by tools such as **mypy**. Beyond those two advantages, the formality of marking a method as abstract gives us a runtime assurance that the concrete subclass really does implement all the required methods.

Now that we've seen how to use an ABC, let's look at defining a new abstraction.

Creating your own abstract base class

We have two general paths to creating classes that are similar: we can leverage duck typing or we can define common abstractions. When we leverage duck typing, we can formalize the related types by creating a type hint using a protocol definition to enumerate the common methods, or we can use a `Union` annotation to enumerate the alternative types. When we use a common abstraction, each subclass explicitly names the class it extends.

There are an almost unlimited number of influencing factors that suggest one or the other approach. While duck typing offers the most flexibility, we may sacrifice the ability to use tools such as **mypy**. An ABC definition can be wordy.

We'll tackle a small problem. We want to build a simulation of games that involve polyhedral dice. These are dice that have 4, 6, 8, 12, or 20 sides. The six-sided dice are conventional cubes. Some sets of dice include 10-sided dice, which are cool, but aren't — technically — a *regular* polyhedron; they're two sets of five "kite-shaped" faces.

One question that comes up is how best to simulate rolls of these different-shaped dice. We can define an abstract class that has the general features of a die. A concrete subclass can supply the missing randomization capability. The `random` module has a very flexible generator. Other choices may be helpful for other applications.

The `abc` module has the foundational definitions for abstract classes:

```
import abc
```

```
class Die(abc.ABC):
    def __init__(self) -> None:
        self.face: int
        self.roll()
```

```
    @abc.abstractmethod
    def roll(self) -> None:
        ...

    def __repr__(self) -> str:
        return f"{self.face}"
```

We've defined a class that inherits from the `abc.ABC` class. Any attempt to create an instance of the `Die` class directly will raise a `TypeError` exception. Attempting to create the instance creates a runtime exception. The annotation is also checked by tools such as **mypy**.

We've marked a method, `roll()`, as abstract with the `@abc.abstractmethod` decorator. This isn't a very complex method, but any subclass needs to match this abstract definition. The matching between abstract and concrete definitions is only checked by tools such as **mypy**. If we don't use tools to prevent the problem, things are likely to break at runtime.

Consider this mess of code and the error message it creates:

```
>>> class Bad(Die):
...     def roll(self, a: int, b: int) -> float:
...         return (a+b)/2
>>> x = Bad()
Traceback (most recent call last):
...
    self.roll()
    ~~~~~~~~~^^
TypeError: Bad.roll() missing 2 required positional arguments: 'a' and 'b'
```

```
>>> class Bad(Die):
...     def roll(self, a: int, b: int) -> float:
...         return (a + b) / 2
```

We can see that this raised a `TypeError` exception when we tried to make an instance of the `Bad` class. The problem is caused by the base class, `__init__()`, not providing the `a` and `b` parameters to this strange-looking `roll()` method. This is valid Python code, but it doesn't make sense in this context. The method will also generate errors from annotation-checking tools, providing ample warning that the method definition doesn't match the abstraction.

Here are what two proper extensions to the `Die` class look like:

```
import random
```

```
class D4(Die):
    def roll(self) -> None:
        self.face = random.choice((1, 2, 3, 4))


class D6(Die):
    def roll(self) -> None:
        self.face = random.randint(1, 6)
```

We've provided methods that provide a suitable definition for the abstract placeholder in the `Die` class. They use vastly different approaches to selecting a random value. The four-sided die uses `random.choice()`. The six-sided die — the common cube most people know — uses `random.randint()`.

Let's go a step further and create another abstract class. This one will represent a handful of dice. Again, we have a number of candidate solutions, and we can use an abstract class to defer the final design choices.

The interesting part of this design is the differences in the rules for games with handfuls of dice:

- In some games, the rules require the player to roll all the dice. The rules for a lot of games with two dice require the player to roll both dice.
- In other games, the rules allow players to save dice and re-roll selected dice. In some games, such as Yacht, the players are allowed at most two re-rolls. In other games, such as Zilch, they are allowed to re-roll until they elect to save their score or roll something invalid and lose all their points, scoring zilch (hence the game's name).

These are dramatically different rules that apply to a simple list of `Die` instances. Here's a class that leaves the initial roll-all-the-dice implementation as an abstraction:

```
class Dice(abc.ABC):
    def __init__(self, n: int, die_class: type[Die]) -> None:
        self.dice = [die_class() for _ in range(n)]

    @abc.abstractmethod
    def roll(self) -> None:
```

```
        ...

    @property
    def total(self) -> int:
        return sum(d.face for d in self.dice)
```

The `__init__()` method expects an integer, `n`, and the class to be used to create `Die` instances, named `die_class`. The type hint is `type[Die]`, telling **mypy** to be on the lookout for any subclass of the Die ABC. We don't expect an instance of any of the `Die` subclasses; we expect the class object itself. We'd expect to see an expression such as `SomeDice(6, D6)` to create a list of six instances of the `D6` class.

We've defined the collection of `Die` instances as a list because that seems to have all the features we need. Some games will identify dice by their position when saving some dice and re-rolling the remainder of them, and the integer list indices seem useful for that.

This subclass implements the roll-all-the-dice rule:

```
class SimpleDice(Dice):
    def roll(self) -> None:
        for d in self.dice:
            d.roll()
```

Each time the application evaluates `roll()`, all the dice are updated. It looks like this:

```
>>> sd = SimpleDice(6, D6)
>>> sd.roll()
>>> sd.total
23
```

Here's another subclass that provides a dramatically different set of methods. Some of these fill in the spaces left by abstract methods. Other methods, however, are unique to the subclass:

```
from collections.abc import Iterable
```

```
class YachtDice(Dice):
    def __init__(self) -> None:
        super().__init__(5, D6)
```

```
        self.saved: set[int] = set()

    def saving(self, positions: Iterable[int]) -> "YachtDice":
        if not all(0 <= n < 6 for n in positions):
            raise ValueError("Invalid position")
        self.saved = set(positions)
        return self

    def roll(self) -> None:
        for n, d in enumerate(self.dice):
            if n not in self.saved:
                d.roll()
        self.saved = set()
```

We've created a set of positions for dice to be saved. This is initially empty. We can use the `saving()` method to provide an iterable collection of integers as positions to save. It works like this:

```
>>> sd = YachtDice()
>>> sd.roll()
>>> sd.dice
[2, 2, 2, 6, 1]
>>> sd.saving([0, 1, 2]).roll()
>>> sd.dice
[2, 2, 2, 6, 6]
```


```
>>> sd = YachtDice()
>>> sd.roll()
>>> sd.dice
[2, 2, 2, 6, 1]
>>> sd.saving([0, 1, 2]).roll()
>>> sd.dice
[2, 2, 2, 6, 6]
```


We improved the hand from three of a kind to a full house.

In these cases — the `Die` class and the `Dice` class — it's not clear that the `abc.ABC` base class and the presence of an `@abc.abstractmethod` decoration are dramatically better than providing a concrete base class with a common set of default definitions. In class definitions where the implementations are more dramatically different, having a concrete base class may not be helpful.

In some languages, an abstraction-based definition is required. In Python, because of duck typing, abstraction is optional. In cases where it clarifies the design intent, use it. In cases where it seems fussy and little more than overhead, set it aside.

Because it's used to define the collections, we'll often use the `collection.abc` names in type hints to describe the protocols that objects must follow. In less common cases, we'll leverage the `collections.abc` abstractions to create our own unique collections.

Demystifying the magic

We've used ABCs, and it's clear they're doing a lot of work for us. Let's look inside the class to see some of what's going on:

```
>>> Die.__abstractmethods__
frozenset({'roll'})
>>> Die.roll.__isabstractmethod__
True
```

The abstract method, `roll()`, is tracked in a specially named attribute, `__abstractmethods__`, of the class. This suggests what the `@abc.abstractmethod` decorator does. This decorator sets `__isabstractmethod__` to mark the method. When Python finally builds the class from the various methods and attributes, the list of abstractions is also collected to create a class-level set of methods that must be implemented.

Any subclass that extends the `Die` abstraction will also inherit this `__abstractmethods__` set. When methods are defined inside the subclass, names are removed from the set as Python builds the class from the definitions. We can only create instances of a class where the set of abstract methods in the class is empty.

Central to this is the way classes are created: a class builds objects. This is the essence of most of object-oriented programming. But what is a class?

1. A class is another object with two jobs: it has the special methods used to create and manage instances of the class, and it also acts as a container for the method definitions for objects of the class. We think of building class objects with the `class` statement, which leaves open the question of how the `class` statement builds the `class` object.
2. The `type` class is the internal object that builds our application classes. When we enter the code for a class, the details of construction are actually the responsibility of methods of the

type class. After type has created our application class, our class then creates the application objects that solve our problem.

The type object is called the **metaclass**: the class used to build classes. This means every class object is an instance of type. Most of the time, we're perfectly happy with letting a class statement be handled by the type class so our application code can run.

Because type is itself a class, it can be extended. A class, abc.ABCMeta, extends the type class to check for methods decorated with @abstractmethod. This is how additional features can be added to the class definition process.

We can use the ABCMeta metaclass explicitly when we create a new class, if we want:

```
class DieM(metaclass=abc.ABCMeta):
    def __init__(self) -> None:
        self.face: int
        self.roll()

    @abc.abstractmethod
    def roll(self) -> None:
        ...
```

We've used metaclass as a keyword parameter when defining the components that make up a class. This means the abc.ABCMeta extension to the built-in type class will be used to create the final class object.

Now that we've seen how classes are built, we can consider other things that we can do when creating and extending classes. Python exposes the binding between the syntactic operators, such as the / operator, and the methods of the implementing class. This allows the float and int classes to do different things with the / operator, but it can also be used for quite different purposes. For example, the pathlib.Path class makes use of the / operator to combine a Path and str object to create a new Path instance. We'll look at operator overloading next.

Operator overloading

Python's operators, +, /, -, *, and so on, are implemented by special methods. We can apply these operators more widely than the built-in numeric types. Doing this can be called "overloading" the operators: letting them work with more than numeric types.

Looking back at the *The collections.abc module* section, earlier in this chapter, we offered some

foreshadowing about how Python connects some built-in features with our classes. When we look at the `collections.abc.Collection` class, it is the ABC for all `Sized`, `Iterable`, `Containers`; it requires three methods that enable two built-in functions and one built-in operator:

- The `__len__()` method is used by the built-in `len()` function
- The `__iter__()` method is used by the built-in `iter()` function, which means it's used by the `for` statement
- The `__contains__()` method is used by the built-in `in` operator

It's not wrong to imagine that the built-in `len()` function has this definition:

```
def len(object: Sized) -> int:
    return object.__len__()
```

When we ask for `len(x)`, it's doing the same thing as `x.__len__()`, but it is shorter, easier to read, and easier to remember. Similarly, `iter(y)` is effectively `y.__iter__()`. Also, an expression like `z in S` is evaluated as if it were `S.__contains__(z)`.

And yes, with a few exceptions, all of Python works this way. We write pleasant, easy-to-read expressions that are implemented by special methods. Some notable exceptions are the logic operations: `and`, `or`, `not`, and `if-else`. These don't map directly to special method definitions. The **is** operator, also, doesn't rely on special methods.

Because almost all of Python relies on the special methods, it means we can change use their behavior to add features. One prominent example of this is in the `pathlib` module:

```
>>> from pathlib import Path
>>> home = Path.home()
>>> home / ".cargo" / "bin" / "uv"
PosixPath('/Users/slott/.cargo/bin/uv')
```

Note: Your results will vary, depending on your operating system and your username and whether or not you have **uv** installed.

What doesn't vary is that the `/` operator is used to connect a `Path` object with string objects to create a new `Path` object.

The `/` operator is implemented by the `__truediv__()` and `__rtruediv__()` methods. In order to make operations commutative, Python has two places to look for an implementation. Given an

expression of `A` *op* `B`, where *op* is any of the Python operators, such as `__add__` for `+`, Python does the following checks for special methods to evaluate the operator:

1. If `B` is a proper subclass of `A`, try `B.__r`*op*`__(A)` before any others. This lets the subclass `B` override an operation from superclass `A`. If there is the expected method, and it returns a value that's not the special `NotImplemented` value, this is the result. If the method doesn't exist, or it returns `NotImplemented`, keep searching.

2. Try `A.__`*op*`__(B)`. If this returns a value that's not the special `NotImplemented` value, this is the result. For a `Path` object expression such as `home / "miniconda3"`, this is effectively `home.__truediv__("miniconda3")`. A new `Path` object is built from the old `Path` object and the string. If the method doesn't exist, or it returns `NotImplemented`, keep searching.

3. Try `B.__r`*op*`__(A)`. This might be the `__radd__()` method for the reverse addition implementation. If this method returns a value other than the `NotImplemented` value, this is the result. Note that the operand ordering is reversed. For commutative operations, such as addition and multiplication, this does not matter. For non-commutative operations, such as subtraction and division, the change in ordering needs to be reflected in the implementation.

Let's return to our handful of dice example. We can implement a `+` operator to add a `Die` instance to a collection of `Dice`. We'll start with a base definition of a class that contains a handful of different kinds of dice. This will be different from the previous `Dice` class, which assumed homogeneous dice. We'll start with some basics and then incorporate the `__add__()` special method:

```
class DDice:
    def __init__(self, *die_class: type[Die]) -> None:
        self.classes = die_class
        self.dice = [dc() for dc in self.classes]
        self.adjust: int = 0

    def plus(self, adjust: int = 0) -> "DDice":
        self.adjust = adjust
        return self

    def roll(self) -> None:
        for d in self.dice:
            d.roll()

    @property
    def total(self) -> int:
```

```
        return sum(d.face for d in self.dice) + self.adjust
```

This shouldn't be much of a surprise. It looks a lot like the `Dice` class defined previously. We've added an `adjust` attribute; this is set by the `plus()` method so we can use `DDice(D6, D6, D6).plus(2)`. It fits better with some tabletop role-playing games (TTRPGs).

Also, recall that we provide the *types* of the dice to the `DDice` class, not instances of dice. We use the class object, `D6`. The instances of a given class are created by `DDice` in the `__init__()` method.

Here's the cool part: we can use the plus operator with `DDice` objects, `Die` classes, and integers to define a complex roll of the dice:

```
    def __add__(self, die_class: Any) -> "DDice":
        match die_class:
            case type() if issubclass(die_class, Die):
                new_classes = self.classes + (die_class,)
                new = DDice(*new_classes).plus(self.adjust)
                return new
            case int() as adj:
                new = DDice(*self.classes).plus(self.adjust + adj)
                return new
            case _:
                return NotImplemented

    def __radd__(self, die_class: Any) -> "DDice":
        match die_class:
            case type() if issubclass(die_class, Die):
                new_classes = (die_class,) + self.classes
                new = DDice(*new_classes).plus(self.adjust)
                return new
            case int() as adj:
                new = DDice(*self.classes).plus(self.adjust + adj)
                return new
            case _:
                return NotImplemented
```

These two methods are similar in many ways. Adding more `Die` objects tries to preserve the ordering. Adding a new adjustment increases the total adjustments. We check for three separate kinds of operands:

- If the argument value, `die_class`, is a type, and it's a subclass of the `Die` class, then we're

adding another `Die` object to a `DDice` collection. It's an expression like `DDice(D6) + D6 + D6`. It creates a new `DDice` collection.

- If the argument value is an integer, then we're adding an adjustment to a set of dice. This is something like `DDice(D6, D6, D6) + 2`. This, also, creates a new `DDice` collection with a new adjustment value.
- If the argument value is neither a subclass of `Die` nor an integer, then something else is going on, and this class doesn't have an implementation. This may be some kind of bug, or it might be that the other class involved in the operation can provide an implementation. Returning `NotImplemented` gives the other object a chance at performing the operation.

Because we've provided `__radd__()` as well as `__add__()`, the operator can be used with a `DDice` on the left or right side of the `+`. We can use expressions such as `D6 + DDice(D6) + D6` and `2 + DDice(D6, D6)`.

Python operators are completely generic, and the expected type hint must be `Any`. We can only narrow down the applicable types through runtime checks. The `match-case` statement permits very flexible structural type matching. Tools such as **mypy** are astute in following branching logic to confirm that an integer object was properly used in an integer context.

"But wait," you say. "My favorite game has rules that call for $3d6 + 2$." This is shorthand for rolling three six-sided dice and adding 2 to the result. In many TTRPGs, this kind of abbreviation is used to summarize the dice.

Can we use multiplication to do this? Can we permit `3 * DDice(D6) + 2`?

There's no reason why not. For multiplication, we only need to worry about integers. A dice expression such as `D6 * D6` isn't used in any of the rules, but `3*D6` seems really common. Here's the required special methods:

```
    def __mul__(self, n: Any) -> "DDice":
        match n:
            case int():
                new_classes = self.classes * n
                return DDice(*new_classes).plus(self.adjust)
            case _:
                return NotImplemented

    __rmul__ = __mul__
```

These two methods are actually identical. We can use a simple assignment, `__rmul__ = __mul__`, to create a second reference to the method with a new name. This method follows a similar design pattern to the `__add__()` and `__radd__()` methods. For any given `Die` class, we'll create several instances of the given class. This lets us use `3 * DDice(D6) + 2` as an expression to define a dice-rolling rule. The Python operator precedence rules still apply, so the `3 * DDice(D6)` portion is evaluated first.

Python's use of the various *__op__()* and *__rop__()* methods works out extremely well for applying the various operators to objects that are immutable: strings, numbers, and tuples being the primary examples. Our handful of dice presents a bit of a head-scratcher because the state of the individual dice can change. What's important is that we treat the composition of the hand as immutable. Each operation on a `DDice` object creates a new `DDice` instance.

What about mutable objects? When we write an assignment statement such as `some_list += [some_item]`, we're mutating the value of the `some_list` object. The `+=` statement does the same thing as the more complex expression `some_list.extend([some_item])`. Python supports this with operators with names such as `__iadd__()` and `__imul__()`. These are "in-place" operations, designed to mutate objects instead of create new instances.

For example, consider the following:

```
>>> y = DDice(D6, D6)
>>> y += D6
```

This can be processed in one of two ways:

- We can implement `__iadd__()`. The object can mutate itself in place to add one more dice.
- If we don't implement `__iadd__()`, the statement is evaluated as if it were `y = y.__add__(D6)`. The object `y` creates a new, immutable object, and that's given the old object's variable name. That's how `string_variable += "."` works: under the hood, `string_variable` is not mutated; it's replaced.

If it makes sense for an object to be mutable, we can support in-place mutation of a `DDice` object with this method:

```
    def __iadd__(self, die_class: Any) -> "DDice":
        match die_class:
            case type() if issubclass(die_class, Die):
```

```
                self.classes += (die_class,)
                self.dice = [dc() for dc in self.classes]
                return self
            case int() as adj:
                self.adjust += adj
                return self
            case _:
                return NotImplemented
```

The `__iadd__()` method appends to the internal collection of dice. It follows rules similar to the `__add__()` methods: when a class is provided, an instance is created, and it's added to the `self.dice` list; if an integer is provided, it's added to the `self.adjust` value.

We can now perform incremental changes to a single dice-rolling rule. We can mutate the state of a single `DDice` object using assignment statements. The creation of complex dice looks like this:

```
>>> y = DDice(D6, D6)
>>> y += D6
>>> y += 2
>>> y.roll()
>>> y.dice
[5, 6, 2]
```

This builds the $3d6 + 2$ dice roller in incremental pieces.

The use of the internal special method names allows for seamless integration with other Python features. We can build classes using `collections.abc` that fit with existing collections. We can override the methods implementing the Python operators to create easy-to-use syntax.

We can also leverage the special method names to add features to Python's built-in generic collections. We'll turn to that topic next.

Extending built-ins

Python has two collections of built-ins that we might want to extend. We can broadly classify these into the following:

- Immutable objects, including numbers, strings, bytes, and tuples. These will can have extended operators defined. In the *Operator overloading* section of this chapter, we looked at how we can provide arithmetic operations for objects of a `Dice` class.

- Mutable collections, including sets, lists, and dictionaries. When we look at the definitions in `collections.abc`, these are sized, iterable containers, three distinct aspects that we might want to focus on. In the *The collections.abc module* section of this chapter, we looked at creating an extension to the `Mapping` ABC.

There are other built-in types, but these two groupings are generally applicable to a variety of problems. For example, we could create a new collection: a dictionary that rejects duplicate values.

The built-in dictionary always updates the value associated with a key. This can lead to odd-looking code that works. Take the following example:

```
>>> d = {"a": 42, "a": 3.14}
>>> d
{'a': 3.14}
```

Also look at the following example:

```
>>> {1: "one", True: "true"}
{1: 'true'}
```

These are well-defined behaviors. It may seem odd to provide two keys in the expression but have only one key in the result, but the rules for building dictionaries make these inevitable and correct (even if confusing) results.

We may, however, not like the behavior of silently ignoring a key. It may make our application needlessly complex to worry about the possibility of duplicates. Let's create a new kind of dictionary that won't update items once they've been loaded.

Studying `collections.abc`, we need to extend a mapping, with a changed definition of only the `__setitem__()` method to prevent updating an existing key. Working at the interactive Python prompt, we can try this:

```
>>> from collections.abc import Hashable, Any
>>> class NoDupDict(dict[Hashable, Any]):
...     def __setitem__(self, key, value) -> None:
...         if key in self:
...             raise ValueError(f"duplicate {key!r}")
...         super().__setitem__(key, value)
```

When we put it to use, we see the following:

```
>>> nd = NoDupdict()
>>> nd["a"] = 1
>>> nd["a"] = 2
Traceback (most recent call last):
  ...
  File "<doctest examples.md[10]>", line 1, in <module>
    nd["a"] = 2
  File "<doctest examples.md[7]>", line 4, in __setitem__
    raise ValueError(f"duplicate {key!r}")
ValueError: duplicate 'a'
```

We're not done, but we're off to a good start. This dictionary rejects duplicates under some circumstances. However, it isn't blocking duplicate keys when we try to construct a dictionary from another dictionary. We don't want this to work:

```
>>> NoDupdict({"a": 42, "a": 3.14})
{'a': 3.14}
```

So we've got some work to do. Some expressions properly raise exceptions, whereas other expressions still silently ignore duplicate keys.

The problem is not all methods that set items in the mapping are using the `__setitem__()` special method. There are a number of methods that change the state of a dictionary. To alleviate the problem demonstrated, we'll need to override the `__init__()` method, also.

We'll also need to add type hints to our initial draft. This will let us leverage tools such as **mypy** to confirm that our implementation will work in general. Here's a version with `__init__()` added:

```
from collections.abc import Iterable, Hashable, Mapping
from typing import cast, Any

type DictInit = (
    Iterable[tuple[Hashable, Any]]
    | Mapping[Hashable, Any]
    | None
)
```

```
class NoDupdict(dict[Hashable, Any]):
    def __setitem__(self, key: Hashable, value: Any) -> None:
        if key in self:
            raise ValueError(f"duplicate {key!r}")
        super().__setitem__(key, value)

    def __init__(self, init: DictInit = None, **kwargs: Any) -> None:
        match init:
            case Mapping():
                super().__init__(init, **kwargs)
            case Iterable():
                for k, v in cast(Iterable[tuple[Hashable, Any]], init):
                    self[k] = v
            case None:
                super().__init__(**kwargs)
            case _:
                super().__init__(init, **kwargs)
```

This version of the `NoDupDict` class implements an `__init__()` method that will work with a variety of data types. We enumerated the various types using the `DictInit` type variable. This hint is a union of three types: a sequence of *key-value* pairs, a mapping, or a None object. (For more on type unions, see *Chapter 7*.) In the case of a sequence of key-value pairs, we can use the previously defined `__setitem__()` to raise an exception in the event of duplicate key values.

This covers the initialization use cases, but — still — it doesn't cover every method that can update a mapping. We still have to implement `update()`, `setdefault()`,`__or__()`, and `__ior__()` to extend all the methods that can mutate a dictionary. While this is a pile of work to create, the work is encapsulated in a dictionary subclass that we can use in our application. This subclass is completely compatible with built-in classes; it implements many methods we didn't write, and it has one extra feature we did write.

We've built a more complex dictionary that extends the core features of a Python `dict` class. Our version adds a feature to reject duplicates. We've also touched on the use of `abc.ABC` (and `abc.ABCMeta`) to create ABCs. There are times when we might want to take more direct control of the mechanics of creating a new class. We'll turn next to metaclasses.

Metaclasses

As we noted earlier, creating a new class involves work done by the `type` class. The job of the `type` class is to create an empty class object so the various definitions and attribute assignment statements can then initialize the final, usable class we need for our application.

Figure 6.3 shows how `type` works:

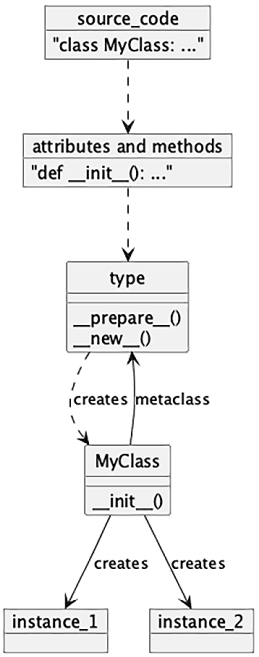


Figure 6.3: How type creates MyClass

The `class` statement is used to locate the metaclass; if no special `metaclass=` is provided, then the `type` class is used. The `type` class will prepare a new, empty dictionary, called a namespace, and then the various statements in the class populate this container with attributes and method definitions. Finally, the "new" step completes the creation of the class; this is generally where we can make our changes.

Figure 6.4 shows how a new class, `SpecialMeta`, builds a new class for us:

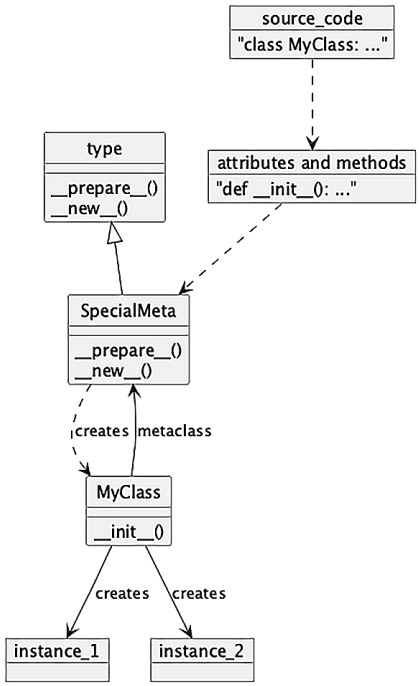


Figure 6.4: Extending the type class

If we use the `metaclass=` option when creating a class, we change the metaclass that's used. In the preceding diagram, `SpecialMeta` is a subclass of the `type` class, and it can do some special processing for our class definitions.

While there are some clever things we can do with this technique, it's important to keep metaclasses in perspective. They change the way class objects are built, with the potential to redefine what it means to be a class. This can drastically shift the foundation of Pythonic object-oriented programming. It can lead to frustration when people reading and maintaining the code can't figure out why something works; it should not be undertaken lightly.

Let's look at a metaclass that builds a few small features into a class definition for us. Let's continue to extend the dice simulation examples from earlier in this chapter. We may have a number of classes of die, each an instance of the abstract base class `Die`. We'd like them all to have an audit log surrounding the `roll()` method supplied by the implementation. We'd like to track each roll separately, perhaps so someone can review them for their statistical validity.

Because we don't want to force the programmers of various kinds of dice to include any extra or new code, we prefer to add logging to the ABC for all `Die` classes. We can also adjust the concrete implementation of the `roll()` method to create logging output.

This is a tall order. It's made a little more challenging because we're working with abstract classes. This requires some care to disentangle abstract class construction from concrete class construction. We don't want to force programmers to change their concrete `Die` class definitions.

To solve this problem using metaclasses, we need to do three things to each concrete `Die`-related class that gets built:

1. Extend the `ABCMeta` metaclass. We need to support the `@abc.abstractmethod` decoration, so we want all the existing metaclass features from the built-in `type` metaclass.
2. Inject a `logger` attribute into each class. It's common to have the logger name match the class name; this is easy to do in a metaclass. We can create the logger as part of the class, prior to any instances of the class being created.
3. Wrap the concrete `roll()` method into a function that uses the programmer's supplied `roll()` method, but also writes a message to the logger. This is similar to the way a method decorator works.

The metaclass definition needs a `__new__()` method to make slight adjustments to the way the final class is built. We don't need to extend the `__prepare__()` method. Our `__new__()` method will use `abc.ABCMeta.__new__()` to build the final class object. It looks like this:

```
import logging
from functools import wraps
from typing import Any


class DieMeta(abc.ABCMeta):
    def __new__(
        cls: type,
        name: str,
        bases: tuple[type, ...],
        namespace: dict[str, Any],
        **kwargs: Any,
    ) -> "DieMeta":
        if "roll" in namespace and not getattr(
            namespace["roll"], "__isabstractmethod__", False
```

```
        ):
            namespace.setdefault("logger", logging.getLogger(name))

            original_method = namespace["roll"]

            @wraps(original_method)
            def logged_roll(self: "DieLog") -> None:
                original_method(self)
                self.logger.info(f"Rolled {self.face}")

            namespace["roll"] = logged_roll
        new_object = cast(
            "DieMeta", abc.ABCMeta.__new__(cls, name, bases, namespace)
        )
        return new_object
```

The `__new__()` method is given a bewildering pile of argument values:

- The `metaclass` parameter is a reference to the metaclass doing the work. Python doesn't generally create and use instances of metaclasses. Instead, the metaclass itself is passed as a parameter to each method. It's a bit like the `self` value provided to an object, but it's the class, not an instance of a class.
- The `name` parameter is the name of the target class, taken from the original `class` statement.
- The `bases` parameter is the list of base classes. These are the mixins, sorted into method resolution order. In this example, it will be the superclass we'll define that uses this metaclass, `DieLog`, shown shortly.
- The `namespace` parameter is a dictionary that was started by the `__prepare__()` method of the built-in `type` class. The dictionary was updated when the body of the class was executed; `def` statements and assignment statements will create items in this dictionary. When we get to the `__new__()` method, the methods (and variables) of the class are staged here, waiting to build the final class object.
- The `kwargs` parameter will have any keyword arguments provided as part of the class definition. If we used a statement such as `class D6L(DieLog, otherparam="something")` to create a new class, then the `otherparam` would be one of the `kwargs` to `__new__()`.

The `__new__()` method must return the new class definition. Generally, this is the result of using the superclass `__new__()` method to build the class object. In our case, the superclass method is `abc.ABCMeta.__new__()`.

Within this method, the `if` statement checks to see whether the class being built defined the needed `roll()` method. If the method is marked with the `@abc.abstractmethod` decorator, then the method will have an attribute of `__isabstractmethod__` and the value of the attribute will be `True`. For a concrete method — without a decorator — there will be no `__isabstractmethod__` attribute. The condition confirms that there's a `roll()` method and whether that `roll()` method is concrete.

For classes with a concrete `roll()` method, we'll add `"logger"` to the namespace that was built, providing a default value of an appropriately named logger. If a logger is already present, we'll leave it in place.

Next, `namespace["roll"]` picks out the function defined in the concrete class, the `roll` method. We'll define a replacement method, `logged_roll`. To be sure the new `logged_roll()` method looks like the original method, we've used the `@wraps` decorator. This will copy the original method name and docstring onto the new method, making it look like the definition originally present in the class. This revised definition is then put back into the namespace so it can be incorporated into the new class.

Finally, we evaluate `abc.ABCMeta.__new__()` with the metaclass, the class name, the base classes, and the namespace that we modified if there was a concrete implementation of the `roll()` method. The `__new__()` operation finalizes the class, doing all the original Python housekeeping.

It can be awkward to use a metaclass; for this reason, it's common to provide a superclass that uses the metaclass. This means our application can extend the superclass without having to fuss about with an extra `metaclass=` parameter in the class definition:

```
class DieLog(metaclass=DieMeta):
    logger: logging.Logger

    def __init__(self) -> None:
        self.face: int
        self.roll()

    @abc.abstractmethod
    def roll(self) -> None:
        ...
```

This superclass, `DieLog`, will built by the `DieMeta` metaclass. Any subclass of this class will also be built by the `DieMeta` metaclass.

Now, our application can create subclasses of `DieLog`, without having to worry about the details of the metaclass: we don't have to remember to include `metaclass=` in the definition. Our final application classes are quite streamlined:

```
class D6L(DieLog):
    def roll(self) -> None:
        """Some documentation on D6L"""
        self.face = random.randrange(1, 7)
```

We've created a dice roller here that logs each roll in a logger named after the class. Here's how it looks when using it:

```
test_d6l = """
>>> random.seed(42)
>>> d = D6L()
```

```
>>> d2 = D6L()
>>> d2.face
6
```

When we configure logging, we'll see the roll details written to the log. An easy way to enable logging is to use this line of code: `logging.basicConfig(level=logging.INFO)`. There are a lot of details in the `logging` module. See the Python standard library documentation for how to configure it.

The details of the logging aspect of this `D6L` class are completely divorced from the application-specific processing of this class. We can change the metaclass to change details of the logging, knowing that all of the relevant application classes will be changed when the metaclass changes.

Since a metaclass changes how a class is built, there are no boundaries on the kinds of things a metaclass can do. The common advice is to keep the metaclass features very small because they're obscure. For example — as written — the `logged_roll()` method of the metaclass will discard any return value from the concrete `roll()` method in a subclass. This may be surprising.

Recall

Here are some of the key points in this chapter:

- Using ABC definitions is a way to create class definitions with placeholders. This is a handy technique, and can be somewhat clearer than using `raise NotImplementedError` in unimplemented methods.
- ABCs and type hints provide ways to create class definitions. An ABC is a type hint that can help to clarify the essential features we need from an object. It's common, for example, to use `Iterable[X]` to emphasize that we need one aspect of a class implementation.
- The `collections.abc` module defines ABCs for Python's built-in collections. When we want to make our own unique collect class that can integrate seamlessly with Python, we need to start with the definitions from this module.
- Creating your own ABC leverages the `abc` module. The `abc.ABC` class definition is often a perfect starting point for creating an ABC.
- The bulk of the work is done by the `type` class. It's helpful to review this class to understand how classes are created by the methods of `type`.
- Python operators are implemented by special methods in classes. We can — in a way — "overload" an operator by defining appropriate special methods so that the operator works with objects of our unique class.
- Extending built-ins is done via a subclass that modifies the behavior of a built-in type. We'll often use `super()` to leverage the built-in behavior.
- We can implement our own metaclasses to change — in a fundamental way — how Python class objects are built.

Exercises

We've looked at the concept of defining abstract classes to define some — but not all — common features of two objects. Take a quick look around to see how you can apply these principles to your own work. A script can often be restated as a class, with each major step of the work a separate method. Do you have similar-looking scripts that — perhaps — share a common abstract definition? Another place to find things that are partially related is in the classes that describe data files. A spreadsheet file often has small variations in layout; this suggests they have a common abstract relationship, but a method needs to be part of an extension to handle the variations in the layouts.

When we think about the `DDice` class, there's yet another enhancement that would be nice. Right now, the operators are all defined for `DDice` instances only. In order to create a hand of dice, we need to — somewhere — use a `DDice` constructor. This leads to `3*DDice(D6)+2`, which seems to be needlessly wordy. It would be nicer to be able to write `3*d6+1`. This implies some changes to the design:

1. Since we can't (easily) apply operators to classes, we have to work with instances of classes. If we use `d6 = D6()` to create a `Die` instance, then `d6` becomes a viable operand. can be an operand.
2. The `Die` class needs a `__mul__()` method and an `__rmul__()` method. When we multiply a `Die` instance by an integer, this will create a `DDice` instance populated with the die's type, `DDice(type(self))`. This is because `DDice` expects a type and it creates its own instances from the type.

This creates a circular relationship between `Die` and `DDice`. It doesn't present any real problems because both definitions are in the same module. We can use strings in the type hints, so having a `Die` method use a type hint of `-> "DDice"` works out nicely. Tools such as **mypy** can use strings for forward references to types that haven't been defined yet.

Now, look back over some of the examples we looked at in previous chapters. Can we leverage an abstract class definition to perhaps simplify the various ways in which `Sample` instances need to behave?

For an example application, refer to *Chapter 1*. The "*Reading a big script*" section shows a script that does a lot of processing. Is there a place for an ABC that defines the common features of a test result? One subclass might be a success and the other subclass might be a failure. Does this abstraction help the design?

Look at the `DieMeta` example. As written, the `logged_roll()` method of the metaclass will discard any return value from the concrete `roll()` method in a subclass. This may not be appropriate in all cases. What kind of rewrite is required to make the metaclass method wrapper return a value from the wrapped method? Does this change the `DieLog` superclass definition?

Can we use the superclass to provide a logger? (It seems like the answer should be a resounding "yes.") More importantly, can we use a decorator to provide logging for a concrete `roll()` method? Write this decorator. Then consider whether or not we can trust developers to include this decorator. Should we trust other developers to use the framework correctly? While we can imagine developers forgetting to include the decorator, we can also imagine unit tests to confirm that log entries are

written. Which seems better: a visible decorator with a unit test or a metaclass that tweaks code invisibly?

Summary

In this chapter, we focused on identifying objects, especially objects that are not immediately apparent. Objects should have both data and behaviors, but properties can be used to blur the distinction between the two. The DRY principle is an important indicator of code quality, and inheritance and composition can be applied to reduce code duplication.

In the next two chapters, we'll cover several of the built-in Python data structures and objects, focusing on their object-oriented properties and how they can be extended or adapted.

7

Python Type Hints

Throughout this book, almost all the examples have included type hints. Python added syntax for "annotations" back in Python 3.0. (See *PEP 3107*: `https://peps.python.org/pep-3107/`.) This syntax was applied to the problem of type hints starting with Python 3.6. (See *PEP 526*: `https://peps.python.org/pep-0526/`.) Since then, the idea of a formalized type system has evolved and grown.

Currently, the details are in `https://typing.readthedocs.io/en/latest/spec/`. This document describes the type system in detail.

The type hints are optional, and some developers feel they're a burden because it's slightly more code that they have to think about.

The type hints provide important clarification, and many developers feel they're essential for describing the designer's intention. Further, the type hints can be checked by tools, giving us helpful feedback when we've done something that may not work. Some tools make extensive use of type hints. The `dataclases` module, for example, builds a class for us, using the type hints in the template class definition we provide.

In this chapter, we'll look a little more deeply at type hints. We'll cover the following topics:

- Type hints and how they're used in object-oriented programming

- Static type checking, and lint-checking tools
- Runtime value checking using packages such as **Pydantic**

We'll start with an overview of some basics of hints as they related to object-oriented programming in Python.

Type hints and object-oriented programming

Almost all the examples so far have included type hints. We've used them for argument values in class methods. We've used them for return types of methods, too.

Consider a class model like the one in *Figure 7.1*:

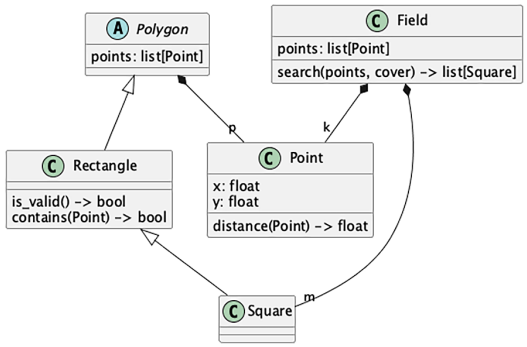


Figure 7.1: A typical class design

We've defined an abstract class, `Polygon`, and several concrete classes, `Point`, `Rectangle`, `Square`, and `Field`. We've referred to two built-in classes, `list` and `float`.

It's possible, using UML, to carefully annotate the diagram with enough graphic details to make the attribute definitions redundant. Doing this can clutter the diagram with boxes and text, making it difficult to interpret.

It seems much more clear to capture the essential relationships with Python type hints than visual cues. Text such as `points: list[Point]` seems clear enough to help understand what a `Polygon` is. The step from sketching on a whiteboard to the initial outline of the code isn't as big when we've used type annotations to capture the essential ideas.

There are two essentials to starting with a UML sketch and moving toward Python:

- Annotate the attributes as well as method parameters and return values with type names. For parameters, providing both the parameter name and type tends toward clutter, so focus on the type. For an example, look at the `Rectangle` class. The definition of the `contains()` method isn't valid Python, but it suggests what the Python will look like: `def contains(self, point: Point) -> bool:`.
- Leverage the generic built-in types, such as `list`, `dict`, `set`, and `tuple`. We'll look at generics later, in section *Generic types*.

Type hint tools such as **Mypy** and **Pyright** automatically focus on functions and methods that have complete hints for all parameters and the return type. Since hints are optional, definitions without hints are quietly ignored. We can use command-line options to report on any definitions with incomplete or missing type hints.

For the most part, the core annotation syntax is very easy to use. Therefore, we strongly suggest using. We'll move on to some more nuanced type hint constructs.

Optionality and unions

The Python language includes an object, `None`, that doesn't have any attributes, and generally raises an exception for most operations. It behaves like `False` in a Boolean context. It's commonly used as a placeholder object.

One of the most common uses is to define optional parameters to a function. Consider this little function:

```
from random import randint

def roll_dice(sides: list[int] | None = None) -> list[int]:
    dice_mix = sides if sides is not None else [6, 6]
    return [randint(1, s) for s in dice_mix]
```

The value of sides is either a list or the `None` object. The syntax of `type | type` is a kind of union between two types. We specify that something is optional by using `| None`.

In this example, we've permitted the following two ways of using this `rolldice()` function:

```
>>> roll_dice([6, 6, 6])
[6, 1, 1]
>>> roll_dice()
```

```
[6, 3]
```

A "union" of types is the fancy term for a collection of alternative types. The syntax Python uses for type hints allows the `None` object as a stand-in for `NoneType`. While it's technically true that `None` is the one-and-only instance of `NoneType`, it seems a bit fussy to insist on a type here.

Consider this more sophisticated function:

```
from random import randint

def roll_ndice(
        dice: int | list[int] | None = None,
        faces: int | None = None
    ) -> list[int]:
    match dice:
        case int() as n:
            dice_mix = n * [faces or 6]
        case None:
            if faces is None:
                dice_mix = [6, 6]
            else:
                dice_mix = [faces]
        case list() as dice_mix:
            assert faces is None, "faces must be None when dice is a list"
        case _:
            raise TypeError(f"can't parse {dice=!r}")
    return [randint(1, s) for s in dice_mix]
```

The value for `dice` can be any of three distinct types. The `match` statement determines what to do with each of the variant types.

At some point, a union can get out of hand. After three or four types, we might want to rethink our design and create a proper class hierarchy, with distinct subclasses instead of a union of a pile of types. If we switch to a class hierarchy, leveraging polymorphism, then the processing details become methods of the classes instead of cases in a `match-case` statement.

Between subclasses and unions, we have all the type complexity we can make use of. A union is sometimes called "one of": the object will have one of the types named. A subclass hierarchy can be called "all of": the object contains attributes defined by all of the parent types. Next, we'll take a second look at the complications of methods that have — perhaps — too many alternatives for

parameters.

Overloaded methods

All of Python code is generic with respect to type. If we don't use type hints, it's the same as using the `Any` type. Consequently, we can design Python functions that accept, well, anything. This means the body of the function needs to disentangle the parameters to determine what to do. The `match` statement was designed for kind of disentangling of complex parameters.

In the example shown in the previous section, there is a constraint that isn't clearly defined in the type hints. Consequently, type-checking tools, such as **Mypy** and **Pyright**, can't be absolutely sure we're using the `roll_ndice()` function properly.

Here's the function's signature — the mix of parameters and return types:

```
def roll_ndice2(
        dice: int | list[int] | None = None,
        faces: int | None = None
    ) -> list[int]:
```

This function has two distinct rules for how it can be used:

- Provide a list of dice sizes, for example, `roll_ndice([10, 10])`. When providing a list, a value for the `faces` parameter must **not** be applied.
- Provide the number of dice and, optionally, the number of faces, for example, `roll_ndice(2, 10)`.

The first form for the parameters (only providing a list) isn't specified in the type hints, and can't be checked in advance.

We have other approaches available, most notably, defining the function as having overloaded type signatures. This uses the `@overload` decorator from the `typing` module. To use this, we write out the two competing signatures, and a final implementation.

In our case, we have two overloaded signatures, which we could define like this:

```
from typing import overload

@overload
def roll_ndice2(
```

```
    dice: list[int] | None
) -> list[int]:
    ...

@overload
def roll_ndice2(
    dice: int | None = None,
    faces: int | None = None
) -> list[int]:
    ...

def roll_ndice2(
        dice: int | list[int] | None = None,
        faces: int | None = None
    ) -> list[int]:
    # The body goes here...
```

There can be any number of alternative definitions. But, pragmatically, if you have too many, your function might be too overwhelmingly complicated. These overload definitions are used by type-hint checkers to make sure the function is used in one of the acceptable patterns.

The overloaded signatures need to be syntactically complete. Instead of an actual body, the Python ellipsis is used: `...`. The `pass` statement is also a possibility, but the ellipsis has become a popular way to mark code that will never be executed.

After the overloads comes the implementation method. Frequently, this will use `match-case` statements to disentangle the argument values. This is the same as the previous `roll_ndice` example. It's important to note that the implementation doesn't change; the overloaded definitions are to help people (and tools) understand the alternatives for using the method.

Overloaded definitions often raise questions about the design. They suggest that, perhaps, too much is going on in one function. In our example, there are two separate things going on:

- Figuring out the mix of dice to roll
- Actually rolling the dice

The two overloaded function definitions suggest there are (at least) two ways to construct the mix of dice. Very likely, there will be more good ideas. Perhaps this can be restated as an abstract base class that defines the common dice-rolling procedure. We could then write two subclasses to implement the two ways to specify the mix of dice.

We've seen a number of type hint features. Another important part of Python are the generic types and how we can provide more useful type hints based on these built-in types.

Generic types

Python comes with a large number of built-in types. Some of these are containers. A Python container — such as a list — is generic with respect to the types it can contain.

Pragmatically, we'll often write software that uses a list of integers. The generic list type accepts a parameter, allowing us to use `list[int]` to state that our software will use only integers in some specific list.

Similarly, we might have a dictionary that maps strings to other strings. The `dict[str, str]` type hint captures this mapping, telling people reading the code (and tools checking the code) the intended use of the dictionary.

This can be composed, of course. We might have a set of two-tuples; the tuples are composed of a pair of float values. This can be described by `set[tuple[float, float]]`. Tools can examine the code to be sure the structure is used appropriately.

Many built-in types are parameterized. Here's a breakdown:

- As noted previously, the generic collections: `set`, `list`, `dict`.
- The collection classes defined in the `collections` module are all generic, and accept parameters. These include `defaultdict`, `ChainMap`, `Counter`, and `deque`.
- The abstract base classes for collections, defined in the `collections` module are also generic.
- The `typing.NamedTuple` definition lets us define new kinds of immutable tuples and provide useful names for the members. This will be covered in *Chapter 8*, and *Chapter 9*.
- A `typing.TypedDict` definition lets us define new kinds of dictionaries with specific keys and specific types associated with the keys.

We'll use the generic collections heavily throughout the book. Python offers the generics to make it easy to use a sophisticated data structure. Using the type hints can help make the application of the generic data structure more clear.

The final topic we'd like to look at is the concept of protocols and how it helps with Python's duck typing.

Protocols and duck typing

Python has a number of built-in protocols. A protocol is used to define a narrow aspect of an object.

Consider, for example, objects that are containers, and support iteration. We describe these using the protocol called `Iterable`. This protocol demands that an implementing class provide the `__iter__()` method.

What's important about a protocol is that we don't include the protocols in a class definition. We include the required method, for example, the `__iter__()` method. But we don't include an explicit marker that the class implements the protocol.

What we do in Python is write methods or functions that require objects that implement some given protocol. We might, for example, change the dice examples shown previously to require `Iterable[int]` instead of `list[int]`.

The relevant statement is this:

```
    return [randint(1, s) for s in dice_mix]
```

The value of `dice_mix` can be any object that supports the `Iterable` protocol. This includes lists, sets, dictionaries, tuples, strings, files, and any customized class with an `__iter__()` method. This includes generator functions, also, since they're iterable.

The function's definition undergoes only the smallest change. Replace `list[int]` with `iterable[int]` in the parameter definitions and the `case` clause. The return type is still `list[int]`, since the expression is a list comprehension.

Using `list[int]` for the parameter type is an artificial constraint: any iterable will work. Using the `Iterable[int]` protocol specifies the barest minimum requirement for a source of integers that allow this expression in the `return` statement to work properly. The definition lets tools warn us of potential mistakes without placing needless constraints on the data types in use.

We can, of course, define our own protocols. They look a lot like class definitions, except they extend the `Protocol` base class.

Python's duck typing approach means any class that has the right methods or attributes can be used; the class hierarchy isn't important, the methods and attributes present at runtime are what's relevant. This mix of methods and attributes — outside the class inheritance hierarchy — can be characterized with a protocol definition. When we're using duck typing, we're relying on a protocol. The type hints provide a formal path so other people reading the code can understand the expected

protocol.

Type hints aren't used by Python at runtime. They're used by tools that confirm that the code and the annotations both agree. Next, we'll look at the tools that we use to check our code for bits of lint that might catch fire as well as disagreements between annotations and code.

Static checking and linting

The **Pyright** tool is commonly used to check the hints for consistency. Another option is the **Mypy** tool. These tools are not built into Python, and require a separate download and install.

Installing tools

We talked about virtual environments and the installation of tools in *Chapter 2*, section *Third party libraries*.

When using the built-in virtual environment manager, `venv`, we'll often install tools with two steps:

1. Activate the environment if it's not already active:

   ```
   % source path/to/venv/bin/activate
   ```

   Often, we'll have put the environments in a handy common directory under our home directory, with a short name, `~/venvs`.

   (Failing to activate the virtual environment for the project you're working on leads to the head-scratching puzzle: "I just installed it, why can't I use it?" The solution is almost always related to environment activation.)

2. Install the tools:

   ```
   % python -m pip install ruff mypy pyright
   ```

   We've listed three tools, you may not really want **all** three. It seemed easier to have one example instead of three.

When using tools such as **poetry** or **uv**, then environment activation isn't as important as making sure the current working directory is the project's top-level directory. The top-level directory has the `pyproject.toml` file. The **uv** command to add a development tool is this:

```
% uv add --dev ruff mypy pyright
```

(Again, we've listed three tools, but you may not really want **all** three; it packs all the alternatives into one example.)

Once the tools are installed, they can be integrated into the unit test processing for our software. Our recommendation is to do type annotation checking **after** running the unit test suite. If the code doesn't work in the first place, checking the annotations isn't always helpful. Once the code works, it helps to make sure the annotations really do match the working code. Often, the annotations didn't evolve as fast as the code did. But sometimes — and these are very important problems to solve — the code has drifted away from the annotations, and the code needs to be fixed.

Checking type hints

Let's say we had a file, `bad_hints.py`, in a `src` directory, with these two functions and a few lines to call the `main()` function:

```
def odd(n: int) -> bool:
    return n % 2 != 0


def main() -> None:
    print(odd("Hello, world!"))


if __name__ == "__main__":
    main()
```

Let's see what happens when we run the `mypy` command at the OS's terminal prompt:

```
% mypy -strict src/bad_hints.py
```

The **mypy** tool is going to reveal a potential problem:

```
ch_07/src/bad_hints.py:15: error: Argument 1 to "odd" has incompatible type
"str"; expected "int"  [arg-type]
```

The code inside the `main()` function will try to evaluate the `odd()` function using a `str` value. This doesn't match the type hint for `odd()` and indicates a potential error. When actually run the code, we'll see that **Mypy** was right all along; the code can't work.

We can also use **Pyright** for this kind of annotation checking.

Comparing tools

One basis for comparison is the pace with which tools are changed. One tiny comparison between the two tools is an observation that the **Pyright** tool tends to handle the `type` statement somewhat more cleanly than the **Mypy** tool.

See *Mypy issue 15238* (`https://github.com/python/mypy/issues/15238`), which was closed while this edition was being prepared. Also see *Mypy 1.12 Released* (`https://mypy-lang.blogspot.com/2024/10/mypy-112-released.html`), which adds support for *PEP 695* (`https://peps.python.org/pep-0695/`).

Lint checking

Checking for lint — bits of fuzz that a dryer accumulates — is called *linting*. The bits of fuzz, when they've accumulated near the heater in a dryer, can lead to a fire. Most of us check the lint filter each time we use the dryer. (Others don't; we sigh and roll our eyes at them.)

Our code can have places where we've done something less than ideal. Too much of this lint and the code may not work because of some nuanced interactions among our poor programming practices. The worst cases are situations where the problem seems to vanish when we try to debug it. For example, enabling logging changes the internal processing in some nuanced way that either prevents or solves the problem. These are often called "heisenbugs", following from Werner Heisenberg's writings about the Observer Effect in quantum physics: the act of observation perturbs the thing being observed. When we stick a thermometer into our chocolate sauce, the mechanism absorbs some heat, changing the temperature of the sauce.

Tools such as **ruff** can be used to check our code for lint. It looks like this:

```
% ruff check src
```

This will examine all the `.py` files in the source directory. It will report on all the potential problems in the code files in that directory.

For the example files in this book, we use a command-line option of `-ignore E402,F811`. This will suppress reports of two potential problems:

- `E402`: Module-level import not at top of file. Since we have multiple examples in a single file,

we often repeat the imports. For real-world Python code, the repetition is a bad idea.

- `F811`: Redefinition of a name. Again, because we repeat imports, there will be names imported more than once. A bad practice outside these text-book examples.

Tools to check type hints and tools to check for lint are critical to creating software that's trustworthy. There numerous choices. We encourage exploration to see which tools provide you with the most useful diagnostic information.

Runtime value checking and the Pydantic package

We've noted that Python's type annotations have no runtime impact. This is true for the language and much of the standard library. There are some cases where one can split a hair to claim that type hints have some impact:

- The `NamedTuple` class definition uses the type hints to define the names of attributes in a tuple
- The `TypedDict` class definition uses the type hints to define the names of attributes in a dictionary
- The `@dataclass` decorator uses the type hints to define the names of attributes in a class derived from the class definition we provide

(We'll look at these alternatives in *Chapter 8*. This is a bit foreshadowing for the next chapter.)

In all of these cases, the annotation information is made available to type-checking tools. Once the desired named tuple, typed dict, or data class has been constructed, there's no additional runtime performance impact from the annotations.

There are, however, third-party packages that can make deep use of type hints for runtime validation of data. We'll take a quick look at one of these, the **Pydantic** package.

First, it must be installed.

When using a tool such as **uv**, this is done via the following command:

```
% uv add pydantic
```

This will update the **pyproject.toml** for the current project. The **uv** tool will install the package

the next time the virtual environment needs to be synchronized.

We can use `pydantic.dataclass` instead of `dataclasses.dataclass` to create a class that can validate input values. There are two general ways to provide the additional validation. One way is to provide some additional methods that are used automatically. The other approach is to include the validation rules as type annotations.

We'll start with an example using additional methods associated with a field. The provided method is used during validation done while creating the object. Generally, there are three steps to validating the data each individual field:

- Any validation processing with a mode of `before` can be done to clean up raw data. This can be tricky because the raw data can be almost anything.
- The built-in parsing and type conversion functions will validate the data.
- Any customized after validation processing can be done. This is the default mode. It's easier to write because the essential type conversion has been done; this validation can focus on ranges of values.

Additionally, the model — as a whole — can also be validated after the individual fields have been checked. This is handy for situations where there are dependencies or consistency rules between multiple fields.

Here's one small example of a **pydantic** dataclass with a field validator:

```
from pydantic import field_validator, Field
from pydantic.dataclasses import dataclass


@dataclass
class Result:
    success: bool
    exit_code: int
    duration: float

    @field_validator('exit_code')
    @classmethod
    def must_be_non_negative(cls, v: int) -> int:
        if v < 0:
            raise ValueError('must be non-negative')
        return v
```

This class definition — like Python's internal dataclasses — comes with numerous features. In addition, it will also validate individual fields. In this example, the `exit_code` field gets an additional check to make sure it's an integer that's not negative.

We can combine the validation rules into the type annotation. This leverages the `Annotated` type hint, defined in the `typing` module. The annotated definition looks like this:

```
from typing import Annotated
from pydantic import Field
from pydantic.dataclasses import dataclass


@dataclass
class Result2:
    success: bool
    exit_code: Annotated[int, Field(ge=0)]
    duration: float
```

In this case, the type hint leads to some runtime behavior. The behavior is a check that's more detailed than a simple validation of the data type. The value must **also** be valid.

Recall

Some key points in this chapter are as follows:

- Type hints help us clarify the relationships between objects.
- We can use hints to show optional relationships, and unions of alternative relationships.
- For methods with complex signatures, we can use hints to provide detailed characterization of the various ways the method can be used.
- We'll make extensive use of generic types to clarify our design intent.
- The core value proposition of duck typing depends on protocols that define the required features of objects.
- Tools help to validate the hints and the code properly align. All the examples in this book were checked with **Mypy** and **Pyright**.
- Use tools to check your code for the fuzzy bits of lint, too. These can build up and lead to problems.

Exercises

All the examples in the book use type hints. It's time to make sure your projects are also using type hints. First, make sure you have a hint-checking tool installed.

Since hints can be added gradually, it makes sense to put them in a little at a time. After putting some hints in one of your exercises, be sure to run the hint-checking tool and see what error messages you get.

For an example application, refer to *Chapter 1*. The "*Reading a big script*" section shows a script that does a lot of processing. Some examples in this chapter show a way to describe some of the JSON documents that are processed by this application.

It can help to extend the examples in this chapter that describe the rest of the JSON document. Once the JSON document structure has been defined fully, the Pydantic class definitions can import the raw JSON text directly to Python objects.

Summary

In this chapter, we've taken a brief tour of the essential features of type hints (also known as annotations). We've looked at the ways we can use hints to clarify the relationships among objects. There are several different kinds of relationships among classes: some are optional, some involve alternatives.

In the next chapter, we'll look at the built-in Python data structures, and how we can apply these classes to our own object-oriented designs.

8

Python Data Structures

Starting in *Chapter 2*, we introduced some foundational concepts in class definitions. *Chapter 3* and *Chapter 6* both built on these foundational definitions to add features to a class.

In many examples, we've leveraged the built-in Python data structures. In this chapter, we'll discuss using these data structures as the basis for class definitions. We'll touch on when they should be used, and how best to make use of them.

The built-in classes are examples of generic types. When writing type hints, parameters can be used to add details; these help clarify how a generic class will be used in a given code context. Additionally, the built-in classes can also be extended by subclasses, where our application can add features to a built-in class.

This chapter will look at several options for defining classes based on built-in types:

- A tuple is a container of data items. It's a little awkward to extend the built-in `tuple` generic to add methods. It's far easier to create a new named tuple with data and methods.
- A `@dataclass` lets us define data elements and methods in a tidy package. This decorator can build a number of standard features into a class, saving us from having to write a lot of double-underscore ("dunder") special methods.
- A dictionary can be used in ways that are similar to a tuple. This can work out nicely. We can

easily extend a dictionary to add methods unique to our application. And, we can formalize the expected list of keys and data types for the dictionary with the `typing.TypedDict` hint.

- We'll also look at extending lists and sets to create collections that have additional features.

Tuples and named tuples

Tuples are objects that can store a specific number of other objects in sequence. A collection of related data items is one important aspect of class design. A tuple is *immutable*, meaning we can't add, remove, or replace objects after a tuple has been created. This may seem like a massive restriction, but the truth is, if you need to modify a tuple, you're using the wrong data type (usually, a `list` type would be more suitable). The primary benefit of tuples' immutability is a tuple of immutable objects (such as strings and numbers and other tuples) has a hash value, allowing us to use them as keys in dictionaries, and members of a set.

A tuple that contains a mutable structure, such as a list, set, or dict, isn't composed of immutable items, and doesn't have a hash value. We'll look closely at this distinction in the next section.

We can use Python's built-in generic `tuple` class to store data. If we also want to associate behavior with a type of tuple, we have two common choices. We can create a subclass of `NamedTuple`, or we can write functions that work with tuples. We'll look at this later, in the section *Named tuples via typing.Named Tuple*. The functional approach is the subject of *Chapter 9*. A bad choice is to try to extend the built-in `tuple` class. This is awkwardly complicated, and not recommended. Since a tuple is immutable, the `__init__()` method isn't used.

Tuples overlap with the idea of coordinates or dimensions. A mathematical (x, y) pair or (r, g, b) color are examples of tuples; the order matters, a lot: the color $(255, 0, 0)$ looks nothing like $(0, 255, 0)$. The primary purpose of a tuple is to aggregate a fixed number of objects into one container.

We create a tuple by separating values with a comma. Usually, tuples are wrapped in parentheses to make them easy to read and to separate them from other parts of an expression, but this is not always mandatory. The following two assignments are identical (they record a stock, the current price, the 52-week high, and the 52-week low, for a rather profitable company):

```
>>> stock = "AAPL", 226.20, 237.49, 164.075
>>> stock2 = ("AAPL", 226.20, 237.49, 164.075)
```

(When the first edition of this book was printed, this stock was trading around US$8 per share; the

stock value has almost doubled with each edition of this book!)

If we're grouping a tuple inside of some other object, such as a function call, list comprehension, or generator, then the parentheses are required. For example, the following function accepts a tuple and a date, and returns a tuple of the date and the middle value between the stock's high and low value:

```
>>> import datetime
>>> def middle(stock, date):
...     symbol, current, high, low = stock
...     return (((high + low) / 2), date)
>>> middle(("AAPL", 226.20, 237.49, 164.075), datetime.date(2024, 11, 22))
(200.7825, datetime.date(2024, 11, 22))
```

In this example, a new two-tuple is created directly inside the function call. The two items are separated by a comma and the entire tuple is cuddled up inside parentheses. When Python displays a tuple, it uses what's called the **canonical** representation; this will always include `()`s, making the `()`s a common practice even when they're not — strictly — required. The `return` statement, in the preceding example, has redundant `()`s around the tuple it creates.

For more information on the syntax, see *section 5.3* (`https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences`) of the Python Tutorial.

We can sometimes wind up with a statement like this:

```
>>> a = 42,
>>> a
(42,)
```

It's sometimes surprising that the `a` variable will be a one-tuple. The trailing comma is what creates an expression list with a single item.

The `middle()` function also illustrates **tuple unpacking**. The first line inside the function unpacks the `stock` parameter into four different variables. The tuple has to be exactly the same length as the number of variables, or it will raise an exception. (As noted previously, if the number of items can vary, the tuple is likely the wrong choice of data structure.)

Unpacking is a very useful feature in Python. A tuple groups related values together to make storing and passing them around simpler; the moment we need to access the pieces, we can unpack

them into separate variables. Of course, sometimes we only need access to one of the variables in the tuple. We can use the same syntax that we use for other sequence types (lists and strings, for example) to access an individual value:

```
>>> stock = "AAPL", 226.20, 237.49, 164.075
>>> high = stock[2]
>>> high
237.49
```

We can even use slice notation to extract larger pieces of tuples, as demonstrated in the following:

```
>>> stock[1:3]
(226.2, 237.49)
```

These examples, while illustrating how flexible tuples can be, also demonstrate a disadvantage: readability. How does someone reading this code know what is in position 2 of a specific tuple? If there's some unpacking code nearby, they can guess, using the name of the variable we assigned it to. But seeing `s[2]`, there's would be no such hint. We don't want to force readers to paw through code to find where the tuple was packed or unpacked to discover what it means.

Accessing tuple members directly is fine in some circumstances, but don't make a habit of it. The index values become what we might call *magic numbers*: numbers that seem to come out of thin air with no apparent meaning within the code. This opacity is the source of many coding errors and leads to hours of frustrated debugging. Try to use tuples only when you know that all the values are going to be useful at once and it's normally going to be unpacked when it is accessed. The ideals are things such as (x, y) coordinate pairs and (r, g, b) colors: the number of items is fixed, the order matters, and the meaning is clear.

One way to provide some useful documentation is to define helper functions. This can help to clarify the way a tuple is used. It's not widely used in Python, but is common in other functional programming languages. Here's an example:

```
>>> def high(quote):
...     symbol, current, high, low = quote
...     return high
>>> high(stock)
237.49
```

We need to keep these helper functions collected together into a single namespace. Doing this causes us to suspect that a class definition can be better than a tuple with a lot of helper functions. The `typing.NamedTuple` class lets us both name the attributes of a tuple, and bundle in useful methods.

Named tuples via typing.NamedTuple

Named tuples are tuples with attitude. They are a great way to create an immutable grouping of data values. When we define a **named tuple**, we're creating a subclass of `typing.NamedTuple`, based on a list of names and data types. We don't need to write an `__init__()` method; it's created for us. For more background, see *Modern Python Cookbook*, *Chapter 7*, for recipes related to named tuples.

Here's an example:

```
from decimal import Decimal
from typing import NamedTuple

class Stock(NamedTuple):
    symbol: str
    current: Decimal
    high: Decimal
    low: Decimal
```

This new class will have a number of methods, including `__init__()`, `__repr__()`, `__hash__()`, and `__eq__()`. These will be based on the generic `tuple` processing with the added benefit of names for the various items. There are more methods, including comparison operations. Here's how we can create a tuple of this class. It looks almost like creating a generic tuple:

```
>>> Stock("AAPL", Decimal('226.20'), Decimal('237.49'), Decimal('164.075'))
Stock(symbol='AAPL', current=Decimal('226.20'), high=Decimal('237.49'),
low=Decimal('164.075'))
```

We can use keyword parameters to make things more clear:

```
>>> s2 = Stock("AAPL", Decimal('226.20'), high=Decimal('237.49'),
low=Decimal('164.075'))
>>> s2
```

```
Stock(symbol='AAPL', current=Decimal('226.20'), high=Decimal('237.49'),
low=Decimal('164.075'))
```

The constructor must have the exact number of arguments to create the tuple. Values can be passed in as positional or keyword arguments.

It's important to recognize that the names are provided at the class level, but we are **not** actually creating class-level attributes. The class-level names we supply are used to build the `__init__()` method; each instance will have the names we provide for the positions within the tuple.

A resulting instance of our `NamedTuple` subclass, `Stock`, can then be unpacked, indexed, sliced, and otherwise treated like a normal tuple. Here's the bonus; we can also access individual attributes by name as if it were an object:

```
>>> s2.high
Decimal('237.49')
>>> s2[2]
Decimal('237.49')
>>> symbol, current, high, low = s2
>>> high
Decimal('237.49')
```

Named tuples are perfect for many use cases. Like strings, tuples and named tuples are immutable, so we cannot modify an attribute once it has been set. For example, the current value of this company's stock has gone down since we started this discussion, but we can't set the new value, as can be seen in the following:

```
>>> s2.current = Decimal('229.87')
Traceback (most recent call last):
  ...
    s2.current = Decimal('229.87')
    ^^^^^^^^^^
AttributeError: can't set attribute
```

The immutability refers only to the tuple itself. This can seem odd, but it's a consequence of the definitions of an immutable tuple. An immutable tuple can contain mutable elements. Consider the following tuple that contains a mutable list:

```
>>> t = ("Relayer", ["Gates of Delirium", "Sound Chaser"])
>>> t[1].append("To Be Over")
>>> t
('Relayer', ['Gates of Delirium', 'Sound Chaser', 'To Be Over'])
```

The object, `t`, is a tuple, which means it's immutable: it contains a string and a list. The mutability of a list inside the `t` object has nothing to do with immutability of the containing object. A list is mutable, irrespective of context. The tuple, `t`, is immutable—nothing can be added or removed. The mutable list is always part of `t`, even if items within the list come and go.

Because the example tuple, `t`, contains a mutable list, it doesn't have a hash value. This shouldn't be too surprising. The `hash()` computation for the tuple must accumulate the hash values from from each item within the tuple. Since the list value of `t[1]` can't produce a hash, the `t` tuple — as a whole — can't produce a hash, either.

Here's what happens when we try:

```
>>> hash(t)
Traceback (most recent call last):
  ...
    hash(t)
TypeError: unhashable type: 'list'
```

We can create methods to compute derived values of the attributes of a named tuple. We can, for example, redefine our `Stock` tuple to include the middle computation as a method (or `@property`):

```
class StockM(NamedTuple):
    symbol: str
    current: Decimal
    high: Decimal
    low: Decimal

    @property
    def middle(self) -> Decimal:
        return (self.high + self.low) / 2
```

We can't change the state, but we can compute values derived from the current state. This lets us couple computations directly to the tuple holding the source data. Here's an object created with this definition of the `Stock` class:

```
>>> s_m = StockM("AAPL", Decimal('226.20'), high=Decimal('237.49'),
low=Decimal('164.075'))
>>> s_m.middle
Decimal('200.7825')
```

The `middle()` method is part of the class definition, not a separate function. The best part? Tools such as **mypy** can look over our shoulder to be sure the type hints all match up properly throughout our application.

The state of a named tuple is fixed when the tuple is created. If we need to be able to change stored data, a `dataclass` may be what we need instead. We'll look at those next.

Dataclasses

Since Python 3.7, dataclasses let us define ordinary objects with a clean syntax for specifying attributes. They look – superficially – very similar to named tuples. This is a pleasant approach that makes it easy to understand how they work. For more details, see *Modern Python Cookbook*, *Chapter 7* for recipes related to dataclasses.

Here's a `dataclass` version of our `Stock` example:

```
from decimal import Decimal
from dataclasses import dataclass


@dataclass
class Stock:
    symbol: str
    current: Decimal
    high: Decimal
    low: Decimal
```

For this case, the definition is nearly identical to the `NamedTuple` definition.

The `@dataclass` decorator transforms the given code into a more complete class. We encountered decorators in *Chapter 6*. We'll dig into them deeply in *Chapter 11*.

As with `NamedTuple`, it's important to recognize that the names are provided at the class level, but are **not** actually creating class-level attributes. The class level names are used to build several methods, including the `__init__()` method; each instance will have the expected attributes. The

decorator transforms what we write into the more complex definition of a class with the expected features.

Because dataclass objects can be stateful, mutable objects, there are a number of extra features available. We'll start with some basics. Here's an example of creating an instance of the `Stock` dataclass:

```
>>> s2 = Stock("AAPL", Decimal('226.20'), high=Decimal('237.49'),
low=Decimal('164.075'))
>>> s2
Stock(symbol='AAPL', current=Decimal('226.20'), high=Decimal('237.49'),
low=Decimal('164.075'))
```

Once instantiated, the `Stock` object can be used like any ordinary class. You can access and update attributes as follows:

```
>>> s2.high
Decimal('237.49')
>>> s2[2]
Decimal('237.49')
>>> symbol, current, high, low = s2
```

As with other objects, we can add attributes beyond those formally declared as part of the dataclass. This isn't always the best idea, but it's supported because this is an ordinary mutable object:

```
Decimal('237.49')

>>> s2.current = Decimal('229.87')
```

Adding attributes like this isn't available for frozen dataclasses, which we'll talk about later in this section. A frozen dataclass is — in a way — a lot like `NamedTuple`.

A dataclass provides a much more useful string representation than we get from the implicit superclass, `object`. By default, dataclasses include an equality comparison, too. This can be turned off in the cases where it doesn't make sense.

Class definitions decorated with `@dataclass` also have many other useful features. For example, you can specify a default value for the attributes of a dataclass. Perhaps the market is currently closed and you don't know what the values for the day are:

```
@dataclass
class StockDefaults:
    name: str
    current: Decimal = Decimal('0.00')
    high: Decimal = Decimal('0.00')
    low: Decimal = Decimal('0.00')
```

You can construct this class with just the stock name; the rest of the values will take on the defaults. But you can still specify values if you prefer, as follows:

```
>>> StockDefaults("GOOG")
StockDefaults(name='GOOG', current=Decimal('0.00'), high=Decimal('0.00'),
low=Decimal('0.00'))
>>> StockDefaults("GOOG", Decimal('166.57'), Decimal('193.31'),
Decimal('129.40'))
StockDefaults(name='GOOG', current=Decimal('166.57'),
high=Decimal('193.31'), low=Decimal('129.40'))
```

We saw earlier that dataclasses support equality comparison by default. If all the attributes compare as equal, then the dataclass objects as a whole also compare as equal. By default, dataclasses do not support other comparisons, such as less than or greater than; this means they can't be sorted. However, you can easily add comparisons if you wish, demonstrated as follows:

```
@dataclass(order=True)
class StockOrdered:
    name: str
    current: Decimal = Decimal('0.00')
    high: Decimal = Decimal('0.00')
    low: Decimal = Decimal('0.00')
```

It's okay to ask "Is that all that's needed?" The answer is yes. The `order=True` parameter to the decorator leads to the creation of all of the comparison special methods. This change gives us the opportunity to sort and compare the instances of this class. It works like this:

```
>>> stock_ordered1 = StockOrdered("GOOG", Decimal('166.57'),
Decimal('193.31'), Decimal('129.40'))
>>> stock_ordered3 = StockOrdered("GOOG")
```

```
>>> stock_ordered3 = StockOrdered("GOOG", Decimal('142.45'),
high=Decimal('151.85'), low=Decimal('84.95'))

>>> stock_ordered1 < stock_ordered2
False
>>> stock_ordered1 > stock_ordered2
True
>>> from pprint import pprint
>>> pprint(sorted([stock_ordered1, stock_ordered2, stock_ordered3]))
[StockOrdered(name='GOOG',
              current=Decimal('0.00'),
              high=Decimal('0.00'),
              low=Decimal('0.00')),
StockOrdered(name='GOOG',
             current=Decimal('142.45'),
             high=Decimal('151.85'),
             low=Decimal('84.95')),
StockOrdered(name='GOOG',
             current=Decimal('166.57'),
             high=Decimal('193.31'),
             low=Decimal('129.40'))]
```

When the dataclass decorator receives the `order=True` argument, it will, by default, compare the values based on each of the attributes in the order they were defined. In this case, it first compares the `name` attribute values of the two objects. If those are the same, it compares the `current` attribute values. If those are also the same, it will move on to `high` and will even include `low` if all the other attributes are equal. The rules follow the definition of a tuple: the order of definition is the order of comparison.

Another interesting feature of dataclasses is `frozen=True`. This creates a class that's similar to `typing.NamedTuple`. There are some differences in what we get as features. We'd need to use `@dataclass(frozen=True, ordered=True)` to create structures such as `NamedTuple`. This leads to a question of "Which is better, a named tuple of a frozen dataclass?" This doesn't have an answer — mostly because *better* is undefined. We haven't explored all of the optional features of dataclasses, such as initialization-only fields and the `__post_init__()` method. Some applications don't need all of these features, and a simple `NamedTuple` may be adequate for a specific application.

There are a few other approaches for creating rich class definitions with less code. Outside the standard library, packages such as `attrs`, `pydantic`, and `marshmallow` provide attribute definition

capabilities that are — in many ways — similar to dataclasses. See `https://jackmckew.dev/dataclasses-vs-attrs-vs-pydantic.html` for a comparison. There's a small example of using **pydantic** in *Chapter 7*.

We've looked at two ways to create unique classes with specific attribute values, named tuples and dataclasses. It's often easier to start with dataclasses and add specialized methods. This can save us a bit of programming because some of the basics, such as initialization, comparison, and string representations, are handled elegantly for us.

It's time to look at Python's built-in generic collections, `dict`, `list`, and `set`. We'll start by exploring dictionaries.

Dictionaries and typed dictionaries

Dictionaries are central to object definition. Each object we create has a dictionary buried inside it to hold the attribute values. We can, of course, be more explicit about using dictionaries.

If you don't know how to create or work with dictionaries, we direct you to the official Python tutorial. See *section 5.5* (`https://docs.python.org/3/tutorial/datastructures.html#dictionaries`). Also, see the *Modern Python Cookbook*, *Chapter 5*, for several recipes that show how dictionaries work.

Dictionaries are extremely versatile. There are two major ways to think about dictionaries:

- We can have dictionaries where all the values are different instances of objects with the same type. We might have a `dict[str, Stock]` dictionary to map a stock name to details of the price history.
- Another very common use case is to each key represent some an attribute of a single object. In this case, the keys are defined by the application's needs and the values often have distinct types. We may, for example, represent a stock described by a dictionary like this: `'name': 'GOOG', 'current': 1245.21, 'range': (1252.64, 1245.18)`. This case clearly overlaps with named tuples, dataclasses, and objects in general. Indeed, there's a special type hint for this kind of dictionary, called `TypedDict`, that looks like a `NamedTuple` type hint. We'll look at this in the *Typed dictionaries* section, later.

In our stock application, we would most often want to look up prices by the stock symbol. We can create a dictionary that uses stock symbols as keys, and tuples (you could also use named tuples or dataclasses as values, of course) of current, high, and low as values, like this:

```
>>> stocks = {
...     "GOOG": Stock("GOOG", Decimal('166.57'), high=Decimal('193.31'),
low=Decimal('129.40')),
...     "MSFT": Stock("MSFT", Decimal('110.41'), high=Decimal('110.45'),
low=Decimal('109.84')),
... }
```

We've mapped the stock symbols to instances of a `Stock` class, defined previously. An interesting design question is the response when a key is missing. We have two choices:

- Raise a `KeyError` exception. Using `stocks["RIMM"]`, for example, will raise an exception.
- Return some kind of default value. This, further, has two additional choices:
  - Add the new default value to the dictionary for future use. When we use `x = stocks.setdefault("RIMM", None)`, we'll either get the existing value associated with the key `"RIMM"` or we'll update the dictionary to insert a `None` object as the value for the key `"RIMM"` and return the `None` object.
  - Leave the dictionary untouched. When we use `x = stocks.get("RIMM", None)`, the dictionary will not be updated.

The default value situation is interesting when we're using a dictionary to collect values.

Let's say we have some `DailyQuote` class definition. This class has a number of attributes with dates and stock prices. It looks like this:

```
import datetime
from decimal import Decimal
from typing import NamedTuple
```

```
class DailyQuote(NamedTuple):
    symbol: str
    date: datetime.date
    price: Decimal
```

We might have a process to partition a big sequence of `DailyQuote` into smaller lists, one for each symbol. The processing looks like this:

```
>>> summary: dict[str, list[DailyQuote]] = {}
>>> for dq in some_source_of_daily_quotes:
...     summary.setdefault(dq.symbol, list())
...     summary[dq.symbol].append(dq)
```

The idea here is that a stock symbol that has not previously been seen needs to be added to the dictionary with an empty list object as the value. After that, the symbol is known, and an object can be appended to the list. At the end of this process, we'd have a dictionary that maps a stock symbol to a sequence of daily quote instances related to the given symbol.

This is used so frequently that there's a separate `defaultdict` class in the `collections` module that implements this for us. Using this class changes the example to the following:

```
>>> from collections import defaultdict

>>> summary: defaultdict[str, list[DailyQuote]] = defaultdict(list)
>>> for dq in some_source_of_daily_quotes:
...     summary[dq.symbol].append(dq)
```

There is an even more specialized variant on a dictionary with a default value. The `Counter` class is — in effect — `defaultdict(int)`. It has a number of additional methods for working with frequency data. Plus it has a very clever `__init__()` method. This class is ideal for gathering frequency information.

For example, we can count the frequency of each symbol in a collection of data with the following:

```
>>> from collections import Counter

>>> frequency = Counter()
>>> for dq in some_source_of_daily_quotes:
...     frequency[dq.symbol] += 1
```

We can use a generator expression to extract the `symbol` attribute from each of the `DailyQuote` objects. That changes the frequency counts to the following pair of statements:

```
>>> symbols = (dq.symbol for dq in some_source_of_daily_quotes)
>>> frequency = Counter(symbols)
```

This example relies on two things that are features of a `Counter`. First, the initialization scans the given iterable sequence of objects and uses `self[k] += 1` for each key value, `k`, in the given iterable. Second, and more important, when a `Counter` has a missing key, the default value that's returned is zero.

The various approaches to handling missing values and default values mean there are a lot different ways to use dictionaries. The base `dict` class, which raises an exception, the `defaultdict` extension, evaluates a function to compute a default, and the `Counter` supplies zero. The core of these options for working with missing values is defined by a special method named `__missing__()`. This method must either raise an exception or return a value. It can also update the dictionary, or — perhaps — write a warning to a log file before supplying a default value.

To extend a dictionary, we can use code such as the following:

```
class StockQuoteSummary(dict[str, list[DailyQuote]]):
    def __missing__(self, symbol: str) -> list[DailyQuote]:
        self[symbol] = list()
        return self[symbol]
    def by_date(self, symbol: str) -> list[DailyQuote]:
        return sorted(self[symbol], key=lambda dq: dq.date)
```

Now, we can accumulate and process a history of `DailyQuote` objects with code such as the following:

```
>>> summary = StockQuoteSummary()
>>> for dq in some_source_of_daily_quotes:
...     summary[dq.symbol].append(dq)

>>> for symbol in summary:
...     print(summary.by_date(symbol))
```

The output shows lists of quotes, properly sorted into ascending order by date.

This shows how we can use dictionary-like features to define our own classes. We can combine data and behavior into a tidy package.

In *Modern Python Cookbook*, *Chapter 8* provides recipes that show how a number of the classes in the `collections` module work.

Previously, we noted that dictionary values can be homogeneous — they all have the same type. We might also want dictionaries to be heterogeneous, with the values having potentially distinct types. We can formalize these more sophisticated dictionaries with a `TypedDict` definition.

Typed dictionaries

We've used tuples, named tuples, and dataclasses to describe a collection of values related to a stock price. We can also use a dictionary for this.

Here are two versions of a dictionary with a variety of values:

```
>>> s_1 = {
...     'symbol': 'GOOG',
...     'current': 1245.21,
...     'range': (1252.64, 1245.18)
... }
```

```
>>> s_2 = dict(
...     symbol='GOOG',
...     current=1245.21,
...     range=(1252.64, 1245.18)
... )
```

If you're familiar with JSON documents, this kind of dictionary with a mixed bag of distinct values should look familiar. (For more information, see `https://www.json.org`.)

Looking back at the way named tuple and dataclass instances are created, this second form, using the `dict()` function, should feel familiar. This is, of course, by design. We want to be able to change data structures without rewriting every bit of syntax in our Python programs.

If we have dictionary keys that are not valid Python names, for example, a string `"current$"` requires using the `{}` form for a dictionary literal. This won't be permitted in the `dict()` form.

We can formalize this stock quote dictionary using the `typing.TypedDict` class. The definition looks a lot like a `typing.NamedTuple` class:

```
from decimal import Decimal
from typing import TypedDict

class Range(TypedDict):
```

```
    low: Decimal
    high: Decimal

class Stock(TypedDict):
    symbol: str
    current: Decimal
    range: Range
```

(When you look back at section *Named tuples via typing.Named tuple*, you'll see how the design has evolved a bit from the first approach with a flat list of fields.)

We can create these objects as follows:

```
>>> s_td = Stock(
...     symbol='GOOG',
...     current=Decimal('166.57'),
...     range=Range(low=Decimal('129.40'), high=Decimal('193.31'))
... )
```

As with most type hints, the types are checked by tools. At runtime, only the keys are checked to make sure the required names are present.

While these look like class definitions, there are some limitations. The most notable is that we can't easily add methods as part of the class definition. If we need to add methods above and beyond what a dictionary offers, it's likely we have an object that might be a dataclass or a class we build ourselves.

What is most interesting about dictionaries in general is the way we can handle optional and required attributes. With a dataclass or a named tuple, we can define an optional attribute as having a value that with a type of `str | None = None`. The attribute will be present, but it may have a value of `None`.

With a typed dictionary, the attribute may not be present at all. This gives us a spectrum of possibilities:

- Attribute is present and has a value of some type
- Attribute is present and has a value of `None`
- Attribute is not present This leads to further design choices.
  - Use `__missing__()` to provide a default or update the object (or both)

  - Raise an exception because the attribute is missing
  - Suggest the collaborating object use the `get()` or `setdefault()` methods to handle the missing attribute

This is a wide variety of alternatives. The question of "missing" data or a "not-applicable" attribute is a profound one. Each application will have unique requirements for dealing with data that doesn't apply, can't be found, or can't be trusted, or is known to be erroneous. For text-book examples, there's an implication that all data is required and is correct. For real-world applications, this simplistic assumption may be invalid.

To help clarify the rules, there are a few additional type hints we can use with typed dictionary declarations.

The first is a metaclass attribute we can provide when we define the class. The default behavior for a typed dictionary is to require all of the named attributes. We can use `(TypedDict, total=False)` as the base class to make the attributes optional.

We can further fine-tune a non-total typed dictionary with these generic type annotations:

- `name: Required[type]`. This attribute must be present or the dictionary cannot be created. An exception will be raised if it's omitted. This only makes sense when `total=False` is used to define the typed dictionary as a whole.
- `name: NotRequired[type]`. This attribute is optional. This is redundant when `total=True` (which is the default behavior).

We might use something like the following to define typed dictionary with optional attributes:

```
from decimal import Decimal
import datetime
from typing import TypedDict, NotRequired

class StockN(TypedDict):
    symbol: str
    name: NotRequired[str]
    current: Decimal
    range: Range
    date: NotRequired[datetime.date]
```

This definition introduces two optional attributes. If a value is provided, it's expected to be of the proper type. We could — either through malice or as a bug — try to create a value of the `StockN`

class with an invalid attribute value:

```
>>> s = StockN(symbol="RIMM", name=None, current=Decimal('123.45'),
...     range=Range(low=Decimal('1.00'), high=Decimal('200.00')))
>>> s
{'symbol': 'RIMM', 'name': None, 'current': Decimal('123.45'), 'range':
{'low': Decimal('1.00'), 'high': Decimal('200.00')}}
```

This isn't an error (yet). Somewhere else in the application, finding `None` where a string belongs may lead to an exception being raised. A tool such as **mypy** or **pyright** will spot this kind of problem and raise the expected errors.

Dictionary design choices

Given dictionaries, typed dictionaries, and dataclasses, how do we decide how to represent attribute values of an object? We can rank the techniques like this:

1. For a lot of cases, dataclasses offer a number of helpful features with less code writing. They can be immutable, or mutable, giving us a wide range of options.
2. For cases where the data is immutable, a `NamedTuple` can be slightly more efficient than a frozen dataclass by about 5% — not much. What tips the balance here is an expensive attribute computation. While a `NamedTuple` can have properties, if the computation is very costly and the results are used frequently, it can help to compute it in advance, something a `NamedTuple` isn't good at. Check out the documentation for dataclasses and their `__post_init__()` method as a better choice in the rare case where it's helpful to compute an attribute value in advance.
3. Dictionaries are ideal when the complete set of keys isn't known in advance. When we're starting a design, we may have throwaway prototypes or proofs of concept using dictionaries. When we try to write unit tests and type hints, we often need to ramp up the formality and use a typed dictionary.

Because of the similar syntax, it's relatively easy to try different designs to see which works better for the problem. We can decide which criteria we're using for "better": faster, easier to test, uses less memory, easier to understand, and so on. Sometimes, all three converge and there's one best choice. More often, it's a trade-off.

In *Chapter 12*, we'll return to this performance question by looking at other data structure alternatives.

We'll include a side-bar on dictionary keys, hash codes, and equality before we move on to lists and sets.

Dictionary keys

To be usable as a dictionary key, an object must be **hashable**, that is, have a `__hash__()` method to convert the object's state into a unique integer value for rapid lookup in a dictionary or set. The built-in `hash()` function uses the `__hash__()` method of the object's class. For example, strings map to integers based on numeric codes for the characters in the string, while tuples combine hashes of the items inside the tuple. The reason for this is an important part of the way Python works.

Any two objects that are considered equal (such as strings with the same characters or tuples with the same values) **must** also have the same hash value.

Note the asymmetry between equality and matching hash values. If two strings have the same hash value, they could still turn out to be unequal. Think of hash equality as an approximation for an equality test: if the hashes aren't equal, don't bother looking at the details. If the hashes are equal, invest the time in checking each attribute value or each item of the tuple, or each individual character of the string.

Here's an example of two integers with the same hash value that are not actually equal:

```
>>> x = 2020
>>> y = 2305843009213695971
>>> hash(x) == hash(y)
True
>>> x == y
False
```

(The magic number? $M_{61} = 2^{61} - 1$.)

When we use these values as keys in a dictionary, a hash collision algorithm will keep them separated. The situation leads to a microscopic slowdown in these rare cases of hash collisions. This is why dictionary lookup isn't **always** immediate: a hash collision might slow down access.

The built-in mutable objects — including lists, dictionaries, and sets — cannot be used as dictionary

keys because they don't provide hash values. We can, however, create our own class of objects that are both mutable and provide a hash value; this is unsafe because a change to the object's state can make it difficult to find the key in the dictionary.

It is certainly possible to create a class with a mixture of mutable and immutable attributes and confine a customized hash computation to the immutable attributes. Because of the differences in behavior between the mutable and immutable features, this seems like it's really two objects that collaborate, not a single object with mutable and immutable features. If we refactor the two aspects into separate classes, we can use the immutable part for dictionary keys and keep the mutable part in the dictionary value.

In contrast, there are no limits on the types of objects that can be used as dictionary values. We can use a string key that maps to a list value, for example, or we can have a nested dictionary as a value in another dictionary.

Speaking of lists, it's time to dig a little more deeply into Python's `list` collection.

Lists

Python's generic list structure is integrated into a number of language features. We can, for example, visit all the items in a list without explicitly requesting an iterator object, and we can construct a list (as with a dictionary) with very simple-looking comprehension syntax. Further, list comprehensions and generator expressions turn lists into a veritable Swiss Army knife of computing functionality.

If you don't know how to create or append to a list, how to retrieve items from a list, or what *slice notation* is, we direct you to the official Python tutorial. See *section 3.1.3* (`https://docs.python.org/3/tutorial/introduction.html#lists`) and *section 5.1* (`https://docs.python.org/3/tutorial/datastructures.html#more-on-lists`) of the Python Tutorial. Also, the *Modern Python Cookbook*, *Chapter 4*, has several recipes related to lists.

In Python, lists should normally be used when we want to store several instances of the *same* type of object: lists of strings or lists of numbers. We'll often use a type hint, `list[T]`, to specify some type, `T`, of object kept in the list, for example, `list[int]` or `list[str]`.

A list of distinct types requires some care. If it's a few distinct types, then it might be described by a hint such as `list[X | Y]`, where `X` and `Y` are two type names.

Lists are ordered. Often, this is the order in which they were inserted. But they can also be sorted into other orders. We'll look at list sorting in section *Sorting lists*.

Lists are mutable, so items can be added, replaced, and removed from the list. This can be handy for reflecting the state of some more complex objects.

Like dictionaries, Python lists use an extremely efficient and well-tuned internal data structure so we can worry about what we're storing, rather than how we're storing it. The Python standard library expands on lists to provide some specialized data structures for queues and stacks. Python doesn't make a distinction between lists based on arrays and lists that use links. Generally, the built-in list data structure can serve a wide variety of purposes.

Don't use lists for collecting different attributes of individual items. Tuples, named tuples, dictionaries, and objects would all be more suitable for collecting different kinds of attribute values. Our first `Stock` data examples at the beginning of the chapter stored the current price, minimum price, and maximum price, each a different attribute with a distinct meaning in a single sequence. This wasn't a good idea, and named tuples, dataclasses, and typed dictionaries were clearly superior.

Here's a rather convoluted counterexample that demonstrates how we could perform a data frequency count using a list. It is much more complicated than the equivalent using `Counter`. It illustrates the effect that choosing the right (or wrong) data structure can have on the readability (and performance) of our code. This is demonstrated as follows:

```
import string

CHARACTERS = list(string.ascii_letters) + [" "]


def letter_frequency(sentence: str) -> list[tuple[str, int]]:
    frequencies = [(c, 0) for c in CHARACTERS]
    for letter in sentence:
        index = CHARACTERS.index(letter)
        frequencies[index] = (letter, frequencies[index][1] + 1)
    non_zero = [(letter, count) for letter, count in frequencies if count >
    0]
    return non_zero
```

This code starts with a list of possible characters. The `string.ascii_letters` attribute provides a string of all the letters, lowercase and uppercase, in order. We convert the string to a list and concatenate a one-character list with a space. The value of `CHARACTERS` is the entire domain of available characters for the frequency list.

The first line inside the function uses a list comprehension to turn the `CHARACTERS` list into a list of

tuples, assigned to `frequencies`. Then, we iterate over each of the characters in the sentence. We find the index of a character in the `CHARACTERS` list. We use that index the tuple in the `frequencies` list by creating a new tuple. This is rather difficult to read! It performs poorly and it does a bunch of pointless memory management, creating and deleting tuple objects.

Finally, we filter the list by examining each tuple and keeping only pairs where the count is greater than zero. This removes the letters we allocated space for but never saw.

Besides being longer than dictionary-based examples, the `CHARACTERS.index(letter)` operation can be very slow. The worst case is to examine each of the characters in the list for a match. On average, it will search half the list. Compare this with a dictionary that does a hash computation and examines one item for a match (except in the case of a hash collision where there's a tiny probability of examining more than one).

The type hint describes the type of the objects in the list. We summarized it as `list[tuple[str, int]]`. Each of the items in the resulting list will be a two-tuple. This lets tools such as **mypy** confirm that the operations in the code respect the structure of the list overall and each tuple within the list.

Like dictionaries, lists are objects, too. For an overview of the methods available, see *section 5.1* (`https://docs.python.org/3/tutorial/datastructures.html#more-on-lists`) of the Python Tutorial.

For the complete list of methods, see the *Sequence Types* (`https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range`) section of the Python Standard Library documentation. The documentation can be found here: `https://docs.python.org/3/library/index.html`.

Sorting lists

Without any parameters, the `sort()` method of a `list` object will generally do as expected. If we have a `list[str]` object, the `sort()` method will place the items in alphabetical order. This operation is case-sensitive, so all capital letters will be sorted before lowercase letters; that is, `Z` comes before `a`. If it's a list of numbers, they will be sorted in numerical order. If a list of tuples is provided, the list is sorted by considering the elements in the tuple in order. If a mixture containing unsortable items is supplied, the sort will raise a `TypeError` exception.

If we want to place objects of classes we've defined ourselves into a list and make those objects sortable, we have to do a bit more work. The special `__lt__()` method (which stands for *less than*)

must be defined on the class to make instances of that class comparable. The `sort()` method on the list will access this method on each object to determine where it goes in the list. This method should return `True` if an instance of a class is somehow less than the passed argument value, and `False` otherwise.

Often, when we need comparisons like this, we'll define a dataclass. As discussed in section *Dataclasses*, the `@dataclass(order=True)` decorator will ensure that all of the comparison methods are built for us. A named tuple also has the ordering operations defined by default.

One tricky situation that arises with sorting is handling a data structure sometimes called a **tagged union**. A union is a description of an class of objects where one or more attributes are not **always** relevant. If an attribute's relevance depends on another attribute's value, this can be seen as a union of distinct subtypes with a tag to distinguish between the two types.

Here's some example data, where a tag value, the **Data Source** column, is required to decide how to interpret the remaining columns. Some values of **Data Source** tell us to use the timestamp, whereas other values tell us to use the creation date.

Data Source	Timestamp	Creation Date	Name, Owner, etc.
Local	1607280522.68012		"Some File", etc.
Remote		"2020-12-06T13:47:52.849153"	"Another File", etc.
Local	1579373292.452993		"This File", etc.
Remote		"2020-01-18T13:48:12.452993"	"That File", etc.

Table 8.1: Sample Data

How can we sort these into a single, coherent order? We'd like to have a single, consistent data type in our list, but the source data has two distinct subtypes.

A simple-seeming `if row.data_source == "Local":` can work to distinguish values, but it can be confusing logic for tools such as **mypy** to work with. One or two *ad hoc* `if` statements aren't too bad, but the design principle of throwing `if` statements at the problem isn't very scalable.

In this example, we can consider **Timestamp** as the preferred representation. This means we only need to compute timestamps from the creation date string for the items where the data source is "Remote." In this example, either the float value or the string would sort into an order properly. This happens to work out well because the string is in the carefully designed ISO format. If it were in American month-day-year format, it would require conversion to a timestamp to be useful.

Converting all of the various input formats to Python's native `datetime.datetime` objects is another

choice. This has the advantage of being distinct from any of the input formats. While this is a little more work, it gives us more flexibility because we're not tied to a source data format — a format that may change in the future. The concept is to make every variant input format convert to a single, common `datetime.datetime` instance.

What's central is treating the two subtypes as if they're a single class of objects. This doesn't always work out well. Often, this is a design constraint that sneaks up on us when we have additional customers or additional sources of data.

We'll start an implementation with a single type that supports both subtypes of data. This is not ideal, but it matches the source data and is often how we start tackling this kind of data. Here's the essential class definition:

```
from typing import cast, Any
from dataclasses import dataclass
import datetime
from datetime import timezone


@dataclass(frozen=True)
class MultiItem:
    data_source: str
    timestamp: float | None
    creation_date: str | None
    name: str
    owner_etc: str

    def __lt__(self, other: Any) -> bool:
        if self.data_source == "Local":
            self_datetime = datetime.datetime.fromtimestamp(
                cast(float, self.timestamp), tz=timezone.utc
            )
        else:
            self_datetime = datetime.datetime.fromisoformat(
                cast(str, self.creation_date)
            ).replace(tzinfo=timezone.utc)
        if other.data_source == "Local":
            other_datetime = datetime.datetime.fromtimestamp(
                cast(float, other.timestamp), tz=timezone.utc
            )
        else:
            other_datetime = datetime.datetime.fromisoformat(
```

```
                cast(str, other.creation_date)
            ).replace(tzinfo=timezone.utc)
        return self_datetime < other_datetime
```

The `__lt__()` method compares an object of the `MultiItem` class to another instance of the same class. Because there are two implicit subclasses, we have to check the tag attributes, `self.data_source` and `other.data_source`, to see which of the various combinations of fields we're dealing with. We'll do a conversion from a timestamp or a string into a common representation. Then, we can compare the two common representations.

The conversion processing is nearly duplicate code. Later in this section, we will look at refactoring this to remove the redundancy. The `cast()` operations are required to make it clear to **mypy** that the item will not be `None`. While we know the rules that match the tag (the **Data Source** column) and the two kinds of values, those rules need to be stated in a way so that **mypy** can exploit them. The `cast()` operation does nothing; it exists to tell **mypy** what the data will be at runtime.

The `cast()` operation is a claim about the intent and the design, with no runtime impact. No conversion happens, nor does it do any runtime type-checking.

The following output illustrates this class in action when it comes to sorting:

```
>>> mi_0 = MultiItem("Local", 1607262522.000000, None, "Some File", "etc.
0")
>>> mi_1 = MultiItem("Remote", None, "2020-12-06T13:47:52.000001", "Another
File", "etc. 1")
>>> mi_2 = MultiItem("Local", 1579355292.000002, None, "This File", "etc.
2")
>>> mi_3 = MultiItem("Remote", None, "2020-01-18T13:48:12.000003", "That
File", "etc. 3")
>>> file_list = [mi_0, mi_1, mi_2, mi_3]
>>> file_list.sort()
```

```
>>> from pprint import pprint
>>> pprint(file_list)
[MultiItem(data_source='Local',
           timestamp=1579355292.000002,
           creation_date=None,
```

```
           name='This File',
           owner_etc='etc. 2'),
 MultiItem(data_source='Remote',
           timestamp=None,
           creation_date='2020-01-18T13:48:12.000003',
           name='That File',
           owner_etc='etc. 3'),
 MultiItem(data_source='Remote',
           timestamp=None,
           creation_date='2020-12-06T13:47:52.000001',
           name='Another File',
           owner_etc='etc. 1'),
 MultiItem(data_source='Local',
           timestamp=1607262522.0,
           creation_date=None,
           name='Some File',
           owner_etc='etc. 0')]
```

The comparison rules were applied among the various subtypes that were conflated into a single class definition. If the rules are more complex, however, this use of `if` statements can become unwieldy.

Only the `__lt__()` method is required to enable sorting. To be complete, it may be helpful for the class to implement the similar `__gt__()`, `__eq__()`, `__ne__()`, `__ge__()`, and `__le__()` methods. This ensures that all of the `<`, `>`, `==`, `!=`, `>=`, and `<=` operators also work properly.

One approach is by implementing `__lt__()` and `__eq__()`, and then applying the `@total_ordering` class decorator to supply the rest:

```
from functools import total_ordering


@total_ordering
@dataclass(frozen=True)
class MultiItemTO:
    data_source: str
    timestamp: float | None
    creation_date: str | None
    name: str
    owner_etc: str
```

```
    @property
    def datetime(self) -> datetime.datetime:
        if self.data_source == "Local":
            return datetime.datetime.fromtimestamp(
                cast(float, self.timestamp), tz=timezone.utc
            )
        else:
            return datetime.datetime.fromisoformat(
                cast(str, self.creation_date)
            ).replace(tzinfo=timezone.utc)

    def __eq__(self, other: object) -> bool:
        return self.datetime == cast(MultiItemTO, other).datetime

    def __lt__(self, other: object) -> bool:
        return self.datetime < cast(MultiItemTO, other).datetime
```

When we provide some combination of `<` (or `>`) and `=`, the `@total_order` decorator can deduce the remaining logic operator implementations. For example, $a \geq b \equiv \neg a < b$. The implementation of `__ge__(self, other)` is `not self < other`.

This also suggests a new design principle. When confronted with two subtypes — in this case, local times with one format and remote times with a distinct format — it helps to convert to some unifying representation. We've added the `datetime` property to compute a single, standard `datetime.datetime` object as needed.

Rather than add sophisticated sortability features to the class, we can extract a sortable attribute value when we need it. This will encapsulate the comparisons to those few places where the `datetime` attribute is actually needed. We can use this to provide a "key extraction" function to the `sort()` method. This argument to `sort()` is a function that translates each original object in a simpler object that supports comparison. In our case, we'd like a function to extract either the `timestamp` or the `creation_date` for comparison.

It turns out that we only need to define the `datetime` property. This is a subset of the `MultiItemTO` definition shown previously. It doesn't need the `@total_order` decorator, or the `__lt__()` or `__eq__()` methods.

Here's how we use a class with only the `datetime` attribute to compare objects:

```
>>> from operator import attrgetter
>>> file_list.sort(key=attrgetter('datetime'))
```

We've used the `attrgetter` operator to get a named attribute value and use this for comparison when sorting. When working with tuples or dictionaries, `itemgetter()` can be used to extract a specific item by key or position. There's even `methodcaller()`, which returns the result of a method call on the object being sorted. Refer to the `operator` module documentation for more information.

This segregates the sorting interface from other aspects of the class, leading to a pleasant simplification. We can leverage this kind of design to provide other kinds of sorts. We might, for example, sort by name only. This is slightly simpler because no conversion method is required:

```
>>> file_list.sort(key=lambda item: item.name)
```

We've created a lambda object, a tiny no-name function that takes an item as an argument and returns the value of `item.name`. A lambda is a function, but it doesn't have a name, and it can't have any statements. It only has a single expression. If you need statements (for example, a `try-except` clause), you need a conventional function definition instead of a lambda.

There's rarely one single sort order for data objects. Providing the key function as part of the `sort()` method lets us define a wide variety of sorting rules without creating complex class definitions.

After looking at dictionaries and now lists, we can turn our attention to sets.

Sets

Lists are extremely versatile tools that suit many container object applications. But they are not useful when we want to ensure that objects in a collection are unique. For example, a song library may contain many songs by the same artist. If we want to sort through the library and create a list of all the artists, we would have to constantly check the list to see whether we've added the artist already, to avoid adding them again.

This is where sets come in. Sets come from mathematics, where they represent an unordered group of unique items. We can try to add an item to a set five times, but "is a member of a set" doesn't change after the first time we add it.

For more information on sets, see *section 5.5* (`https://docs.python.org/3/tutorial/datastructures.html#sets`) of the Python Tutorial. Also, the *Modern Python Cookbook*, *Chapter 4*, has

several recipes related to sets.

In Python, sets can hold any hashable object, not just strings or numbers. Hashable objects implement the `__hash__()` method. These are the same objects that can be used as keys in dictionaries; so again, mutable lists, sets, and dictionaries are out. Like mathematical sets, an object is or is not a member of a given set; it can't be a member multiple times.

If we're trying to create a collection of song artists, we can add the names to the set. This example starts with a list of `(song, artist)` tuples and creates a set of the artists:

```
>>> song_library = [
...     ("Phantom Of The Opera", "Sarah Brightman"),
...     ("Knocking On Heaven's Door", "Guns N' Roses"),
...     ("Captain Nemo", "Sarah Brightman"),
...     ("Patterns In The Ivy", "Opeth"),
...     ("November Rain", "Guns N' Roses"),
...     ("Beautiful", "Sarah Brightman"),
...     ("Mal's Song", "Vixy and Tony"),
... ]
>>> artists = set()
>>> for song, artist in song_library:
...     artists.add(artist)
```

There is no built-in syntax for an empty set as there is for empty lists and empty dictionaries; we create a set using the `set()` constructor. However, we can use the curly braces (borrowed from dictionary syntax) to create a set, so long as the set contains simple values. If we use colons, it's a dictionary literal. If we just separate values with commas, it's a set, for example `'value', 'value2'`.

Items can be added individually to the set using the `add()` method, and a set can be updated in bulk using the `update()` method. If we run the script shown previously, we see that the set works as advertised:

```
>>> artists
{'Opeth', "Guns N' Roses", 'Vixy and Tony', 'Sarah Brightman'}
```

If you're paying attention to the output, you'll notice that the items are not printed in the order they were added to the set. Indeed, each time you run this, you may see the items in a different order, depending on the hash randomization key in use.

As with lists and dictionaries, there's a set comprehension that builds a set from a source object. It looks like this:

```
>>> artists = set(artist for _, artist in song_library)
```

The album name is assigned to the _ variable and ignored. The artist name is used to build a set.

Sets are inherently unordered due to a hash-based data structure used for efficient access to the members. Because of this lack of ordering, sets cannot have items looked up by index. The primary purpose of a set is to divide the world into two groups: *objects in the set*, and *objects not in the set*. It is easy to check whether an object is in a set or to iterate over the items in a set. Concepts such as sorting don't apply: we have to convert the set to a list. This output shows all three of these activities:

```
>>> "Opeth" in artists
True
>>> alphabetical = list(artists)
>>> alphabetical.sort()
>>> alphabetical
["Guns N' Roses", 'Opeth', 'Sarah Brightman', 'Vixy and Tony']
```

```
>>> for artist in artists:
...     print(f"{artist} plays good music")
...
Opeth plays good music
Guns N' Roses plays good music
Vixy and Tony plays good music
Sarah Brightman plays good music
```

Each time you run this final example, you may see the items in a different order.

The primary feature of a set is uniqueness. Because an item can only appear once, sets are often used to deduplicate data. We'll touch an a few of the many features of sets.

The `union()` method is the most common and easiest to understand. It takes a second set as a parameter and returns a new set that contains all elements that are in *either* of the two sets; if an element is in both original sets, it will only show up once in the new set. Union is like a logical `or` operation. Indeed, the `|` operator can be used on two sets to perform the union operation, if you don't like calling methods.

Conversely, the `intersection()` method accepts a second set and returns a new set that contains only those elements that are in *both* sets. It is like a logical `and` operation, and can also be referenced using the `&` operator.

The `symmetric_difference()` method tells us what's distinct; it is the set of objects that are in one set or the other, but not in both. It uses the `^` operator. The following example illustrates these methods by comparing some artists preferred by two different people:

```
>>> dusty_artists = {
...     "Sarah Brightman",
...     "Guns N' Roses",
...     "Opeth",
...     "Vixy and Tony",
... }
>>> steve_artists = {"Yes", "Guns N' Roses", "Genesis"}
```

Here are three examples of union, intersection, and symmetric difference:

```
>>> all = dusty_artists | steve_artists
>>> all
{'Genesis', 'Vixy and Tony', 'Sarah Brightman', 'Opeth', "Guns N' Roses",
'Yes'}

>>> both = dusty_artists.intersection(steve_artists)
>>> both
{"Guns N' Roses"}

>>> not_both = dusty_artists ^ steve_artists
>>> not_both
{'Genesis', 'Sarah Brightman', 'Opeth', 'Vixy and Tony', 'Yes'}
```

There is an intersection operator, `&`, but we used the `intersection()` method in the preceding example. Some people like the operator notation. Some people like the method notation.

The order for the set elements isn't fixed. You may see the results in a different order. If you want to see the same results as we get when unit testing the code for the book, you need to set the `PYTHONHASHSEED` environment variable to "42". This will replace the usual hash randomization with a value that provides consistent results.

It is valuable to know that sets are much more efficient than lists when checking for membership using the `in` keyword. If you use the `value in container` syntax on a set or a list, it will return

`True` if one of the elements in `container` is equal to `value`. However, when searching a list, it will look at every object in the container until it finds the value. The bigger the list, the longer this search takes. When searching a set, on the other hand, it computes a hash the value and checks for membership. The time is (almost) constant. (The slight variability comes from the tiny possibility of hash collisions.)

Three types of queues

We'll look at an application of the list structure to create a queue. A queue is a special kind of buffer, summarized as **First In First Out** (**FIFO**). The idea is to act as a temporary stash so one part of an application can write to the queue while another part consumes items from the queue.

A database might have a queue of data to be written to disk. When our application performs an update, the local cache version of the data is updated so all other applications can see the change. The write to the disk, however, may be placed in a queue for a writer to deal with a few milliseconds later.

When we're looking at files and directories, a queue can be a handy place to stash details of the directories to be processed later. We'll often represent a directory as the path from the root of the filesystem to the file of interest. The algorithm works like this:

$q \leftarrow \emptyset$
$q \leftarrow q +$ **starting directory**
while $q \neq \emptyset$ **do**
    $d \leftarrow \text{next}(q)$
    if d **is a file then**
        process(d)
    else if d **is a directory then**
        $q \leftarrow q + \text{contents}(d)$
    end if
end while

We can visualize this list-like structure as growing via the `append()` method and shrinking via `pop(0)`. The `append()` method puts items at the end. The `pop(0)` method takes the next item from the front of the queue.

Figure 8.1 shows the operations on a queue:

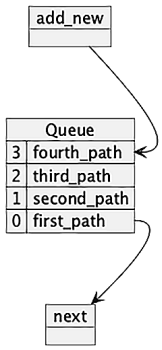


Figure 8.1: Queue concept

The idea is for the queue to grow and shrink: each directory grows the queue and each file shrinks the queue. Eventually, all the files and directories have been processed and the queue is empty.

We have several ways to implement a queue in Python:

1. Use a **list** with the the `pop()` and `append()` methods.
2. Use the `collections.deque` class, which supports `popleft()` and `append()` methods. A "deque" is a double-ended queue. This is an elegant queue implementation that's faster than a simple list for the specific operations of appending and popping.
3. Use the `queue` module. This queue class is often used for multithreading, but it can also be used for our single thread application to examine a directory tree. This uses methods named `get()` and `put()`. Since this structure is designed for concurrency, it locks the data structure to assure that each change is atomic and can't be interrupted by other threads. This is the subject of *Chapter 14*.

The `heapq` module also provides a queue, but it does some extra processing. It keeps items in priority order, not the order they were put into the queue, breaking the FIFO expectation.

Each of these implementations is slightly different. This suggests we might want to create handy wrapper classes around them to provide a uniform interface. We can create class definitions like the following:

```
from pathlib import Path


class ListQueue(list[Path]):
```

```
    def put(self, item: Path) -> None:
        self.append(item)

    def get(self) -> Path:
        return self.pop(0)

    def empty(self) -> bool:
        return len(self) == 0
```

This shows the three essential operations for a queue. We can put something into the queue, appending it to the end. We can get something from the queue, removing the item at the head of the queue. Finally, we can ask whether the queue is empty. We've layered this on a `list` class by extending it to add three new methods: `put()`, `get()`, and `empty()`.

Next is a slightly different implementation. This uses the `collections.deque` class:

```
from pathlib import Path
from typing import Deque


class DeQueue(Deque[Path]):
```

```
    def put(self, item: Path) -> None:
        self.append(item)

    def get(self) -> Path:
        return self.popleft()

    def empty(self) -> bool:
        return len(self) == 0
```

It's hard to see the distinction between this implementation and the generic list implementation. It turns out the `popleft()` method is a higher-speed version of `pop(0)` in a conventional list. Otherwise, this looks very similar to the list-based implementation.

Here's a final version that uses the queue module. This queue module's implementation uses locks to prevent the data structure from being damaged by concurrent access across multiple threads. It's generally opaque to us, except as a tiny performance cost:

```
from pathlib import Path
import queue

class ThreadQueue(queue.Queue[Path]):
```

```
    pass
```

This implementation works because we decided to use the `Queue` class interface as the template for the other two classes. This meant we didn't have to do any real work to implement this class; this design was the overall target for the other class designs.

The type hint is similar to the others. The `Queue` class definition is a generic type, and this code provides a type parameter to specify that the queue will always contain `Path` objects. It does nothing unique; it merely provides a type-specific wrapper for code that's otherwise generic.

These three classes are similar with respect to the three defined methods. We could define an abstract base class for them. Or we could provide the following type hint:

```
PathQueue = ListQueue | DeQueue | ThreadQueue
```

This `PathQueue` type hint summarizes all three types, allowing us to define an object of any of these three classes to use for the final implementation choice.

The question of "which is better?" is answered by the standard response of "what dimension are you measuring?"

- For single-threaded applications, `collections.deque` will be fastest, if that's the goal.
- For multithreaded applications, `queue.Queue` is required to provide a data structure that can be read and written by multiple concurrent threads. We'll return to this in *Chapter 14*.

While we can often leverage a built-in structure, such as the generic `list` class, for a wide variety of purposes, it may not be ideal. The other two implementations offer advantages over the built-in list. Python's standard library, and the broader ecosystem of external packages available through the Python Package Index (PyPI), can provide improvements over generic structures. What's important

is having a specific goal in mind before searching high and low for a "best" package. In our example, the performance difference between `deque` and `list` is small. The time is dominated by the OS work required to gather the raw data. For a large filesystem, perhaps spanning multiple hosts, the tiny difference will add up.

Python's object orientation gives us the latitude to explore design alternatives. We should feel free to try more than one solution to a problem as a way to better understand the problem, and arrive at an acceptable solution.

Recall

We've explored a variety of built-in Python data structures in this chapter. Python lets us do a great deal of object-oriented programming without the overheads of numerous, potentially confusing, class definitions. We can rely on a number of built-in classes where they fit our problem.

In this chapter, we looked at the following:

- Tuples and named tuples let us leverage a simple collection of attributes. We can extend the `NamedTuple` definition to add methods when those are necessary.
- Dataclasses provide sophisticated collections of attributes. A variety of methods can be provided for us, simplifying the code we need to write.
- Dictionaries are an essential feature, used widely in Python. There are many places where keys are associated with values. The syntax for using the built-in dictionary class makes it easy to use.
- Lists and sets are also first-class parts of Python; our applications can make use of these.
- We also looked at three types of queues. These are more specialized structures with more focused patterns of access than a generic list object. The idea of specialization and narrowing the domain of features can lead to performance improvements, too, making the concept widely applicable.

Exercises

The best way to learn how to choose the correct data structure is to do it wrong a few times (intentionally or accidentally!). Take some code you've recently written, or write some new code that uses a list. Try rewriting it using some different data structures. Which ones make more sense? Which ones don't? Which have the most elegant code?

Try this with a few different pairs of data structures. You can look at examples you've done for previous chapter exercises. Are there objects with methods where you could have used dataclasses, `namedtuple`, or `dict` instead? Attempt both and see. Are there dictionaries that could have been sets because you don't really access the values? Do you have lists that check for duplicates? Would a set suffice? Or maybe several sets? Would one of the queue implementations be more efficient? Is it useful to restrict the API to the top of a stack rather than allowing random access to the list?

Have you written any container objects recently that you could improve by inheriting a built-in and overriding some of the *special* double-underscore methods? It takes some work to find out which methods need overriding. You can get some ideas using built-in `dir()` and `help()` functions to explore a class. The Python library reference provides a lot of details.

Are you sure inheritance is the correct tool to apply? Could a composition-based solution be more effective? Try both (if it's possible) before you decide. Try to find different situations where each method is better than the other.

If you were familiar with the various Python data structures and their uses before you started this chapter, you may have been bored. But if that is the case, there's a good chance you use data structures too much! Look at some of your old code and rewrite it to use more self-made classes. Carefully consider the alternatives and try them all out. Which one makes for the most readable and maintainable system?

The `MultiItem` example was trying to handle a variety of subtypes, each with a distinct set of optional fields. The presence of an optional attribute is a suggestion that — perhaps — there are distinct classes struggling to separate from each other. What happens if we distinguish between two closely related but distinct classes: `LocalItem` (which uses `timestamp`) and `RemoteItem` (which uses `created_date`)? We can define a common type hint as `LocalItem | RemoteItem`. If each class has a property such as `creation_datetime` that computes a `datetime.datetime` object, would processing be simpler? Build the two classes; create some test data. How does it look to separate the two subtypes?

Always critically evaluate your code and design decisions. Make a habit of reviewing old code and take note of whether your understanding of *good design* has changed since you wrote it. Software design has a large aesthetic component, and like artists with oil on canvas, we all have to find the style that suits us best.

Summary

We've covered several built-in data structures and attempted to understand how to choose them for specific applications. Sometimes, the best thing we can do is create a new class of objects, but often, one of the built-ins provides exactly what we need. When it doesn't, we can always use inheritance or composition to adapt them to our use cases. We can even override special methods to completely change the behavior of built-in syntaxes.

In the next chapter, we'll discuss how to integrate the object-oriented and not-so-object-oriented aspects of Python. Along the way, we'll discover that it's more object-oriented than it looks at first sight!

9

The Intersection of Object-Oriented and Functional Programming

There are many aspects of Python that appear more like functional programming than object-oriented programming. Although object-oriented programming is the focus of this book, there are compelling use cases for functional programming techniques. With Python, the underlying implementation is object-oriented. However, using some functional design techniques can make code more expressive. In this chapter, we'll be covering a grab bag of Python features that are not strictly object-oriented:

- Built-in functions that take care of common tasks in one call
- An alternative to method overloading
- Functions as objects

This chapter will scratch the surface of a very deep topic. For a deeper dive, see *Functional Python Programming*,

https://www.packtpub.com/en-us/product/functional-python-programming-9781788627061.

We'll start this chapter by looking at some of Python's built-in functions. Some of these are closely related to class definitions, allowing us to use a functional style of programming with the underlying objects.

Python built-in functions

There are numerous functions in Python that perform a task or calculate a result on certain types of objects without seeming to involve obvious methods of the underlying class. They usually abstract common calculations that apply to multiple types of classes. This is duck typing at its best; these functions accept objects that have certain attributes or methods, and are able to perform generic operations using those methods. We've used many of the built-in functions already. We'll take a look at three common ones:

- The `len()` function
- The `reversed()` function
- The `enumerate()` function

We'll start with the `len()` function.

The `len()` function

The `len()` function returns the number of items in some kind of container object, such as a dictionary, set, tuple, or list. You've seen it before, demonstrated as follows:

```
>>> len([1, 2, 3, 4])
4
```

You may wonder why these objects don't have a length method or property instead of having to call a function on them. Technically, they do have a method. Most objects that `len()` will apply to will define a method called `__len__()` that computes the value.

This means an expression like `len(myobj)` will call `myobj.__len__()`.

Why should we use the `len()` function instead of the `__len__()` method? Obviously, `__len__()` is a special double-underscore method, suggesting that we shouldn't call it directly. Why have a `length()` method we can call directly? There must be an explanation for this. The Python

developers don't make such design decisions lightly.

The main reason is efficiency. When we evaluate the `__len__()` method of an object, the object has to look the method up in its namespace, and, if the special `__getattribute__()` method (which is called every time an attribute or method on an object is accessed) is defined on that object, it has to be called as well. If the name isn't in the instance, then each class must be searched in Method Resolution Order until the method is found. The `len()` function doesn't encounter any of this overhead. It actually calls the `__len__()` method on the underlying class directly, so `len(myobj)` maps to `MyObj.__len__(myobj)`.

Another reason is clarity. The `len(x)` syntax can be easier to understand than `x.length()`. The notation fits mathematical formalisms — where functional notation is more common.

It's not perfectly clear, but any class that includes the `Sized` abstract base class will work with the `len()` function. The names don't align well, but the concept of a specific protocol having a specific function that supports that protocol is a pleasant symmetry.

The reversed() function

The `reversed()` function takes any sequence as input and returns a copy of that sequence in reverse order. It is normally used in `for` statements when we want to iterate over items from back to front.

Similar to the `len()` function, `reversed()` prefers to call the `__reversed__()` method on the class for the parameter. If that method does not exist, `reversed` builds the reversed sequence itself using calls to `__len__()` and `__getitem__()`, which are used to define a sequence. We only need to override `__reversed__()` if we want to somehow customize or optimize the process, as demonstrated in the following code:

```
from collections.abc import Sequence, Iterator
from typing import Any


class CustomSequence(Sequence[Any]):
    def __init__(self, arg: Sequence[Any]) -> None:
        self._list = arg
    def __len__(self) -> int:
        # This doesn't seem right, does it?
        return 5
    def __getitem__(self, index: int | slice) -> Any:
        return f"x{index}"
```

```
class FunkyBackwards(list[Any]):
    def __reversed__(self) -> Iterator[Any]:
        return iter("BACKWARDS!")
```

We've called out the `__len__()` method for ignoring the value of `self._list` and returning a literal 5 every time. If the list actually as 5 items, this will work. It's sketchy code, but it helps reveal how Python works internally.

Let's exercise this function on three different kinds of lists:

```
>>> generic = [1, 2, 3, 4, 5]
>>> custom = CustomSequence([6, 7, 8, 9, 10])
>>> funkadelic = FunkyBackwards([11, 12, 13, 14, 15])
>>> for sequence in generic, custom, funkadelic:
...     print(f"{sequence.__class__.__name__}: ", end="")
...     for item in reversed(sequence):
...         print(f"{item}, ", end="")
...     print()
list: 5, 4, 3, 2, 1,
CustomSequence: x4, x3, x2, x1, x0,
FunkyBackwards: B, A, C, K, W, A, R, D, S, !,
```

The `for` statement prints reversed versions of a generic list object, and instances of the `CustomSequence` class and the `FunkyBackwards` class. The output shows that the `reversed()` works on all three of them, but can have very different results.

When we reverse `CustomSequence`, the `__getitem__()` method is called for each item; this returns a string of `x` followed by the index; not the actual value from the sequence. For `FunkyBackwards`, the `__reversed__()` method returns a string, each character of which is output individually in the `for` statement.

The `CustomSequence` class is incomplete. It doesn't define a proper version of the `__iter__()` method, so a forward `for` loop over them will never end. This is the subject of *Chapter 10*.

The enumerate() function

Sometimes, when we're examining items in a container with a `for` statement, we want access to the index (the current position in the container) as well as the current item being processed. The `for`

statement doesn't provide us with indexes, but the `enumerate()` function gives us something better: it creates a sequence of tuples, where the first object in each tuple is the index and the second is the original item.

This is useful because it assigns an index number. It works well for sets or dictionaries where there isn't an inherent index order to the values. It also works for text files, which have an implied line number. Consider some simple code that outputs each of the lines in a file with the associated line numbers. Here's some code with the output:

```
>>> from pathlib import Path
>>> with open(Path.cwd() / "data" / "sample_data.md") as source:
...     for index, line in enumerate(source, start=1):
...         print(f"{index:3d}: {line.rstrip()}")
  1: # Python 3 Object-Oriented Programming
  2:
  3: ## Chapter 9. The Intersection of Object-Oriented and Functional
  Programming
  4:
  5: Some sample data to show how the `enumerate()` function works.
```

The `enumerate()` function is an iterable: it yields tuples until it's out of data. Our `for` statement splits each tuple into two values, and the `print()` function formats them. We used the optional `start=1` on the `enumerate()` function to provide a conventional 1-based sequence of line numbers.

We've only touched on a few of the more important Python built-in functions. As you can see, many of them leverage the foundational object-oriented concepts. They can also work in a functional programming paradigm. There are numerous others in the standard library; some of the more interesting ones include the following:

- `abs()`, `str()`, `repr()`, `pow()`, and `divmod()` map directly to the special methods `__abs__()`, `__str__()`, `__repr__()`, `__pow__()`, and `__divmod__()`
- `bytes()`, `format()`, `hash()`, and `bool()` also map directly to the special methods `__bytes__()`, `__format__()`, `__hash__()`, and `__bool__()`

Section 3.3, *Special method names* of *The Python Language Reference*, provides the details of these mappings. Other interesting built-in functions include the following:

- `all()` and `any()`, which accept an iterable object and return `True` if all, or any, of the items evaluate to true (such as a non-empty string or list, a non-zero number, an object that is not

`None`, or the literal `True`).

- `eval()`, `exec()`, and `compile()`, which execute string as code inside the interpreter. Be careful with these ones; they are not safe, so don't execute code an unknown user has supplied to you (in general, assume all unknown users are malicious, foolish, or both).
- `hasattr()`, `getattr()`, `setattr()`, and `delattr()`, which allow attributes on an object to be manipulated by their string names.
- `zip()`, which takes two or more sequences and returns a new sequence of tuples, where each tuple contains a single value from each sequence.
- And many more! See the interpreter help documentation for each of the functions listed in `help("builtins")`.

What's central is avoiding the narrow viewpoint that an object-oriented programming language must always use `object.method()` syntax for everything. Python strives for readability, and a simple `len(collection)` seems more clear than the slightly more consistent *potential* alternative, `collection.len()`.

An alternative to method overloading

One prominent feature of some object-oriented programming languages is a feature called **method overloading**. Method overloading refers to having multiple method definitions with the same name, each of which accept different sets of parameters. In statically typed languages, this is useful if we want to have a method that accepts either an integer or a string, for example. In non-object-oriented languages, we might need two functions, called `add_s()` and `add_i()` — one for strings, one for integers — to accommodate such situations. In statically typed object-oriented languages, one approach is to permit definition of two methods, both called `add`, one that accepts strings, and one that accepts integers.

In Python, we've already seen that we only need one method, which accepts any type of object. It may have to do some matching of the object's type (for example, if it is a string, convert it to an integer), but only one method is required. The `match-case` statement does sophisticated structural type matching, allowing tremendous flexibility.

We have to distinguish between two varieties of overloading here:

- Overloading a parameter to allow alternative types. There are two approaches for this:
  - In *Chapter 6*, a special `@overload` decorator was used for a particularly complicated

case.

  - Another choice is a type union hint to show that a parameter can have values of type `int | str`.

  These alternative definitions are ways to clarify our intent so tools like **mypy** can confirm that we're using the overloaded parameter properly.

- Overloading the method as a whole by using more complex patterns of parameters, including optional parameters.

For example, an email message method might come in two versions, one of which accepts a parameter for the *from* email address. The other method might look up a default *from* email address instead. Python lets us define parameters that are optional. This has the effect of allowing one method to have distinct signatures using different patterns of parameter.

We've seen some of the possible ways to send argument values to methods and functions in previous examples, but now we'll cover all the details. The simplest function accepts no parameters. It is defined with a name and empty `()` for the parameters.

When calling any function, the values for the positional parameters must be specified in order, and no parameter can be omitted. This is the most common way in which we've specified parameters in our previous examples. Here's how this looks:

```
from typing import Any


def mandatory_params(x: Any, y: Any, z: Any) -> str:
    return f"{x=}, {y=}, {z=}"
```

To call it, we use the following:

```
>>> a_variable = 42
>>> mandatory_params("a string", a_variable, True)
"x='a string', y=42, z=True"
```

Python code is generic with respect to type. This means that any type of object can be passed as an argument value: an object, a container, a primitive, even functions and classes. The preceding call shows a string literal, the value of a variable, and a Boolean literal passed into the function.

Generally, our applications are not completely generic. We've designed them around specific types

that are required in order for the function or method to work properly. That's why we often provide type hints to narrow the domain of possible values. In the rare case when we're writing something truly generic, we can use the `typing.Any` hint to tell tools like **mypy** that we really mean that any object is usable.

We can use tools to locate code with `Any`. The **mypy** tool uses the `-/-disallow-any-expr` option to flag lines that may be in need of some clarity on what types are really important.

Default values for parameters

If we want to make a parameter's value optional, we do this by providing a default value. If the calling code does not supply an argument value for the parameter, it will be assigned the given default value. This means calling code can still choose to override the default by passing in a different value. If a value of `None` is used as the default for optional parameter values, we'll often use the `SomeType | None = None` hint to make it clear that the argument value must be of a given type, otherwise the `None` object will be used.

Here's a function definition with default parameter definitions:

```
def latitude_dms(
    deg: float, min: float, sec: float = 0.0, dir: str | None = None
) -> str:
    if dir is None:
        dir = "N"
    return f"{deg:02.0f}° {min+sec/60:05.3f}{dir}"
```

The first two parameters are mandatory and must be provided. The last two parameters have default argument values and can be omitted.

The `sec` parameter has a useful value of `0.0`. The `dir` parameter, has the generic place-holder of `None`. In this case, a proper default is computed within the body of the function.

There are several ways we can call this function. We can supply all argument values in order, as though all the parameters were positional, as can be seen in the following:

```
>>> latitude_dms(36, 51, 2.9, "N")
'36° 51.048N'
```

Alternatively, we can supply just the mandatory positional argument values in order, allowing one

of the keyword parameters (`sec`) to use a default value, and providing a keyword argument for the `dir` parameter:

```
>>> latitude_dms(38, 58, dir="N")
'38° 58.000N'
```

We've used equals sign syntax when calling a function to skip default values that we aren't interested in.

Surprisingly, we can even use the equals sign syntax to mix up the order of arguments for the positional parameters, so long as all the parameters are given an argument value:

```
>>> latitude_dms(38, 19, dir="N", sec=7)
'38° 19.117N'
```

You may occasionally find it useful to make a *keyword-only* parameter. To use this, the argument value must be supplied as a keyword argument. You can do that by defining a parameter of `*` to separate the keyword-optional parameters from the keyword-only parameters:

```
def kw_only(x: Any, y: str = "defaultkw", *, a: bool, b: str = "only") ->
str:
    return f"{x=}, {y=}, {a=}, {b=}"
```

The `*` is not used as a multiplication operator here. Instead, it's used all alone to break the parameters into two groups.

This function has one positional parameter, `x`, and three keyword parameters, `y`, `a`, and `b`. The `x` and `a` parameters are both mandatory, but `a` can only be passed as a keyword argument. The `y` and `b` are both optional with default values, but if `b` is supplied, it can only be a keyword argument.

Because there are so many parameter definitions, we've sprawled them out onto multiple physical lines. The enclosing `()`'s make this one logical line, and make the whole somewhat easier to read.

This function fails at runtime if you don't pass `a`:

```
>>> kw_only('x')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: kw_only() missing 1 required keyword-only argument: 'a'
```

It also fails at runtime if you try to pass `a` as a positional argument:

```
>>> kw_only('x', 'y', 'a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: kw_only() takes from 1 to 2 positional arguments but 3 were given
```

But you can pass `a` and `b` as keyword arguments:

```
>>> kw_only('x', a='a', b='b')
"x='x', y='defaultkw', a='a', b='b'"
```

For the most part, tools to check type hints will spot invalid use of these functions. We can, of course, baffle the tools with really obscure coding techniques, like using `kw_only(**some_dict)` to provide the arguments as a dictionary.

We can also mark parameters as being supplied only by position. We do this by providing these names before a single `/` parameter to separate the positional-only parameters from the more flexible parameters that follow.

```
>>> pos_only(x=2, y="three")
Traceback (most recent call last):
```

The `/` is not used as the true division operator here. Instead, it's used all alone to break the parameters into two groups.

This function requires argument values for the `x` and `y` parameters to be the first two. Further, named arguments for `x` and `y` are specifically not permitted. Here's what happens at runtime if we try:

```
>>> pos_only(x=2, y="three")
Traceback (most recent call last):
  ...
  File "<doctest hint_examples.__test__.test_pos_only[0]>", line 1, in
  <module>
    pos_only(x=2, y="three")
TypeError: pos_only() got some positional-only arguments passed as keyword
arguments: 'x, y'
>>> pos_only(2, "three")
```

```
"x=2, y='three', z=None"
>>> pos_only(2, "three", 3.14159)
"x=2, y='three', z=3.14159"
```

We must provide argument values for the first two parameters, `x` and `y`, positionally. The third parameter, `z`, can be provided positionally, or with a keyword.

This can feel like it makes things a bit more complicated than they need to be. We agree with this assessment. The `/` for positional-only arguments is a rarity in Python. The most common use is in cases where the implementation of a Python method or function is written in a language like C (or Rust) where the full flexibility of native Python isn't available.

We've seen three separate kinds of parameter possibilities:

- **Positional only**: These are handy in a few cases. If used, the positional-only parameters end with the special `/` parameter. See *PEP 570* for examples: `https://www.python.org/dev/peps/pep-0570`.
- **Either positional or keyword**: This is the case for most parameters. The order is designed to be helpful, and keywords can be used for clarification. More than three positional parameters invites confusion, so a long list of positional parameters isn't a great idea.
- **Keyword only**: After the special `*` parameter, the argument values **must** have a keyword supplied. This can be helpful to make rarely used options more visible. It can help to think of keywords as keys to a dictionary.

Choosing a pattern of parameter use often takes care of itself, depending on which values need to be supplied and which can be left at their defaults. For simple methods with a few argument values, positional parameters are more or less expected. Most mathematical operators have two operands. In some rare cases, as many as three are involved, but it requires tricky type-setting techniques to find a place to put the third operand. For this reason, when we say "a few," we often mean three or fewer. For complex methods with a lot of argument values, using keywords helps to clarify how things work.

Additional details on defaults

One thing to take note of with keyword arguments is that anything we provide as a default argument is evaluated exactly once when the function is first created. The argument is not evaluated again. This means we can't have dynamically generated default values. For example, the following code

won't behave quite as expected:

```
number = 5


def funky_function(x: int = number) -> str:
    return f"{x=}, {number=}"
```

The default value for the `x` parameter is the current value *when the function is defined*. We can see that behavior when we try to evaluate this with different values for the `number` variable:

```
>>> funky_function(42)
'x=42, number=5'

>>> number = 7
>>> funky_function()
'x=5, number=5'
```

The first evaluation looks like our expectation; the default value is the original value of the `number` variable. This is a coincidence. The second evaluation, after changing the global variable, `number`, shows that the function definition has a fixed value for the default — the default value is not re-evaluated.

To make this work, we'll often use `None` as a default value and assign the current value of a global variable within the body of the function:

```
def better_function(x: Optional[int] = None) -> str:
    if x is None:
        x = number
    return f"better: {x=}, {number=}"
```

This `better_function()` does not have a value for the `number` variable bound into the function definition. It uses the current value of a global `number` variable. Yes, this function is implicitly dependent on a global variable, and the docstring should explain that, ideally surrounded by flame emojis to make it clear to anyone reading it how the function's results may not be obviously idempotent.

It's common practice to use a name like `NUMBER` for a global variable; this makes it stand out. Note that we didn't use the `global` statement for a read-only global variable. The `global` statement is

only needed to update the value of a global variable.

A slightly more compact way to set a parameter value to an argument or a default looks like this:

```
def better_function_2(x: Optional[int] = None) -> str:
    x = number if x is None else x
    return f"better: {x=}, {number=}"
```

The `number if x is None else x` expression seems to make it clear that `x` will have the value of the global, `number`, or the argument value provided for `x`.

The "evaluation at definition time" can trip us up when working with mutable containers such as lists, sets, and dictionaries. It seems like a good design decision to make an empty list (or set or dictionary) as a default value for a parameter. We should never do this because the definition will create only one instance of the mutable object as part of the definition. This one object will be reused, demonstrated as follows:

```
def bad_default(tag: str, history: list[str] = []) -> list[str]:
    """A Very Bad Design (VBD™)."""
    history.append(tag)
    return history
```

This is very bad design. We can try to create a history list, `h`, and append things to it. This seems to work. Spoiler alert: the default object is one specific mutable `list` that's implicitly shared:

```
>>> h = bad_default("tag1")
>>> h = bad_default("tag2", h)
>>> h
['tag1', 'tag2']

>>> h2 = bad_default("tag21")
>>> h2 = bad_default("tag22", h2)
>>> h2
['tag1', 'tag2', 'tag21', 'tag22']
```

Whoops, that's not quite what we expected! When we tried to create a second history list, `h2`, it had values from the first history list, `h`. This is because the `h2` default value was the same (and only) default value used for `h`. We can see that using the `is` operator:

```
>>> h
['tag1', 'tag2', 'tag21', 'tag22']
>>> h is h2
True
```

Consequently, we never provide default values that are mutable objects. Lint-checking tools like **ruff** will warn us of using a mutable default value in a function definition. Instead of an empty list, set, or dictionary, we use `None`. It looks like this:

```
def good_default(tag: str, history: Optional[list[str]] = None) ->
list[str]:
    history = [] if history is None else history
    history.append(tag)
    return history
```

This will build a fresh, empty `list[str]` object if no parameter was supplied. This is the best way to work with default values that are also mutable objects.

Variable argument lists

Default values alone do not allow us all the flexibility we might want. One thing that makes Python really slick is the ability to write methods that accept an arbitrary number of positional or keyword arguments without explicitly naming them. We can also pass arbitrary lists and dictionaries into such functions. In other languages, these are sometimes called **variadic arguments**, **varargs**, and the programming techniques required to make them work are often hidden in macro definitions, In Python, all we need to do is be prepared to handle a tuple.

For example, we could write a function to accept a single URL or list of URLs and download the named web page resources. The idea is to avoid the confusing-looking overhead of a singleton list when we only want one page downloaded. We would like `get_pages("http:...")`. This seems nicer than `get_pages(["http:..."])` with a list of one item.

We do this by defining one positional parameter to receive all the argument values. This parameter has to be last (among the positional parameters), and we'll decorate it with a `*` prefix for the parameter name, as follows:

```
from urllib.parse import urlparse
from pathlib import Path


def get_pages(*links: str) -> None:
    for link in links:
        url = urlparse(link)
        name = "index.html" if url.path in ("", "/") else url.path
        target = Path(url.netloc.replace(".", "_")) / name
        print(f"Create {target} from {link!r}")
        # etc.
```

The `*` prefix to the `links` parameter says, *accept any number of argument values and put them all in a tuple named* `links`. If we supply only one argument, it will be a tuple with one element; if we supply no arguments, it will be an empty tuple. Thus, all these function calls are valid:

```
>>> get_pages()

>>> get_pages('https://www.archlinux.org')
Create www_archlinux_org...index.html from 'https://www.archlinux.org'

>>> get_pages('https://www.archlinux.org',
...        'https://dusty.phillips.codes',
...        'https://itmaybeahack.com'
... )
Create www_archlinux_org...index.html from 'https://www.archlinux.org'
Create dusty_phillips_codes...index.html from 'https://dusty.phillips.codes'
Create itmaybeahack_com...index.html from 'https://itmaybeahack.com'
```

Note that our type hint suggested that all of the positional argument values are of the same type, `str`. This is a widespread expectation: the variable parameters feature is little more than syntactic sugar, saving us from writing a dumb-looking list with only one item for those cases where there's only a single URL.

We can also accept arbitrary keyword arguments. These arrive in the function as a dictionary. They are specified with a double-asterisk prefix on the parameter name – `**kwargs`, for example, in the function declaration. This form is commonly used in configuration setups where there are a lot of optional values. The following class allows us to specify a set of options with default values:

```
from typing import Any


class Options(dict[str, Any]):
    default_options: dict[str, Any] = {
        "port": 21,
        "host": "localhost",
        "username": None,
        "password": None,
        "debug": False,
    }

    def __init__(self, **kwargs: Any) -> None:
        super().__init__({**self.default_options, **kwargs})
```

This class leverages a feature of the `__init__()` method for the `dict` class. We have a dictionary of default options, with the boring name of `default_options`, defined as part of the class. The `__init__()` method accepts a dictionary of keyword parameters. It computes a new dictionary from the default values, applying the argument values to create a dictionary that has both. Then, this dictionary is used to initialize the object. Note that the order of the expressions in `{**self.default_options, **kwargs}` matters. We want the supplied keys and values to replace the default keys and values.

Also note that tools to create documentation from the code (like **Sphinx** or **epydoc**) won't uncover the details of the default values. Don't astonish your users: be sure the defaults are described in the documentation.

Here's a session demonstrating the class in action:

```
>>> options = Options(username="dusty", password="Hunter2",
...     debug=True)
>>> options['debug']
True
>>> options['port']
21
>>> options['username']
'dusty'
```

We're able to access our `options` instance using dictionary indexing syntax. The `Options` dictionary includes both default values and the ones we set using keyword arguments.

Note that the parent class is `dict[str, Any]`; the class for a generic dictionary limited to strings for keys. When we initialize the `default_options` object, we provide a matching hint for this variable.

In the preceding example, it's possible to pass arbitrary keyword arguments to the `Options` initializer. This could include keys that don't exist in the default dictionary. This can be good when adding new features to an application. This can be bad when debugging a spelling mistake. Providing the "Port" option instead of the "port" option will lead to two similar-looking options where only one should have existed.

One way to limit the risk of spelling mistakes is to write our own version of a dictionary update to limit changes to replacing existing keys. This can prevent misspellings from creating problems. The solution is interesting and we'll leave it as an exercise for you.

Keyword arguments are also very useful when we need to accept arbitrary arguments to pass to a second function, but we don't know what those arguments will be. We saw this in action in *Chapter 3*, when we were building support for multiple inheritance.

We can, of course, combine the variable argument and variable keyword argument syntax in one function call, and we can use normal positional and default arguments as well. It's hard to find a use for **all** of these features in one place. While the following example is contrived, the syntax is the focus, and this example demonstrates the four methods of providing values for parameters in action:

```
import contextlib
from typing import TextIO, Any
from pathlib import Path


def doctest_everything(
    output: TextIO,
    *directories: Path,
    verbose: bool = False,
    **stems: str,
) -> None:
    if verbose:
        log = print
    else:
        def log(*args: Any, **kwars: Any) -> None:
            pass
```

```
    with contextlib.redirect_stdout(output):
        for directory in directories:
            log(f"Searching {directory}")
            for dirpath, dirnames, filenames in directory.walk():
                remove_excluded(dirnames)

                for name in filenames:
                    if not name.endswith('.py'):
                        continue
                    path = (dirpath / name).relative_to(Path.cwd())
                    log(
                        f"File {path}, "
                        f"{path.stem=}"
                    )
                    options = stems.get(path.stem, "")
                    if options.upper() == "SKIP":
                        log("Skipped")
                        continue
                    doctest_opts = (options.upper().split(",") if options
                    else [])
                    r = run_test(path, doctest_opts )
                    if r.returncode:
                        log(r.stderr)
```

This example has an extraordinary number of parameters. It processes an arbitrary list of directory paths where it runs the **doctest** tool on the files found in those directories. Let's look at each parameter definition in detail:

- The first parameter, `output`, is an open file to which output will be written.
- The `directories` parameter will be given all of the remaining positional (also called non-keyword) arguments. These should all be `Path()` objects.
- The `verbose` parameter is a keyword only because it appears after a `*name` parameter that collects all positional values. It tells the function whether to print information about each file processed.
- Finally, all other keywords create a dictionary called `stems`. These need to be the stem of a filename; the associated value will be a special processing concern. Note that four names — `output`, `directories`, `verbose`, and `stems` — are effectively special filenames that can't be given special processing; these are ordinary parameter names.

> Specifically, if a file stem has a value of `"SKIP"`, the file won't be tested. Any other value is used to create the option flags for the **doctest** tool. Providing a `some_module="ellipsis"` parameter, for example, means the the `ELLIPSIS` option will be given to the **doctest** tool for a module with a name of `some_module.py`.

When needed, this function will create an inner helper function, `log()`. This is either the `print()` function or a placeholder that silences the output. This lets us use `log("message")` everywhere, knowing it can be silenced at runtime.

Within the body of the function, the `with` statement redirects all output normally sent to `sys.stdout` to the desired file. This lets us collect a single log from `print()` functions without having to put `file=` options everywhere. The `for` statement iterates through all the positional argument values collected into the `directories` parameter. Each directory is examined with the `walk()` method to locate all files and subdirectories.

A common part of the top-down walk processing is to remove directory names that should be excluded and avoid the overhead of examining the files inside them. A `remove_excluded()` function — which isn't shown — removes names like `.tox` and `.venv` that should not be examined.

A file's *stem* is the name without any path or suffix. So `ch_03/docs/examples.py` has a stem of `examples`. If the stem is used as a keyword argument, the value of that argument provides additional details of what to do for files with that specific stem. As noted above, the keyword argument `examples='SKIP'` will populate the `**stems` dictionary, and any file with a stem of `examples` will be skipped. (And yes, this design won't work well when multiple files have the same stem in different directories.)

We've delegated the details of running the test tool to a function, `run_test()`. This function is expected to build an environment and command line. It uses the `subprocess.run()` function to invoke the **doctest** tool.

In common cases, the `doctest_everything()` function could be called as follows:

```
doctest_everything(
    sys.stdout,
    Path.cwd() / "ch_02",
    Path.cwd() / "ch_03",
)
```

This command would locate all the `*.py` files in these two directories and run doctest for each file.

The output would appear on the console because we redirected `sys.stdout` back to `sys.stdout`. Very little output would be produced because the `verbose` parameter would have a default value of `False`.

If we want to collect detailed output, we can call it with the help of the following command:

```
doctest_log = Path("doctest.log")
with doctest_log.open("w") as log:
    doctest_everything(
        log,
        Path.cwd() / "ch_04",
        Path.cwd() / "ch_05",
        verbose=True,
        colors="SKIP",
    )
```

This tests files in the given directories and tells us what it's doing. The output is captured in a file instead of scrolling past in the terminal window.

Notice that it is impossible to specify `verbose` as a positional argument; we must pass this as a keyword argument. Otherwise, Python would think the value of `True` or `False` was yet another value for the `*directories` list. A type-hint checking tool will spot the problem. At runtime, the problem will surface when attempting to use the boolean object as if it were a `Path` object.

Also note that any files with the stem of `colors` will be skipped. Consider how it might look if there are a lot of files requiring special processing. It might require numerous keyword arguments, as follows:

```
doctest_everything(
    sys.stdout,
    Path.cwd() / "ch_02",
    Path.cwd() / "ch_03",
    # exceptions...
    first_class="ELLIPSIS",
    test_ecommerce="SKIP",
    vendors="SKIP",
    __init__="SKIP",
    products="SKIP",
    main="SKIP",
    square="SKIP",
    stripe="SKIP",
```

```
)
```

This will test two directories, but won't display any output, since we didn't specify `verbose`. This will apply the `doctest -/-ellipsis` option to any file with a stem of `first_class`. Similarly, any large number of files are all skipped.

Because we can provide any name we choose, and they will all be collected into the value of the `stems` parameter, we can make use of this flexibility to match names of files in the directory structures. There are, of course, a number of limitations on Python identifiers that don't match operating system filenames, making this less than perfect. It does, however, show the amazing flexibility of Python function arguments.

Unpacking arguments

There's one more nifty trick involving positional and keyword parameters. We've used it in some of our previous examples, but it's never too late for an explanation. Given a list or dictionary of values, we can pass a sequence of values into a function as if they were normal positional or keyword arguments. Have a look at this code:

```
def show_args(arg1: Any, arg2: Any, arg3: Any="THREE") -> str:
    return f"{arg1=}, {arg2=}, {arg3=}"
```

The function accepts three parameters, one of which has a default value. But when we have a list of three argument values, we can use the `*` operator inside a function call to unpack a sequence of values so they are applied to the three arguments.

Here's what it looks like when we run it with `*some_args` to provide a three-element iterable:

```
>>> some_args = range(3)
>>> show_args(*some_args)
'arg1=0, arg2=1, arg3=2'
```

The value of `*some_args` has to match the positional parameter definition. Because there's a default value for `arg3`, making it optional, we can provide two or three values to this function.

If we have a dictionary of arguments, we can use the `**` syntax to unpack a dictionary to supply argument values for keyword parameters. It looks like this:

```
>>> more_args = {
... "arg1": "ONE",
... "arg2": "TWO"}
>>> show_args(**more_args)
"arg1='ONE', arg2='TWO', arg3='THREE'"
```

This is often useful when mapping information that has been collected from user input or from an outside source (for example, an internet page or a text file) and needs to be provided to a function or method call. Rather than decompose an external source of data into individual keyword parameters, we simply provide the keyword parameters from the dictionary keys. An expression like `show_args(arg1=more_args['arg1'], arg2=more_args['arg2'])` seems an error-prone way to match a parameter name with the dictionary key. Does each argument name properly match the key used to fetch it from the `more_args` dictionary?

This unpacking syntax can be used in some areas outside of function calls, too. Consider a class `__init__()` method that looks like this:

```
super().__init__({**self.default_options, **kwargs})
```

The expression `{**self.default_options, **kwargs}` will merge dictionaries through a two-step process:

1. Unpack each dictionary into keyword arguments.
2. Assemble a final dictionary from two collections of keyword arguments.

Because the dictionaries are unpacked in order from left to right, the resulting dictionary will contain all the default options, with any of the `kwarg` options replacing some of the keys. Here's an example:

```
>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 11, 'c': 3}
>>> z = {**x, **y}
>>> z
{'a': 1, 'b': 11, 'c': 3}
```

This dictionary unpacking is a handy consequence of the way the ** operator transforms a dictionary into named parameters for a function call.

After looking at sophisticated ways we can provide argument values to functions, we need to look at functions a little more broadly. Python considers functions as one kind of "callable" object. This means functions are objects, and higher-order functions can accept functions as argument values and return functions as results.

Functions are objects too

There are numerous situations where we'd like to pass around a small object that is simply called to perform an action. In essence, we'd like an object that is also a callable function. This is most frequently done in event-driven programming, such as graphical toolkits or asynchronous servers; we'll see some design patterns that use it in *Chapter 11* and *Chapter 12*.

In Python, we don't need to wrap such methods in a class definition because functions are already objects! We can set attributes on functions (though this isn't a common activity), and we can pass them around to be called at a later date. They even have a few special properties that can be accessed directly.

Here's yet another contrived example, sometimes used as an interview question:

```
from typing import Callable


def fizz(x: int) -> bool:
    return x % 3 == 0

def buzz(x: int) -> bool:
    return x % 5 == 0

def name_or_number(
        number: int,
        *tests: Callable[[int], bool]
) -> str:
    for t in tests:
        if t(number):
            return t.__name__
    return str(number)
```

The `fizz()` and `buzz()` functions check to see whether their parameter, `x`, is an exact multiple of another number. This relies on the definition of the modulo operator: if x is a multiple of 3, then 3 divides x with no remainder. Sometimes they say $x \equiv 0 \mod 3$ in math books. In Python, we say `x`

`% 3 == 0`.

The `name_or_number()` function uses any number of test functions, provided as the `tests` parameter value. The `for` statement assigns each function in the `tests` collection to a variable, `t`, then evaluates the variable with the number parameter's value. If the exact division function's value is true, then the result is the function's name. (As we'll see later, there's a subtle bug here.)

Here's how this function looks when we apply it to a number and another function:

```
>>> name_or_number(1, fizz)
'1'
>>> name_or_number(3, fizz)
'fizz'
>>> name_or_number(5, fizz)
'5'
```

In each case, the value of the `tests` parameter is `(fizz,)` a tuple that contains only the `fizz` function. The `name_or_number()` function evaluates `t(number)`, where `t` is the `fizz()` function. When `fizz(number)` is true, the value returned is the value of the function's `__name__` attribute — the `'fizz'` string. Function names are available at runtime as an attribute of the function.

What if we provide multiple functions? Each is applied to the number until one is true:

```
>>> name_or_number(5, fizz, buzz)
'buzz'
```

This is, by the way, not completely correct. What should happen for a number like 15? Is it `fizz` or `buzz` or both? Because the number 15 is both, some work needs to be done in the `name_or_number()` function to collect **all** the names of all the true functions. That sounds like it would make a good exercise.

We can add to our list of special functions. We might define `bazz()` to be true for multiples of seven. This, too, sounds like a good exercise.

If we run this code, we can see that we were able to pass two different functions into our `name_or_number()` function, and get different output for each one:

```
>>> for i in range(1, 11):
...     print(name_or_number(i, fizz, buzz))
```

```
1
2
fizz
4
buzz
fizz
7
8
fizz
buzz
```

We could apply our functions to an argument value using `t(number)`. We were able to get the value of the function's `__name__` attribute using `t.__name__`.

Function objects and callbacks

The fact that functions are top-level objects is most often used to pass them around to be executed at a later date, for example, when a certain condition has been satisfied. Callback functions are common as part of building a user interface: when the user clicks on something, the framework can call a function so the application code can create a visual response. For very long-running tasks, like file transfers, it is often helpful for the transfer library to call back to the application with status on the number of bytes transferred so far — this makes it possible to display status thermometers to show the progress of the transfer.

Let's build an event-driven timer using callbacks so that things will happen at scheduled intervals. This can be handy for an **IoT** (**Internet of Things**) application built on a small CircuitPython or MicroPython device. We'll break this down into two parts: a task, and a scheduler that executes the function object stored in the task:

```
from collections.abc import Callable
from dataclasses import dataclass, field
import heapq
import time

type Callback = Callable[[int], None]


@dataclass(frozen=True, order=True)
class Task:
```

```
    scheduled: int
    callback: Callback = field(compare=False)
    delay: int = field(default=0, compare=False)
    limit: int = field(default=1, compare=False)

    def repeat(self, current_time: int) -> "Task | None":
```

```
        if self.delay > 0 and self.limit > 2:
            return Task(
                current_time + self.delay,
                self.callback,
                self.delay,
                self.limit - 1,
            )
        elif self.delay > 0 and self.limit == 2:
            return Task(
                current_time + self.delay,
                self.callback,
            )
        else:
            return None
```

The `Task` class definition has two mandatory fields and two optional fields. The mandatory fields, `scheduled` and `callback`, provide a scheduled time to do something and a callback function, the thing to be done at the scheduled time. The scheduled time has an `int` type hint; the time module can use floating-point time, for super-accurate operations. We're going to ignore these details. Also, the **mypy** tool is well aware that integers can be coerced to floating-point numbers, so we don't have to be super-fussy-precise about numeric types.

The callback has a hint of `Callable[[int], None]`. This summarizes what the function definition should look like. To match this type hint, any callback function we write should look like `def some_name(an_arg: int) -> None:`. If it doesn't match, **mypy** will alert us to the potential mismatch between our callback function definition and the contract specified by the type hint.

The `repeat()` method can return a task for those tasks that might repeat. It computes a new time for the task, provides the reference to the original function object, and may provide a subsequent delay and a changed limit. The changed limit will count the number of repetitions toward zero, giving us a defined upper limit on processing; it's always nice to be sure that iteration will terminate.

Here's the overall `Scheduler` class that uses these `Task` objects and their associated callback functions:

```
class Scheduler:
```

```
    def __init__(self) -> None:
        self.tasks: list[Task] = []

    def enter(
        self,
        after: int,
        task: Callback,
        delay: int = 0,
        limit: int = 1,
    ) -> None:
        new_task = Task(after, task, delay, limit)
        heapq.heappush(self.tasks, new_task)

    def run(self) -> None:
        current_time = 0
        while self.tasks:
            next_task = heapq.heappop(self.tasks)
            if (delay := next_task.scheduled - current_time) > 0:
                time.sleep(delay)
            current_time = next_task.scheduled
            next_task.callback(current_time)
            if again := next_task.repeat(current_time):
                heapq.heappush(self.tasks, again)
```

The central feature of the `Scheduler` class is a heap queue, a list of `Task` objects kept in a specific order. We mentioned the heap queue in the *Three types of queues* section of *Chapter 8*, noting that the priority ordering made it inappropriate for that use case. Here, however, the heap data structure makes use of the flexibility of a list to keep items in order without the overhead of a complete sort of the entire list. In this case, we want to keep items in order by the time they're required to be executed: "first things first" order. When we push something to a heap queue, it's inserted so the time order will be maintained. When we pop the next thing from the queue, the heap may be adjusted to keep the first things at the front of the queue.

The `Scheduler` class provides an `enter()` method to add a new task to the queue. This method accepts a `delay` parameter representing the interval to wait before executing the callback task, and

the `task` function itself, a function to be executed at the correct time. This `task` function should fit the type hint of `Callback`, defined previously.

There are no runtime checks to ensure the callback function really does meet the type hint. It's only checked by **mypy**. More importantly, the `after`, `delay`, and `limit` parameters should have some validation checks. For example, a negative value of `after` or `delay` should raise a `ValueError` exception. There's a special method name, `__post_init__()`, that a dataclass can use for validation. This is invoked after `__init__()` and can be used for other initialization, pre-computing derived values, or validating that the combination of values is sensible.

The `run()` method removes items from the queue in order by the time they're supposed to be performed. If we're at (or past) the required time, then the value assigned to the `delay` variable by the `:=` operator will be zero or negative, and we don't need to wait; we can perform the callback immediately. If this runs before the required time, then it can sleep until the time arrives.

At the appointed time, we'll update our current time in the `current_time` variable. We'll call the callback function provided in the `Task` object. Once that's finished, then we'll see if the `Task` object's `repeat()` method will provide another repeat task in the queue.

The important things to note here are the lines that touch callback functions. The function is passed around like any other object and the `Scheduler` and `Task` classes never know or care what the original name of the function is or where it was defined. When it's time to call the function, the `Scheduler` simply evaluates the function with `new_task.callback(current_time)`.

Here's a set of callback functions that test the `Scheduler` class:

```
import datetime


def format_time(message: str) -> None:
    now = datetime.datetime.now()
    print(f"{now:%I:%M:%S}: {message}")


def one(timer: float) -> None:
    format_time("Called One")


def two(timer: float) -> None:
    format_time("Called Two")
```

```
def three(timer: float) -> None:
    format_time("Called Three")


class Repeater:
    def __init__(self) -> None:
        self.count = 0

    def four(self, timer: float) -> None:
        self.count += 1
        format_time(f"Called Four: {self.count}")
```

These functions all meet the definition of the `Callback` type hint, so they'll work nicely. The `Repeater` class definition has a method, `four()`, that meets the definition. That means an instance of `Repeater` can also be used.

We've defined a handy utility function, `format_time()`, to write common messages. It uses the format string syntax to add the current time to the message. The three small callback functions output the current time and a short message telling us which of the callbacks has been fired.

Here's an example of creating a scheduler and loading it up with callback functions:

```
if __name__ == "__main__":
    s = Scheduler()
    s.enter(1, one)
    s.enter(2, one)
    s.enter(2, two)
    s.enter(4, two)
    s.enter(3, three)
    s.enter(6, three)
    repeater = Repeater()
    s.enter(5, repeater.four, delay=1, limit=5)
```

This example allows us to see how multiple callbacks interact with the timer.

The `Repeater` class demonstrates that methods can be used as callbacks too, since they are really functions that happen to be bound to an object. Using a method of an instance of the `Repeater` class is a function like any other.

The output shows that events are run in the expected order:

```
01:44:35: Called One
01:44:36: Called Two
01:44:36: Called One
01:44:37: Called Three
01:44:38: Called Two
01:44:39: Called Four: 1
01:44:40: Called Three
01:44:40: Called Four: 2
01:44:41: Called Four: 3
01:44:42: Called Four: 4
01:44:43: Called Four: 5
```

Note that some events have the same scheduled runtime. Scheduled after 2 seconds, for example, both callback functions `one()` and `two()` are defined. They both ran at 01:44:36. There's no rule to decide how to resolve the tie between these two functions. The scheduler's algorithm is to pop an item from the heap queue, execute the callback function, then pop another item from the heap queue; if it has the same execution time, then evaluate the next callback function. Which of the two callbacks is performed first and which is done second is an implementation detail of the heap queue. If order matters to your application, you'll need an additional attribute to distinguish among items scheduled at the same time; a priority number is often used for this.

Because Python is a dynamic language, the contents of a class are not fixed. There are some more advanced programming techniques available to us. In the next section, we'll look at changing the methods of a class.

Using functions to patch a class

One of the things we noted in the previous example was that tools like **mypy** assume the `Callable` attribute, `callback`, was a method of the `Task` class. This leads to a potentially confusing **mypy** error message, `Invalid self argument "Task" to attribute function "callback" with type "Callable[[int], None]"`. In the previous example, the callable attribute was emphatically not a method.

The presence of the confusion means that a callable attribute can be treated as a method of a class. Since we can generally supply extra methods to a class, it means we can patch in additional methods at runtime.

Does it mean we **should** do this? It's perhaps a bad idea, except in a very special situation.

It is possible to add or change a function to an instantiated object, demonstrated as follows. First we'll define a class, `A`, with a method, `show_something()`:

```
class A:
    def show_something(self) -> None:
        print("My class is A")
```

Here's how it looks when we create an instance of this class and execute a method:

```
>>> a_object = A()
>>> a_object.show_something()
My class is A
```

This looks like what we'd expect. We invoke the method on an instance of the class and see the results of the `print()` function. Now, let's patch this object, replacing the `show_something()` method:

```
def patched_show_something() -> None:
    print("My class is NOT A")
```

Here's what it looks like when we patch an object to be able to use this function:

```
>>> a_object = A()
>>> a_object.show_something = patched_show_something
>>> a_object.show_something()
My class is NOT A
```

We've patched the `a_object` object by introducing an attribute value that's a callable function. When we use `a_object.show_something()`, the rule is to look in local attributes first, then look in class attributes. Because of this, we've used a callable attribute to create a localized patch to this instance of the `A` class.

We can create another instance of the class, unpatched, and see that it's still using the class-level method:

```
>>> b_object = A()
>>> b_object.show_something()
My class is A
```

If we can patch an object, you'd think we can also patch the class. We can. It is possible to replace methods on classes as well as replacing methods in objects. If we change the class, we have to account for the `self` argument that will be implicitly provided to methods defined in the class.

It's very important to note that patching a class will change the method for all instances of that object, even ones that have already been instantiated. Obviously, replacing methods like this can be both dangerous and confusing to maintain. Somebody reading the code will see that a method has been called and look up that method on the original class. But the method on the original class is not the one that was called. Figuring out what really happened can become a tricky, frustrating debugging session.

There's a cardinal assumption that needs to underpin everything we write. It's a kind of contract that is essential to understanding how software works:

The code people see in a module file must be the code that is running.

Patching code outside the `class` statement is perfectly awful. With one exception — unit testing.

Breaking this assumption by patching classes or objects will really confuse people. Our previous example showed an instance of class `A` that had a method named `show_something()` with behavior clearly different to the definition for class `A`. That's going to be lead people to distrust your application software.

This technique does have its uses though. Often, replacing or adding methods at runtime (called **monkey patching**) is used in unit testing. If testing a client-server application, we may not want to actually connect to the server while testing the client; this may result in accidental transfers of funds or embarrassing test emails being sent to real people.

Instead, we can set up our test code to replace some of the key methods on the object that sends requests to the server so that it only records that the methods have been called. We'll cover this in detail in *Chapter 13*. Outside the narrow realm of testing, monkey patching is generally a sign of bad design.

In the case of our class in this example, a subclass of `A` with a distinct implementation of the `show_something()` method would make things much more clear than a patched method.

We can use class definitions to create objects that are usable as if they were functions. This gives us another path toward using small, separate functions to build applications.

While Python lets us build classes dynamically, we don't encourage its use in general. These kinds of patches are appropriate for unit testing.

A much more useful technique is creating a callable object. This lets us define functions that are configurable at runtime.

Callable objects

Just as functions are objects that can have attributes set on them, it is possible to create an object that can be called as though it were a function. Any object can be made callable by giving it a `__call__()` method. This method accepts the parameters for the function call and returns the value. Let's make our `Repeater` class, from the timer example, a little easier to use by making it a callable, as follows:

```
class Repeater_2:
    def __init__(self) -> None:
        self.count = 0

    def __call__(self, timer: float) -> None:
        self.count += 1
        format_time(f"Called Four: {self.count}")
```

This example isn't much different from the earlier class; all we did was change the name of the `repeater` function to `__call__()` and pass the object itself as a callable. How does this work? We can do the following interactively to create an example callable object:

```
>>> rpt2 = Repeater_2()
```

At this point, we've created a callable object, `rpt2()`. When we evaluate something like `rpt2(1)`, Python will evaluate `rpt2.__call__(1)` for us because there's a `__call__()` method defined for the object's class. It looks like this:

```
>>> from unittest.mock import Mock, patch
>>> expected_date = datetime.datetime(2019, 10, 26, 11, 12, 13)
>>> mock_method = Mock(return_value=expected_date)
```

```
>>> rpt2 = Repeater_2()
>>> with patch('datetime.datetime', now=mock_method):
...     rpt2(42)
...     rpt2(43)
...     rpt2(44)
11:12:13: Called Four: 1
11:12:13: Called Four: 2
11:12:13: Called Four: 3
```

As a teaser for *Chapter 13*, we've included a little of the test technology in this example. Each of the callback functions uses a `format_time` function to provide a timestamp along with a message. We've patched the `datetime.datetime` class so the `now()` method will return a known date-time object. This makes it easy to write a doctest example, because the time that's returned is always the expected date and time.

Each call to the `rpt2` object works as if it were a call to a function. Plus, as a bonus, the callable object increments a counter that's entirely encapsulated by the class definition.

Here's an example of using this variation on the `Repeater_2` class definition with a `Scheduler` object:

```
    s2 = Scheduler()
    s2.enter(5, Repeater_2(), delay=1, limit=5)
    s2.run()
```

Note that, when we make the `enter()` call, we pass as an argument the `Repeater_2()` object. Those two parentheses create a new instance of the class. The instance that is created has the `__call__()` method, which can be used by the `Scheduler` instance. When working with callable objects, it's essential to create an instance of a class; it's the object that's callable, not the class.

At this point, we've seen two different kinds of callable objects:

1. Python's functions, built with the `def` statement.
2. Callable objects. These are instances of a class with the `__call__()` method defined.

Generally, the simple `def` statement is all we need. Callable objects, however, can do something an ordinary function can't do. Our `Repeater_2` class counts the number of times it was used. An ordinary function is stateless. A callable object can be stateful. This needs to be used with some care, but some algorithms can have a dramatic performance improvement from saving results in a

cache, and a callable object is a great way to save results from a function so they don't need to be recomputed.

Recall

We've touched on a number of ways that object-oriented and functional programming techniques are part of Python:

- Python built-in functions provide access to special methods that can be implemented by a wide variety of classes. Almost all classes, most of them utterly unrelated, provide an implementation for `__str__()` and `__repr__()` methods, which can be used by the built-in `str()` and `repr()` functions. There are many functions like this where a function is provided to access implementations that cut across class boundaries.
- Some object-oriented languages rely on "method overloading" — a single name can have multiple implementations with different combinations of parameters. Python provides an alternative, where one method name can have optional, mandatory, position-only, and keyword-only parameters. This provides tremendous flexibility.
- Functions are objects and can be used in ways that other objects are used. We can provide them as argument values; we can return them from functions. A function has attributes, also.

Exercises

You've probably used many of the basic built-in functions before now. We covered several of them, but didn't go into a great deal of detail. Play with `enumerate`, `zip`, `reversed`, `any`, and `all` until you know you'll remember to use them when they are the right tool for the job. The `enumerate` function is especially important because not using it results in some pretty ugly `while` statements.

Also explore some applications that pass functions around as callable objects, as well as using the `__call__()` method to make your own objects callable. You can get the same effect by attaching attributes to functions or by creating a `__call__()` method on an object. In which case would you use one syntax, and when would it be more suitable to use the other?

The relationship between arguments, keyword arguments, variable arguments, and variable keyword arguments can be a bit confusing. We saw how painfully they can interact when we covered multiple inheritance. Devise some other examples to see how they can work well together, as well as to understand when they don't.

The `Options` example for using `**kwargs` has a potential problem. The `update()` method inherited from the `dict` class will add or replace keys. What if we only want to replace key values? We'd have to write our own version of `update()` that will update existing keys and raise a `ValueError` exception when a new key is provided.

The `name_or_number()` function example has a blatant bug. It is not completely correct. For a number 15, it will not report both `"fizz"` and `"buzz"`. Fix the `name_or_number()` function to collect all the names of all the true functions. A good exercise.

The `name_or_number()` function example has two test functions, `fizz()`, and `buzz()`. We need an additional function, `bazz()`, to be true for multiples of seven. Write the function and be sure it works with the `name_or_number()` function. Be sure that the number 105 is handled correctly.

It's helpful to review the previous case studies and combine them into a more complete application. The chapter case studies tend to focus on details, avoiding the overall integration of a more complete application. We've left the integration as work for you to allow you to dig more deeply into the design.

Summary

We covered a grab bag of topics in this chapter. Each represented an important non-object-oriented feature that is popular in Python. Just because we can use object-oriented principles does not always mean we should!

However, we also saw that Python typically implements such features by providing a syntax shortcut to traditional object-oriented syntax. Knowing the object-oriented principles underlying these tools allows us to use them more effectively in our own classes.

We discussed a series of built-in functions. There are a whole bunch of different syntaxes available to us when calling functions with arguments, keyword arguments, and variable argument lists. Context managers are useful for the common pattern of sandwiching a piece of code between two method calls. Even functions are objects, and, conversely, any normal object can be made callable.

In the next chapter, we'll look at a design pattern that is so fundamental to Python programming that it has been given special syntax support: the iterator pattern.

Join our community Discord space

Join our Python Discord workspace to discuss and know more about the book: `https://packt.link/dHrHU`

10

The Iterator Pattern

We've discussed how many of Python's built-ins and idioms seem, at first blush, to fly in the face of object-oriented principles, but are actually providing access to the objects with a functional syntax. In this chapter, we'll discuss how the `for` statement is actually a lightweight wrapper around a set of object-oriented design patterns. We'll also see a variety of extensions to this syntax that automatically create even more types of object. We will cover the following topics:

- What design patterns are
- The iterator protocol — one of the most powerful design patterns
- List, set, and dictionary comprehensions
- Generator functions, and how they build on other patterns

We'll start with an overview of what design patterns are and why they're so important.

Design patterns in brief

When engineers and architects decide to build a bridge, a tower, or a building, they follow certain principles to ensure structural integrity. There are various possible designs for bridges (suspension and cantilever, for example), but if the engineer doesn't use one of the standard designs, and doesn't have a brilliant new design, it is likely the bridge they design will collapse.

Design patterns are an attempt to apply this same formal definition for correctly designed structures to software engineering. There are many different design patterns for solving different general problems. Design patterns are applied to solve a common problem faced by developers in some specific situation. What's central to a pattern is that it is reused. This means it may be used in a unique context perhaps not obvious from the way the pattern is often described. One clever solution is a good idea. Two similar solutions might be a coincidence. Three or more reuses of an idea and it starts to look like a repeating pattern.

Knowing common object-oriented design patterns and choosing to use them in our software does not, however, guarantee that we are creating a *correct* solution. In 1907, the Québec Bridge (to this day the longest cantilever bridge in the world, just short of a kilometer long) collapsed before construction was completed, because the engineers who designed it grossly underestimated the weight of the steel used to construct it. Similarly, in software development, we may incorrectly choose or apply a design pattern, and create software that *collapses* under normal operating situations or when stressed beyond its original design limits.

Any one design pattern proposes a set of objects interacting in a specific way to solve a general problem. The job of the programmer is to recognize when they are facing a specific version of such a problem, then to choose and adapt the general pattern to their precise needs.

In this chapter, we'll look deeply at the iterator design pattern. This pattern is so powerful and pervasive that the Python developers have provided multiple syntaxes to access the object-oriented principles underlying the pattern. We will be covering other design patterns in the next two chapters. Some of them have language support and some don't, but none of them seem to be so intrinsically a part of the Python coder's daily life as the iterator pattern.

Iterators

We can think of an **iterator** as an object with a `next()` method and a `done()` method; the latter returns `True` if there are no items left in the sequence. In a programming language without built-in support for iterators, the iterator would be used like this:

```
iterator = some_collection.iterator()
while not iterator.done():
    item = iterator.next()
    # do something with the item from some_collection...
```

This example omits the class for `some_collection`, which needs to implement an `iterator()` method to return a stateful object to iterate through items in the collection. It also omits the class for an iterator, which is initialized with the value of `some_collection` and then handles the `next()` and `done()` methods to return each item.

In Python, iteration is available across many language features, so the method of an iterator to return the next item gets a special name, `__next__`. This method can be accessed using the `next(iterator)` built-in function. Rather than a `done()` method, Python's iterator protocol raises the `StopIteration` exception to notify the client that the iterator has completed and there is no more data. Finally, we have the much more readable `for item in iterator:` syntax to actually access items in an iterator instead of messing around with a `while` statement, and the explicit creation of the iterator instance. Let's look at each these features in more detail.

The iterator protocol

The `Iterator` abstract base class, in the `collections.abc` module, defines the protocol for iteration. Any iterator must offer the `__next__()` method to provide the next item in a collection.

Additionally, any `Collection` class definition must be `Iterable`. To be `Iterable`, the protocol needs to provide an `__iter__()` method; this method creates an `Iterator` object for the items in the collection.

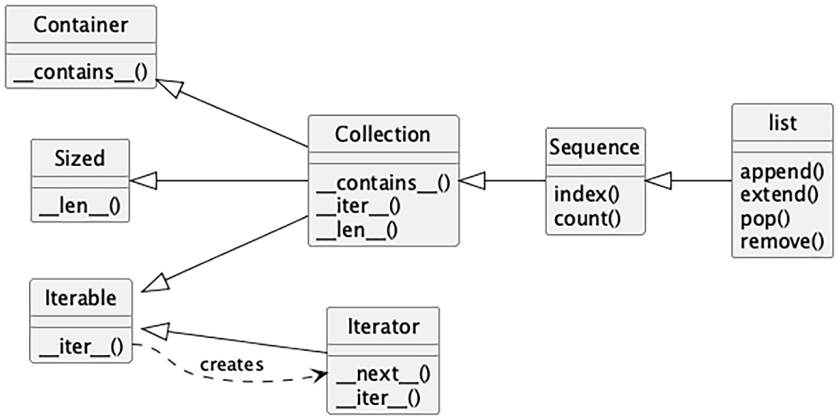


Figure 10.1: The abstractions for Iterable

As mentioned, an `Iterator` class must define a `__next__()` method. The `for` statement (and other features that support iteration) use this to get each element from the sequence. In addition, every

`Iterator` class must also fulfill the `Iterable` interface. This means an `Iterator` will also provide an `__iter__()` method and return itself as the result.

This might sound a bit confusing, so have a look at the following example. Note that this is a very verbose way to solve this problem. It's not optimal; it's exploratory. It demonstrates the details of iteration and the two protocols in question. Later in this chapter, we'll look at several more readable ways to get this effect:

```
from typing import Iterable, Iterator


class CapitalIterable(Iterable[str]):
    def __init__(self, string: str) -> None:
        self.string = string

    def __iter__(self) -> Iterator[str]:
        return CapitalIterator(self.string)


class CapitalIterator(Iterator[str]):
    def __init__(self, string: str) -> None:
        self.words = [w.capitalize() for w in string.split()]
        self.index = 0

    def __next__(self) -> str:
        if self.index == len(self.words):
            raise StopIteration()

        word = self.words[self.index]
        self.index += 1
        return word
```

This example of the iterator protocol defines a `CapitalIterable` class whose job is to loop over each of the words in a string and output them with the first letter capitalized. We formalized this by using the `Iterable[str]` type hint as a superclass to make it clear what our intention is. Most of the work of this iterable class is delegated to the `CapitalIterator` implementation. One way to interact with this iterator is as follows:

```
>>> iterable = CapitalIterable('the quick brown fox jumps over the lazy
dog')
```

```
>>> iterator = iter(iterable)
>>> while True:
...     try:
...         print(next(iterator))
...     except StopIteration:
...         break
...
The
Quick
Brown
Fox
Jumps
Over
The
Lazy
Dog
```

This example first constructs an iterable, assigning it to a variable with the boringly obvious name of `iterable`. It then retrieves a `CapitalIterator` instance from the `iterable` object. The distinction may need explanation; the `iterable` object is a collection of elements that can be iterated over. The `iterator` object, on the other hand, represents a specific state during the process of visiting the items in an iterable; some of the items have been consumed and some have not. For a collection that's a sequence, the state is often an index value that gets incremented. Two different iterator objects might be at different places in the list of words, but each iterator has it's own private state and can mark only one place.

Each time `next()` is called on the iterator, it does two things: it updates its internal state to point to the next item, and it also returns another item from the iterable, in order. The state change for visiting items is entirely encapsulated in the iterator. Eventually, the iterator will be exhausted (won't have any more elements to return from the iterable), in which case a `StopIteration` exception is raised. This is generally used to exit from any containing `while` or `for` statement.

Python has a simpler syntax for constructing an iterator from an iterable:

```
>>> for i in iterable:
...     print(i)
...
The
Quick
```

```
Brown
Fox
Jumps
Over
The
Lazy
Dog
```

As you can see, the `for` statement, in spite of not looking remotely object-oriented, is actually a shortcut to some fundamentally object-oriented design principles. Keep this in mind as we discuss **comprehensions**, as they, too, appear to be the polar opposite of an object-oriented tool. Yet, they use the same iteration protocol as `for` statements and are another kind of shortcut.

The number of iterable collections in Python is large. We're not surprised when strings, tuples, and lists are iterable. A set, clearly, must be iterable, even if the order of elements may be difficult to predict. A mapping will iterate over the keys by default; other iterators are available. A file iterates over the available lines. A regular expression has a method, `finditer()`, that is an iterator over each instance of a matching substring that it can find. The `Path.glob()` and `Path.walk()` methods will iterate over items in a directory. The `range()` object is also an iterator. You get the idea: anything even vaguely collection-like will support some kind of iterator.

Comprehensions

Comprehensions offer simple, but powerful, syntax, allowing us to transform or filter an iterable object in as little as one line of code. The resultant object can be a perfectly normal list, set, or dictionary, or it can be a *generator expression* that can be efficiently consumed while keeping just one element in memory at a time.

List comprehensions

List comprehensions are one of the most powerful tools in Python, so people tend to think of them as advanced. They're not. Indeed, we've taken the liberty of littering previous examples with comprehensions, assuming you would understand them. A comprehension is fundamental to Python; it can handle many of the most common operations in application software.

Let's have a look at one of those common operations: namely, converting a list of items into a list of derived items. Specifically, let's assume we just read a list of strings from a file, and now we want to convert it into a list of integers. We know that every item in the list is an integer, and we want to do

some activity (say, calculate an average) on those numbers. Here's one simple way to approach it:

```
>>> input_strings = ["1", "5", "28", "131", "3"]

>>> output_integers = []
>>> for num in input_strings:
...     output_integers.append(int(num))
```

This works fine and it's only three lines of code. If you aren't used to comprehensions, you may not even think that it looks ugly! Now, look at the same code using a list comprehension:

```
>>> output_integers = [int(num) for num in input_strings]
```

We're down to one line and, importantly for performance, we've dropped an `append` method call for each item in the list. Overall, it's pretty easy to tell what's going on, even if you're not used to comprehension syntax.

The square brackets indicate, as always, that we're creating a list. Inside this list is a `for` clause that iterates over each item in the input sequence, assigning each item to the `num` variable. The only thing that may be confusing is what's happening between the list's opening brace and the start of the `for` statement. Whatever expression is provided here is applied to *each* of the items in the input list. The item in question is referenced by the `num` variable from the `for` clause. So, this expression applies the `int` function to each element. The collected result forms a new list.

Terminology-wise, we call this a **mapping**. We are applying the result expression, `int(num)` in this example, to map values from the source iterable to create a resulting iterable list.

That's all there is to a basic list comprehension. Comprehensions are highly optimized, making them faster than `for` statements when processing a large number of items. When used wisely, they're also more readable. These are two compelling reasons to use them widely.

Converting one list of items into a related list isn't the only thing we can do with a list comprehension. We can also choose to exclude certain values by adding an `if` statement inside the comprehension. We call this a **filter**. Have a look:

```
>>> output_integers = [int(num) for num in input_strings if len(num) < 3]
>>> output_integers
[1, 5, 28, 3]
```

The essential difference between this example and the previous one is the `if len(num) < 3` clause. This extra code excludes any strings with more than two characters. The `if` clause is applied to each element **before** the final `int()` function, so it's testing the length of a string. Since our input strings are all integers at heart, it excludes any number over 99.

A list comprehension can be used to map input values to output values, applying a filter along the way to include or exclude any values that meet a specific condition. A great many algorithms involve mapping and filtering operations.

Any iterable can be the input to a list comprehension. In other words, anything we can wrap in a `for` statement can also be used as the source for a comprehension.

For example, text files are iterable; each call to `__next__()` on the file's iterator will return one line of the file. We can examine the lines of a text file by naming the open file in the `for` clause of a list comprehension. We can then use the `if` clause to extract interesting lines of text. This example finds a subset of lines in a test file:

```
>>> from pathlib import Path
```

```
>>> source_path = Path('src') / 'iterator_protocol.py'
>>> with source_path.open() as source:
...     examples = [line.rstrip()
...         for line in source
...         if ">>>" in line]
```

In this example, we've added some whitespace to make the comprehension more readable (list comprehensions don't *have* to fit on one physical line even though they're one logical line). This example creates a list of lines that have the »> prompt in them. The presence of `">>>"` suggests there might be a doctest example in this file. Each line has `rstrip()` applied to remove trailing whitespace, like the `n` that ends each line of text returned by the iterator. The resulting list object, `examples`, suggests some of the test cases that can be found within the code. (This isn't as clever as doctest's own parser.)

Let's extend this example to capture the line numbers for each example with a »> prompt in it. This is a common requirement, and the built-in `enumerate()` function helps us pair a number with each item provided by the iterator:

```
>>> source_path = Path('src') / 'iterator_protocol.py'
>>> with source_path.open() as source:
...     examples = [(number, line.rstrip())
...         for number, line in enumerate(source, start=1)
...         if ">>>" in line]
```

The `enumerate()` function consumes an iterable, providing an iterable sequence of two-tuples of a number and the original item. If the line passes our `">>>" in line` test, we'll create a two-tuple of the number and the cleaned-up text. We've done some sophisticated processing in—effectively—one line of code. Essentially, though, it's a filter and a mapping. First, it extracts lines from the source, then it filters the lines that match the given `if` clause, then it evaluates the `(number, line.rstrip())` expression to create resulting tuples. The result is collected into a list object. The ubiquity of this iterate-filter-map-collect pattern drives the idea behind a list comprehension.

Set and dictionary comprehensions

Comprehensions aren't restricted to lists. We can use a similar syntax with braces (`{}`) to create sets and dictionaries as well. Let's start with sets. One way to create a set is to wrap a list comprehension in the `set()` constructor, which converts it to a set. But why waste memory on an intermediate list that gets discarded when we can create a set directly?

Here's an example that uses a named tuple to model author/title/genre triples, and then retrieves a set of all the authors that write in a specific genre:

```
from typing import NamedTuple


class Book(NamedTuple):
    author: str
    title: str
    genre: str
```

```
%>>> from typing import NamedTuple
%>>> class Book(NamedTuple):
%...     author: str
%...     title: str
%...     genre: str
>>> books = [
```

```
...     Book("Pratchett", "Nightwatch", "fantasy"),
...     Book("Pratchett", "Thief Of Time", "fantasy"),
...     Book("Le Guin", "The Dispossessed", "scifi"),
...     Book("Le Guin", "A Wizard Of Earthsea", "fantasy"),
...     Book("Jemisin", "The Broken Earth", "fantasy"),
...     Book("Turner", "The Thief", "fantasy"),
...     Book("Phillips", "Preston Diamond", "western"),
...     Book("Phillips", "Twice Upon A Time", "scifi"),
... ]
```

We've defined a small library of instances of the `Book` class. We can create a set from each of these objects by using a set comprehension. It looks a lot like a list comprehension, but uses instead of `[]`:

```
>>> fantasy_authors = {b.author for b in books if b.genre == "fantasy"}
```

The comprehensions!set comprehensionshighlighted set comprehension sure is compact. If we were to use a list comprehension, of course, Terry Pratchett would have been listed twice. As it is, the nature of sets removes the duplicates, and we end up with the following:

```
{'Jemisin', 'Le Guin', 'Pratchett', 'Turner'}
```

Note that sets don't have a defined ordering, so your output may differ from this example. For testing purposes, it helps to set the `PYTHONHASHSEED` environment variable to a fixed value to impose a predictable order. This introduces a tiny security vulnerability, so it's only suitable for testing.

Still using braces, we can introduce a colon to make `key:value` pairs required to create a dictionary comprehension. For example, it may be useful to quickly look up the author or genre in a dictionary if we know the title. We can use a dictionary comprehension to map titles to `books` objects:

```
>>> fantasy_titles = {b.title: b for b in books if b.genre == "fantasy"}
```

Now, we have a dictionary, and can look up books by title using the normal syntax, `fantasy_titles['Nightwatch']`. We've created a high-performance index from a lower-performance sequence.

In summary, comprehensions are comprehensions!set comprehensionsnot advanced Python, nor

are they features that subvert object-oriented programming. They are a more concise syntax for creating a list, set, or dictionary from an existing iterable source of data.

Generator expressions

Sometimes we want to process a new sequence without building a new list, set, or dictionary into system memory. We might be iterating over items one at a time, and don't actually care about having a complete container (such as a list or dictionary) created. Processing one item at a time means we only need the current object available in memory at any one moment. But when we create a container, all the objects have to be stored in that container before we start processing them.

For example, consider a program that processes log files. A very simple log might contain information in this format:

```
Apr 05, 2021 20:03:29 DEBUG This is a debugging message.
Apr 05, 2021 20:03:41 INFO This is an information method.
Apr 05, 2021 20:03:53 WARNING This is a warning. It could be serious.
Apr 05, 2021 20:03:59 WARNING Another warning sent.
Apr 05, 2021 20:04:05 INFO Here's some information.
Apr 05, 2021 20:04:17 DEBUG Debug messages are only useful if you want to
figure something out.
Apr 05, 2021 20:04:29 INFO Information is usually harmless, but helpful.
Apr 05, 2021 20:04:35 WARNING Warnings should be heeded.
Apr 05, 2021 20:04:41 WARNING Watch for warnings.
```

Log files for popular web servers, databases, or email servers can contain many gigabytes of data (one of the authors once had to clean nearly two terabytes of logs off a misbehaving system). If we want to process each line in the log, we can't use a list comprehension; it would create a list containing every line in the file. This probably wouldn't fit in RAM and could bring the computer to its knees, depending on the operating system.

If we used a `for` statement on the log file, we could process one line at a time before reading the next one into memory. Wouldn't it be nice if we could use comprehension syntax to get the same effect?

This is where **generator expressions** come in. They use the same syntax as comprehensions, but they don't create a final container object. We call them **lazy**: they reluctantly produce values on demand. To create a generator expression, wrap the comprehension in `()` instead of `[]` or .

The following code extracts a subset of lines from a log file in the previously presented format. It will use a generator expression to define the filter; it outputs a new log file that contains only the `WARNING` lines:

```
>>> from pathlib import Path
>>> full_log_path = Path.cwd() / "data" / "sample.log"
>>> warning_log_path = Path.cwd() / "data" / "warnings.tab"

>>> with open(full_log_path) as source:
...     warning_lines = (line for line in source if "WARN" in line)
...     with open(warning_log_path, 'w') as target:
...         for line in warning_lines:
...             target.write(line)
```

We've opened the `sample.log` file, a file perhaps too large to fit in memory. A generator expression will filter out the warnings (in this case, it uses the `if` syntax and leaves the line unmodified). This is lazy, and doesn't really do anything until we consume its output. We can open another file into which to write a subset of log lines. The final `for` statement consumes each individual line from the `warning_lines` generator. At no time is the full log file read into memory; the processing happens one line at a time.

If we run it on our sample file, the resulting `warnings.log` file looks like this:

```
Apr 05, 2021 20:03:53 WARNING This is a warning. It could be serious.
Apr 05, 2021 20:03:59 WARNING Another warning sent.
Apr 05, 2021 20:04:35 WARNING Warnings should be heeded.
Apr 05, 2021 20:04:41 WARNING Watch for warnings.
```

Of course, with a short input file, we could have safely used a list comprehension, doing all the processing in memory. When the file is millions of lines long, the generator expression will have a huge impact on both memory and speed.

The core of a comprehension is the generator expression. Wrapping a generator in `[]` creates a list. Wrapping a generator in creates a set. Using and : to separate keys and values creates a dictionary. Wrapping a generator in `()` is still a generator expression, not a tuple. To make a tuple, we must explicitly use the `tuple()` function. (We can also use the `list()`, `dict()`, and `set()` functions to make the comprehension's result clearer.)

Generator expressions are frequently most useful inside function calls. For example, we can call

`sum`, `min`, or `max` on a generator expression instead of a list, since these functions process one object at a time. We're only interested in the aggregate result, not any intermediate container.

In general, of the four options, a generator expression should be used whenever possible. If we don't actually need a list, set, or dictionary, but simply need to filter or apply a mapping to items in a sequence, a generator expression will be most efficient. If we need to know the length of a list, or sort the result, remove duplicates, or create a dictionary, we'll have to use the comprehension syntax and create a collection.

Generator functions have a small limitation: they're expressions. If we want to make use of statements such as `try` or `match`, we'd really like to write an entire function that will behave like a generator expression.

Generator functions

Generator functions embody the essential features of a generator expression, which is the generalization of a comprehension. The generator function syntax looks even less object-oriented than anything we've seen, but we'll discover that once again, it is a syntax shortcut to create a kind of iterator object. It helps us build processing following the standard iterator-filter-mapping pattern.

Let's take the log file example a little further. If we want to decompose the log into columns, we'll have to do a more significant transformation as part of the mapping step. This will involve a regular expression to find the timestamp, the severity word, and the message as a whole. We'll look at a number of solutions to this problem to show how generators and generator functions can be applied to create the objects we want.

Here's another version of a log file parser. This extracts a CSV-formatted file for further analysis. It also avoids generator expressions entirely, making it a bit more deeply nested:

```
import csv
import re
from pathlib import Path
from typing import Match, cast


def extract_and_parse_1(source_log_path: Path, warning_tab_path: Path) ->
None:
    with warning_tab_path.open("w", newline="") as target:
        writer = csv.writer(target, delimiter="\t")
```

```
        pattern = re.compile(r"(\w\w\w \d\d, \d\d\d\d \d\d:\d\d:\d\d) (\w+)
        (.*)")
        with source_log_path.open() as source:
            for line in source:
                if line_match := pattern.match(line):
                    line_groups = line_match.groups()
                    if "WARN" in line_groups[1]:
                        writer.writerow(line_groups)
```

We've defined a regular expression to match three groups:

- The complex date string expression, `(\w\w\w \d\d, \d\d\d\d \d\d:\d\d:\d\d)`, is a generalization of strings such as `"Apr 05, 2021 20:04:41"`. Some folks prefer `(\w{3} \d{2}, \d{4} \d{2}:\d{2}:\d{2})`, which saves counting the pattern elements.
- The severity level expression, `(\w+)`, matches a run of letters, digits, or underscores. This will match words such as INFO and DEBUG.
- An optional message expression, `(.*)`, will collect all characters to the end of the line.

This pattern is assigned to the `pattern` variable.

The decomposition of the line into groups involves two steps. First, we apply `pattern.match()` to the line of text to create a `Match` object. Then, we interrogate the `Match` object for the sequence of groups that matched.

This deeply nested function seems maintainable, but writing so many levels of indent in so few lines is a potential problem. The deeply nested code means a shift in focus from file processing to line processing to pattern matching and from there to processing the matches. More alarmingly, if there is some irregularity in the file, and we want to handle the case where `pattern.match(line)` returns `None`, the `if` statement doesn't do anything. Error handling may lead to even deeper levels of nesting. Deeply nested conditional processing leads to statements where the conditions under which those statements are executed can be obscure, muddled by shifting contexts.

The reader of this code has to mentally integrate all of the preceding `if` statements to work out the condition under which processing happens. This can be a problem with this kind of design.

Now let's consider a more object-oriented solution, without any shortcuts. We'll break this into two parts: the initial class definition for the iterator, followed by a function to use the iterator:

```
import csv
import re
from pathlib import Path
from typing import Match, cast, Iterator, TextIO


class WarningReformat(Iterator[tuple[str, ...]]):
    pattern = re.compile(r"(\w\w\w \d\d, \d\d\d\d \d\d:\d\d:\d\d) (\w+)
    (.*)")

    def __init__(self, source: TextIO) -> None:
        self.insequence = source

    def __iter__(self) -> Iterator[tuple[str, ...]]:
        return self

    def __next__(self) -> tuple[str, ...]:
        line = self.insequence.readline()
        while line:
            if match := self.pattern.match(line):
                groups = match.groups()
                if "WARN" in groups[1]:
                    return groups
            line = self.insequence.readline()
        raise StopIteration
```

```
def extract_and_parse_2(full_log_path: Path, warning_log_path: Path) ->
None:
    with warning_log_path.open("w", newline="") as target:
        writer = csv.writer(target, delimiter="\t")
        with full_log_path.open() as source:
            filter_reformat = WarningReformat(source)
            for line_groups in filter_reformat:
                writer.writerow(line_groups)
```

We've defined a formal `WarningReformat` iterator that emits the three-tuple of the date, warning, and message. We've used a type hint of `tuple[str, ...]` because it matches the output from the `self.pattern.match(line).groups()` expression: it's a sequence of strings, with no constraint on how many will be present. The iterator is initialized with a `TextIO` object, something file-like that has a `readline()` method.

This `__next__()` method reads lines from the file, discarding any lines that are not `WARNING` lines. When we encounter a `WARNING` line, we parse it and return the three-tuple of strings.

The `extract_and_parse_2()` function uses an instance of the `WarningReformat` class in a `for` statement. This will evaluate the `__next__()` method repeatedly to process each subsequent `WARNING` line. When the source iterator runs out of lines, the `WarningReformat` class raises a `StopIteration` exception to tell the collaborating `for` statement that there's no more data. This is distinct from previous generators where the generator function does this implicitly. When implementing the `__next__()` method, the exception must be raised explicitly. It's pretty ugly compared to the other examples, but it's also powerful; now that we have a class in our hands, we can do whatever we want with it.

With that background behind us, we finally get to see true generators in action. This next example does *exactly* the same thing as the previous one: it creates an object with a `__next__()` method that raises `StopIteration` when it's out of inputs:

```
import csv
import re
from pathlib import Path
from typing import Match, cast, Iterator, Iterable


def warnings_filter(source: Iterable[str]) -> Iterator[tuple[str, ...]]:
    pattern = re.compile(r"(\w\w\w \d\d, \d\d\d\d \d\d:\d\d:\d\d) (\w+)
    (.*)")
    for line in source:
        if "WARN" in line:
            yield tuple(cast(Match[str], pattern.match(line)).groups())


def extract_and_parse_3(full_log_path: Path, warning_log_path: Path) ->
None:
    with warning_log_path.open("w", newline="") as target:
        writer = csv.writer(target, delimiter="\t")
        with full_log_path.open() as infile:
            filter = warnings_filter(infile)
            for line_groups in filter:
                writer.writerow(line_groups)
```

The `yield` statement in the `warning_filters()` function is the key to generators. When Python sees `yield` in a function, it takes that function and wraps it up in an object that follows the `Iterator`

protocol, not unlike the class defined in our previous example. Think of the `yield` statement as similar to the `return` statement; it returns a line. Unlike `return`, however, the function is only suspended. When it is called again (via `next()`), it will start where it left off — on the line after the `yield` statement — instead of at the beginning of the function. In this example, there is no line *after* the `yield` statement, so it jumps to the next iteration of the containing `for` statement. Since the `yield` statement is inside an `if` statement, it only yields lines that contain `WARNING`.

While it looks like this is nothing more than a function looping over the lines, it is actually creating a special type of object, a generator object:

```
>>> print(warnings_filter([]))
<generator object warnings_filter at 0xb728c6bc>
```

What this function does is create and return a generator object. In this example, an empty list was provided, and a generator was built. The generator object has `__iter__()` and `__next__()` methods on it, just like the one we created from a class definition in the previous example. (Using the `dir()` built-in function on it will reveal what else is part of a generator.) Whenever the `__next__()` method is called, the generator runs the function until it reaches a `yield` statement. It then suspends execution, retaining the current state and returning the value from `yield`. The next time the `__next__()` method is called, it restores the state and picks up execution where it left off.

This generator function is nearly identical to this generator expression:

```
warnings_filter = (
    tuple(cast(Match[str], pattern.match(line)).groups())
    for line in source
    if "WARN" in line
)
```

```
warnings_filter = (
    tuple(cast(Match[str], pattern.match(line)).groups())
    for line in source
    if "WARN" in line
)
```

We can see how these various patterns align. The generator expression has all the elements of the statements, slightly compressed, and in a different order:

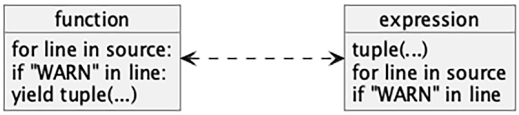


Figure 10.2: Generator functions compared with generator expressions

A comprehension, then, is a generator wrapped in `[]` or to create a concrete object. In some cases, it can make sense to use `list()`, `set()`, or `dict()` as a wrapper around a generator. This is helpful when we're considering replacing a generic collection with a customized collection of our own. Changing `list()` into `MySpecialContainer()` seems like a slightly lower risk than changing `[]` to a class name.

The generator expression has the advantage of being short and appearing right where it's needed. The generator function has a name and parameters, meaning it can be reused. More importantly, a generator function can have multiple statements and more complex processing logic in the cases where statements are needed. One common reason for switching from a generator expression to a complete generator function is to add exception handling.

Yielding items from another iterable

Often, when we build a generator function, we end up in a situation where we want to yield data from another iterable object, possibly a list comprehension or a generator expression that we constructed inside a generator. Or, perhaps we need to yield some external items that were passed into the function. We'll look at how to do this with the `yield from` statement.

Let's adapt the generator example a bit so that instead of accepting an input file, it accepts the name of a directory. The idea is to keep our existing warnings filter generator in place, but tweak the structure of the functions that use it. We'll operate on iterators as both input and result; this way, the same function could be used regardless of whether the log lines came from a file, memory, the web, or another iterator.

This version of the code illustrates a new `file_extract()` generator. This does some basic setup before yielding information from the `warnings_filter()` generator:

```
import csv
import re
from pathlib import Path
```

```
from typing import Match, cast, Iterator, Iterable


def file_extract(path_iter: Iterable[Path]) -> Iterator[tuple[str, ...]]:
    for path in path_iter:
        with path.open() as infile:
            yield from warnings_filter(infile)


def extract_and_parse_d(directory: Path, warning_log_path: Path) -> None:
    with warning_log_path.open("w", newline="") as target:
        writer = csv.writer(target, delimiter="\t")
        log_files = list(directory.glob("sample*.log"))
        for line_groups in file_extract(log_files):
            writer.writerow(line_groups)
```

Our top-level `extract_and_parse_d()` function has a slight change to use the `file_extract()` function instead of opening a file and applying `warnings_filter()` to one file. The `file_extract()` generator will yield all of the `WARNING` lines from *all* of the files provided in the argument value.

The `yield from` syntax is a useful shortcut when writing chained generators.

What's central in this example is the laziness of each of the generators involved. Consider what happens when the `extract_and_parse_d()` function, the client, makes a demand:

1. The client evaluates `file_extract(log_files)`. Since this is in a `for` statement, there's an `__iter__()` method evaluation.
2. The `file_extract()` generator gets an iterator from the `path_iter` iterable, and uses this to get the next `Path` instance. The `Path` object is used to create a file object that's provided to the `warnings_filter()` generator.
3. The `warnings_filter()` generator uses the file's iterator over lines to read until it finds a `WARNING` line, which it parses, yielding exactly one tuple. The fewest number of lines are read to find this line.
4. The `file_extract()` generator is yielding from the `warnings_filter()` generator, so the single tuple is provided to the ultimate client, the `extract_and_parse_d()` function.
5. The `extract_and_parse_d()` function writes the single tuple to the open CSV file, and then demands another tuple. This request goes to `file_extract()`, which pushes the demand down to `warnings_filter()`, which pushes the demand to an open file to provide lines until

a WARNING line is found.

Each generator is lazy and provides one response, doing the least amount of work it can get away with to produce the result. This means that a directory with a huge number of giant log files is processed by having one open log file, and one current line being parsed and processed. It won't fill memory no matter how large the files are.

This hand-off from generator to generator allows us to build sophisticated processing from simple elements.

We've seen how generator functions can provide data to other generator functions. We can do this with ordinary generator expressions also. We'll make some small changes to the `warnings_filter()` function to show how we can create a stack of generator expressions.

Generator stacks

The generator function (and the generator expression) for the `warnings_filter()` function makes an unpleasant assumption. The use of `cast()` makes a claim to **mypy** that's—perhaps—a bad claim to make. Here's an example:

```
warnings_filter = (
    tuple(cast(Match[str], pattern.match(line)).groups())
    for line in source
    if "WARN" in line
)
```

The use of `cast()` is a way of claiming that `pattern.match()` will always yield a `Match[str]` object. This isn't a great assumption to make. Someone may change the format of the log file to include a multiple-line message, and our `WARNING` filter would crash every time we encountered a multi-line message. (The subsequent lines would not have the expected format of a timestamp, a severity, and a message.)

Here's a message that would cause problems followed by a message that's easy to process:

```
Jan 26, 2015 11:26:01 INFO This is a multi-line information
message, with misleading content including WARNING
and it spans lines of the log file WARNING used in a confusing way
```

```
Jan 26, 2015 11:26:13 DEBUG Debug messages are only useful if you want to
figure something out.
```

The first line has the word `WARNING` in a multi-line message that will break our assumption about lines that contain the text "`WARN`". We need to handle this with a little more care.

We can rewrite this generator expression to create a generator function, and add an assignment statement (to save the `Match` object) and an `if` statement to further decompose the filtering. We can use the walrus operator, `:=`, to save the `Match` object, also.

We could reframe the generator expression as the following generator function:

```
def warnings_filter(source: Iterable[str]) -> Iterator[Sequence[str]]:
    pattern = re.compile(r"(\w\w\w \d\d, \d\d\d\d \d\d:\d\d:\d\d) (\w+)
    (.*)")
    for line in source:
        if match := pattern.match(line):
            if "WARN" in match.group(2):
                yield match.groups()
```

As we noted, this complex filtering tends toward deeply nested `if` statements, which can create logic that's difficult to summarize. In this case, the two conditions aren't terribly complex. An alternative is to change this into a series of map and filter stages, each of which does a separate, small transformation on the input. We can decompose the matching and filtering into the following:

- Map the source line to an `Match[str] | None` object using the `pattern.match()` method
- Filter to reject any `None` objects, passing only good `Match` objects and applying the `groups()` method to create a `List[str]`
- Filter the strings to reject the non-`WARN` lines, and pass the `WARN` lines

Each of these stages is a generator expression following the standard pattern. We can expand the `warnings_filter` expression into a stack of three expressions:

```
possible_match_iter = (pattern.match(line) for line in source)
group_iter = (match.groups() for match in possible_match_iter if match)
warnings_filter = (group for group in group_iter if "WARN" in group[1])
```

These expressions are, of course, utterly lazy. The final `warnings_filter` uses the `group_iter`

iterable. This iterable gets matches from another generator, `possible_match_iter`, which gets source text lines from the `source` object, an iterable source of lines. Since each of these generators is getting items from another lazy iterator, there's only one line of data being processed through the `if` clause and the final expression clause at each stage of this process.

Note that we can exploit the surrounding `()` to break each expression into multiple lines. This can help show the map or filter operation that's embodied in each expression.

We can inject additional processing as long as it fits this essential mapping-and-filtering design pattern. Before moving on, we're going to switch to a slightly more friendly regular expression for locating lines in our log file:

```
pattern = re.compile(
    r"(?P<dt>\w\w\w \d\d, \d\d\d\d \d\d:\d\d:\d\d)"
    r"\s+(?P<level>\w+)"
    r"\s+(?P<msg>.*)"
)
```

This regular expression is broken into three adjacent strings; Python will automatically concatenate string literals. The expression uses three named groups. The date-time stamp, for example, is group number one, a hard-to-remember bit of trivia. The `?P<dt>` inside the `()` means the `groupdict()` method of a `Match` object will have the `dt` key in a resulting dictionary of match groups. As we introduce more processing steps, we'll need to be much clearer about the intermediate results.

Here's a diagram of a regular expression that may be helpful:

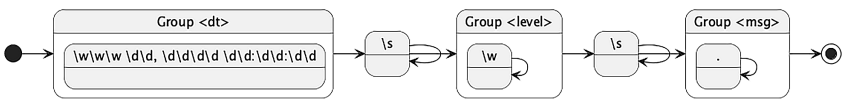


Figure 10.3: Log line regular expression diagram

Let's expand this example to transform the date-time stamp to another format. This involves injecting a transformation from the input format to the desired output format. We can do this in one big gulp, or we can do it in a series of small sips.

This sequence of steps makes it easier to add or change one individual step without breaking the entire processing pipeline:

```
possible_match_iter = (pattern.match(line) for line in source)
group_iter = (match.groupdict() for match in possible_match_iter if match)
warnings_iter = (group for group in group_iter if "WARN" in group["level"])
dt_iter = (
    (
        datetime.datetime.strptime(g["dt"], "%b %d, %Y %H:%M:%S"),
        g["level"],
        g["msg"],
    )
    for g in warnings_iter
)
warnings_filter = ((g[0].isoformat(), g[1], g[2]) for g in dt_iter)
```

We've created two additional stages. One parses the input time to create a Python `datetime` object; the second stage formats the `datetime` object as an ISO. Breaking the transformation down into small steps lets us treat each mapping operation and each filtering operating as discrete, separate steps. We can add, change, and delete with a little more flexibility when we create these smaller, easier-to-understand steps. The idea is to isolate each transformation into a separate object, described by a generator expression.

The result of the `dt_iter` expression is an iterable over anonymous tuples. This is a place where a `NamedTuple` can add clarity. See *Chapter 8*, for more information on `NamedTuple`.

We have an additional way to look at these transformational steps, using the built-in `map()` and `filter()` functions. These functions provide features similar to generator expressions, using another, slightly different syntax:

```
possible_match_iter = map(pattern.match, source)
good_match_iter = filter(None, possible_match_iter)
group_iter = map(lambda m: m.groupdict(), good_match_iter)
warnings_iter = filter(lambda g: "WARN" in g["level"], group_iter)
dt_iter = map(
    lambda g: (
        datetime.datetime.strptime(g["dt"], "%b %d, %Y %H:%M:%S"),
        g["level"],
        g["msg"],
    ),
    warnings_iter,
)
warnings_filter = map(lambda g: (g[0].isoformat(), g[1], g[2]), dt_iter)
```

The lambda objects are anonymous functions. A lambda is a callable object with parameters and a single expression that is evaluated and returned. There's no name and no statements in the body of a lambda. Each stage in this pipeline is a discrete mapping or filtering operation. While we can combine mapping and filtering into a single `map(lambda ..., filter(lambda ..., source))`, this can be too confusing to be helpful.

The stack of generators works by applying small transformations to respond to the demand for data by some collaborating object that has a `for row in warnings_filter:` statement. Here are the stages:

1. The `possible_match_iter` applies the `pattern.match()` to a line of input.
2. The `good_match_iter` uses the special `filter(None, source)` to pass non-`None` objects, and rejects `None` objects. Using `None` is a special case of the `filter()` function; it saves us from writing a lambda object to test for "falsey" objects such as `None` or a zero-length string.
3. The `group_iter` uses a lambda to evaluate `m.groups()` for each object, `m`, in `good_match_iter`.
4. The `warnings_iter` will filter the `group_iter` results, keeping only the `WARN` lines and rejecting all others.
5. The `dt_iter` performs a conversion from the source datetime format to a generic `datetime` object.
6. The final `warnings_filter` expression reformats the `datetime` object in a different string format.

We've seen a number of ways of approaching a complex map-filter problem. We can write nested `for` and `if` statements. We can create explicit `Iterator` subclass definitions. We can create iterator-based objects using function definitions that include the `yield` statement. This provides us with the formal interface of the `Iterator` class without the lengthy boilerplate required to define `__iter__()` and `__next__()` methods of an object. Additionally, we can use generator expressions and even comprehensions to apply the iterator design pattern in a number of common contexts.

The iterator pattern is a foundational aspect of Python programming. Every time we work with a collection, we'll be iterating through the items, and we'll be using an iterator. Because iteration is so central, there are a variety of ways to tackle the problem. We can use `for` statements, generator functions, and generator expressions, and we can build our own iterator classes.

Recall

This chapter looked at a design pattern that seems ubiquitous in Python, the iterator. The Python iterator concept is the foundation of the language and is used widely. In this chapter, we examined a number of aspects:

- Design patterns are good ideas that we see repeated in software implementations, designs, and architectures. A good design pattern has a name, and a context where it's usable. Because it's only a pattern, not reusable code, the implementation details will vary each time the pattern is followed.
- The `Iterator` protocol is one of the most powerful design patterns because it provides a consistent way to work with data collections. We can view strings, tuples, lists, sets, and even files as iterable collections. A mapping contains a number of iterable collections, including the keys, the values, and the items (key and value pairs).
- List, set, and dictionary comprehensions are short, pithy summaries of how to create a new collection from an existing collection. They involve a source iterable, an optional filter, and a final expression to define the objects in the new collection.
- Generator functions build on other patterns. They let us define iterable objects that have map and filter capabilities.

Exercises

If you don't use comprehensions in your daily coding very often, the first thing you should do is search through some existing code and find some `for` statements. See whether any of them can be trivially converted to a generator expression or a list, set, or dictionary comprehension.

Test the claim that list comprehensions are faster than the `for` statement. This can be done with the built-in `timeit` module. Use the help documentation for the `timeit.timeit` function to find out how to use it. Basically, write two functions that do the same thing, one using a list comprehension and one using a `for` statement to iterate over several thousand items. Pass each function into `timeit.timeit`, and compare the results. If you're feeling adventurous, compare generators and generator expressions as well. Testing code using `timeit` can become addictive, so bear in mind that code does not need to be hyperfast unless it's being executed an immense number of times, such as on a huge input list or file.

Play around with generator functions. Start with basic iterators that require multiple values

(mathematical sequences are canonical examples; the Fibonacci sequence is overused if you can't think of anything better). Try some more advanced generators that do things such as take multiple input lists and somehow yield values that merge them. Generators can also be used on files; can you write a simple generator that shows lines that are identical in two files?

Extend the log processing exercise to replace the `WARNING` filter with a time range filter; all the messages between January 26, 2015, 11:25:46, and January 26, 2015, 11:26:15, for example.

Once you can find `WARNING` lines or lines within a specific time, combine the two filters to select only the warnings within the given time. You can use an `and` condition within a single generator, or combine multiple generators, in effect building an `and` condition. Which seems more adaptable to changing requirements?

When we presented the class `WarningReformat(Iterator[Tuple[str, ...]])`: example of an iterator, we made a questionable design decision. The `__init__()` method accepted an open file as an argument value and the `__next__()` method used `readline()` on that file. What if we change this slightly and create an explicit iterator object that we use inside another iterator?

```
def __init__(self, source: TextIO) -> None:
    self.insequence = iter(source)
```

If we make this change, then `__next__()` can use `line = next(self.insequence)` instead of `line = self.insequence.readline()`. Switching from `object.readline()` to `next(object)` is an interesting generalization. Does it change anything about the `extract_and_parse_2()` function? Does it permit us to use generator expressions along with the `WarningReformat` iterator?

Take this one further step. Refactor the `WarningReformat` class into two separate classes, one to filter for `WARN` and a separate class to parse and reformat each line of the input log. Rewrite the `extract_and_parse_2()` function using instances of these two classes. Which is "better"? What metric did you use to evaluate "better"?

Look at the recipes section of the `itertools` module. How can the `itertools.partition()` function be used to partition data?

Summary

In this chapter, we learned that design patterns are useful abstractions that provide best-practice solutions to common programming problems. We covered our first design pattern, the iterator, as

well as numerous ways that Python uses and abuses this pattern for its own nefarious purposes. The original iterator pattern is extremely object-oriented, but it is also rather ugly and verbose to code around. However, Python's built-in syntax abstracts the ugliness away, leaving us with a clean interface to these object-oriented constructs.

Comprehensions and generator expressions can combine container construction with iteration in a single line. Generator functions can be constructed using the `yield` statement.

We'll cover several more design patterns in the next two chapters.

11

Common Design Patterns

In the previous chapter, we introduced the concept of a design pattern, and covered the iterator pattern, a pattern so useful and common that it has been abstracted into the core of the programming language itself. In this chapter, we'll be reviewing other common patterns and how they are implemented in Python. As with iteration, Python often provides an alternative syntax to make working with such problems simpler. We will focus on the Python implementations for these patterns, especially when the pattern is a first-class part of the Python language.

In this chapter, we'll look at the following:

- The Decorator pattern
- The Observer pattern
- The Strategy pattern
- The Command pattern
- The State pattern
- The Singleton pattern

Consistent with the practice in the book *Design Patterns: Elements of Reusable Object-Oriented Software*, we'll capitalize the pattern names. This can help them stand out from ordinary English usage.

We'll start with the Decorator pattern. This is used to combine different kinds of functionality into a single resulting object.

The Decorator pattern

The Decorator pattern allows us to *wrap* an object that provides core functionality with other objects that alter this functionality. Any object that uses the decorated object will interact with it in exactly the same way as if it were undecorated (that is, the interface of the decorated object is identical to that of the core object).

The Decorator design pattern is part of the conceptual background behind the Python language decorator, used to add features to functions and classes.

There are two primary uses of the Decorator pattern:

- Enhancing the response of a component as it sends data to a second component
- Supporting multiple optional behaviors

The second option is often an alternative to multiple inheritance, focused on composition. We can construct a core object, and then create a decorator wrapping that core. Since the decorator object has the same interface as the core object, we can even wrap the new object in other decorators. In the next page, we've shown how it looks in a UML diagram.

Here, **Core** and all the decorators implement a specific **Interface**. The dashed lines show "implements" or "realizes." The decorators maintain a reference to the core instance of that **Interface** via composition. When called, the decorator does some added processing before or after calling its wrapped interface. The wrapped object may be another decorator, or the core functionality. While multiple decorators may wrap each other, the object at the end of the chain of all those decorators provides the core functionality.

It's essential that each of these is providing an implementation of a common feature. The intent is to provide a composition of processing steps from the various decorators, applied to the core. Often decorators are small, typically a function definition without any state.

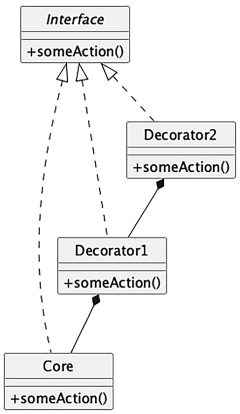


Figure 11.1: Decorator pattern in UML

In Python, because of duck typing, we don't need to formalize these relationships with an abstract interface definition. It's sufficient to make sure the classes have matching methods. In some cases, we may define `typing.Protocol` as a type hint to help tools like **mypy** reason about the relationships.

A Decorator example

Let's look at an example from network programming. We want to build two things: a small server that provides some data, and a client that interacts with that server. The server will be simulating rolling complex handfuls of dice. The client will request a handful and wait for an answer that contains some random numbers.

This example has two processes interacting via a TCP socket, a way to transmit bytes among computer systems. Sockets are created by a server that listens for connections. When a client attempts to connect to the socket, the server must accept the new connection, and the two processes can then pass bytes back and forth; for this example, there will be a request from client to server and a response from server to client. The TCP socket is part of the foundation for HTTP, around which the World Wide Web is built.

The interfaces for both client and server are similar. Both processes will use the `socket.send()` method to transmit a string of bytes through the socket. They'll also use `socket.recv()` to receive bytes; the parameter is the upper bound on the number of bytes to accept. It can help to check the documentation to confirm your understanding of these interfaces.

We'll start with an interactive server that waits for a connection from a client and then responds to the request. We'll call this module `socket_server.py`. Here's the general outline:

```
import contextlib
import socket


def main_1() -> None:
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind(("localhost", 2401))
    server.listen(1)
    with contextlib.closing(server):
        while True:
            client, addr = server.accept()
            dice_response(client)
```

The `server` is bound to the "public" socket, using a more or less arbitrary port number of `2401`. This is where the server is listening for connection requests. When a client tries to connect to this socket, a child socket is created so the client and server can talk, leaving the public socket ready for more connections. A web server will often use multiple threads to allow a large number of concurrent sessions. We're not using threads, and a second client is forced to wait until the server is done with the first client. It's a coffee shop queue with exactly one barista making espressos.

TCP/IP sockets have both a host address and a port number. The port number must be above `1023`. Port numbers `1023` and below are reserved and require special OS privileges. We chose port `2401` because it doesn't seem to be used for anything else.

The `dice_response()` function does all the real work of our service. It accepts a `socket` parameter so it can respond to the client. It reads bytes with a client request, creates a response, then sends it. In order to handle exceptions gracefully, the `dice_response()` function looks like this:

```
def dice_response(client: socket.socket) -> None:
    request = client.recv(1024)
    try:
        # Future: response = dice.dice_roller(request)
        response = dice_roller_ex(request)
```

```
    except (ValueError, KeyError) as ex:
        response = repr(ex).encode("utf-8")
    client.send(response)
```

We've wrapped another function, `dice_roller()`, in an exception handler. This is from a separate module, `dice`. This is a common pattern to separate error-handling and other overheads from the real work of computing a dice roll and responding to the client with useful numbers for their game. Here's what this underlying "so the work" `dice_roller()` function looks like. The following function builds the bytes-formatted value:

```
import random


def dice_roller_ex(request: bytes) -> bytes:
    request_text = request.decode("utf-8")
    numbers = [random.randint(1, 6) for _ in range(6)]
    response = f"{request_text} = {numbers}"
    return response.encode("utf-8")
```

This isn't too sophisticated. We'll expand on this in the *The Command pattern* section later in this chapter. For now, however, it will provide a sequence of random numbers, and encode them into a string of bytes.

Note that we're not really doing anything with the `request` object that came from the client. For the first few examples, we'll be reading these bytes and ignoring them. The `request` is a placeholder for a more complex request describing how many dice to roll and how many times to roll them.

We can leverage the Decorator design pattern to add features. The decorator will wrap the core `dice_response()` function, which is given a `socket` object that it can read and write. To make use of the design pattern, it's important to exploit the way this function relies on the `socket.send()` and `socket.recv()` methods when we add features. We need to preserve the interface definition as we add decorations.

To test the server, we can write a very simple client. This is an entirely separate module that will be executed as a separate process. This client will connect to the port the server is listening on; it makes a request and outputs the response. The essence is this function:

```
import socket
```

```
def main() -> None:
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.connect(("localhost", 2401))
    count = input("How many rolls: ") or "1"
    pattern = input("Dice pattern nd6[dk+-]a: ") or "d6"
    command = f"Dice {count} {pattern}"
    server.send(command.encode("utf8"))
    response = server.recv(1024)
    print(response.decode("utf-8"))
    server.close()
```

This client application asks two questions and creates a fairly complex-looking string, a `command` string, that contains a count and dice-rolling pattern. Right now, the server doesn't use this command. This is a teaser for a more sophisticated dice roller.

To use these two separate applications, follow these steps:

1. Open two terminal windows, side by side. (It can help to change the window titles to "client" and "server." Users of macOS Terminal can use the **change title** item in the **shell** menu. Windows users can use the `title` command.)
2. In the server window, start the server application:

   ```
   % python src/socket_server.py
   ```

3. In the client window, start the client application:

   ```
   % python src/socket_client.py
   ```

4. Enter your responses to the prompts in the client window – for example:

   ```
   How many rolls: 2
   Dice pattern nd6[dk+-]a: d6
   ```

5. The client will send the command, read the response, print it to the console, and exit. Run the client as many times as you want to get a sequence of dice rolls.

The work environment with two terminal windows will look something like this:

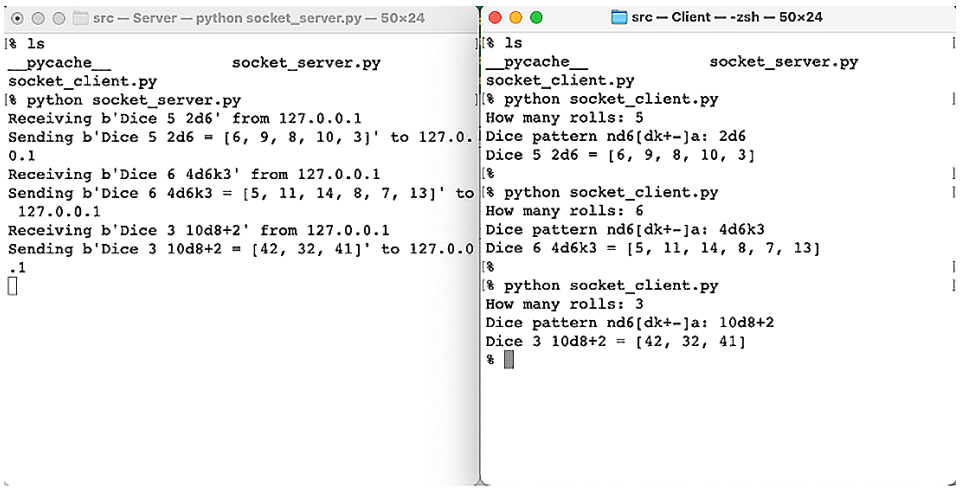


Figure 11.2: Server and client terminal windows

On the left side is the server. We started the application, and it started listening on port `2401` for clients. On the right side is the client. Each time we run the client, it connects to the public socket; the connection operation creates a child socket that can be used for the rest of the interaction. The client sends a command, the server responds to that command, and the client prints it.

(The results show more sophisticated dice rolling than the server's place-holder `dice_roller_ex()` function produces.)

Now, looking back at our server code, we see two sections. The `dice_response()` function reads data and sends data back to the client via a `socket` object. The remaining script is responsible for creating that `socket` object.

The `dice_response()` function, awkwardly, ignores the request input, and always rolls six six-sided dice. This needs to be fixed, of course, and that's the subject of the *The Command pattern* section.

For this simple server, however, there are features missing. Logging of requests and responses is the most important gap in this implementation. We'll design a decoration class to customize the socket behavior without having to extend or modify the socket itself.

We'll add a *logging* decoration. This object outputs any data being sent to the server's console before it sends it to the client:

```
class LogSocket:
    def __init__(self, socket: socket.socket) -> None:
        self.socket = socket

    def recv(self, count: int = 0) -> bytes:
        data = self.socket.recv(count)
        print(f"Receiving {data!r} from {self.socket.getpeername()[0]}")
        return data

    def send(self, data: bytes) -> None:
        print(f"Sending {data!r} to {self.socket.getpeername()[0]}")
        self.socket.send(data)

    def close(self) -> None:
        self.socket.close()
```

An instance of the `LogSocket` class provides a `socket` object with a decoration added to it. The object offers the `send()`, `recv()`, and `close()` interface to clients using it. A better decoration could properly implement all of the arguments to `send`, (which actually accepts an optional `flags` argument), but let's keep our example simple. Whenever `send()` is called on an instance of the `LogSocket` class, it logs the output to the screen before sending data to the client using the original socket. Similarly, for `recv()`, it reads and logs the data it received.

We only have to change one line in our original code to use this decorated socket. Instead of calling the `dice_response()` function with the original client socket, we call it with a decorated socket:

```
def main_2() -> None:
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind(("localhost", 2401))
    server.listen(1)
    with contextlib.closing(server):
        while True:
            client, addr = server.accept()
            logging_socket = cast(socket.socket, LogSocket(client)) # New
            dice_response(logging_socket)
            client.close()
```

We've decorated the core `socket` with a `LogSocket`. The `LogSocket` will print to the console as well as invoking methods of the socket it decorates. The essential processing in the `dice_response()` function doesn't change, because the `LogSocket` instance behaves like the underlying `socket` object.

Note that we needed to use an explicit `cast()` to tell a tool like **mypy** the `LogSocket` instance would provide a similar interface to an ordinary `socket`. For a simple case like this, we have to ask ourselves why we didn't just extend the `socket` class and override the `send` method. A subclass could call `super().send()` and `super().recv()` to do the actual sending, after we logged it. Decoration offers an advantage over inheritance: a decoration can be reused among various classes in various class hierarchies. In this specific little example, there aren't too many socket-like objects, so the possibilities of reuse are limited.

If we switch our focus to something more generic than a `socket`, we can create potentially reusable decorators. Processing strings or bytes seems more common than processing a `socket`. Changing the structure can give us some desirable flexibility in addition to reuse potential. Originally, we broke the processing into a `dice_response()` function that handled the socket reading and writing, separate from a `dice_roller()` function that works with bytes. Because the `dice_roller()` function consumes the request bytes and produces response bytes, it can be a little simpler to expand and add features to this function.

We can have a family of related decorators. We can decorate already decorated objects. The idea is to give ourselves flexibility through composition. Let's rework the logging decorator to focus on the bytes request and response instead of the `socket` object. The following should look similar to the earlier example but with some code shifted around to reside in a single `__call__()` method:

```
type Address = tuple[str, int]


class LogRoller:
    def __init__(self, dice: Callable[[bytes], bytes], remote_addr: Address)
    -> None:
        self.dice_roller = dice
        self.remote_addr = remote_addr

    def __call__(self, request: bytes) -> bytes:
        print(f"Receiving {request!r} from {self.remote_addr}")
        dice_roller = self.dice_roller
        response = dice_roller(request)
        print(f"Sending {response!r} to {self.remote_addr}")
        return response
```

Here's a second decorator that compresses data using `gzip` compression on the resulting bytes:

```
import gzip
import io
```

```
class ZipRoller:
    def __init__(self, dice: Callable[[bytes], bytes]) -> None:
        self.dice_roller = dice

    def __call__(self, request: bytes) -> bytes:
        dice_roller = self.dice_roller
        response = dice_roller(request)
        buffer = io.BytesIO()
        with gzip.GzipFile(fileobj=buffer, mode="w") as zipfile:
            zipfile.write(response)
        return buffer.getvalue()
```

This decorator compresses the response data before sending it on to the client. It decorates an underlying `dice_roller` object that computes a response to a request.

Now that we have these two decorators, we can write code that piles one decoration on top of another:

```
def dice_response(client: socket.socket) -> None:
    request = client.recv(1024)
    try:
        remote_addr = client.getpeername()
        roller_1 = ZipRoller(dice.dice_roller)
        roller_2 = LogRoller(roller_1, remote_addr=remote_addr)
        response = roller_2(request)
    except (ValueError, KeyError) as ex:
        response = repr(ex).encode("utf-8")
    client.send(response)
```

The intent here is to separate three aspects of this application:

- Doing the underlying computation
- Decorating the computation to compress the resulting document
- Decorating the computation to write a log

We can apply the zip or logging to any similar application that works with receiving and sending bytes. We can, if we want, make the zipping operation a dynamic choice, also. We might have a

separate configuration file to enable or disable the GZip feature. This means something like the following:

```
if config.zip_feature:
    roller_1 = ZipRoller(dice.dice_roller)
else:
    roller_1 = dice.dice_roller
```

This lets us have a dynamic set of decorations. Try writing this using a multiple inheritance mixin and see how confused it becomes!

Decorators in Python

The Decorator pattern is central to Python. There are, of course, additional options. For example, we can use monkey-patching — changing the class definition at runtime — to get a similar effect. For example, `socket.socket.send = log_send` will change the way the built-in socket works by replacing a method with another function. There are sometimes surprising implementation details that can make this unpleasantly complex. We discourage using monkey patches.

Single inheritance, where the *optional* calculations are done in one large method with a bunch of `if` statements, could be an option. Multiple inheritance should not be written off just because it's not suitable for the specific example seen previously.

In Python, it is very common to use this pattern on functions. As we saw in a previous chapter, functions are objects too. In fact, function decoration is so common that Python provides a special syntax to make it easy to apply such decorators to functions.

For example, we can look at the logging example in a more general way. Instead of logging only `send` calls on sockets, we may find it helpful to log all calls to certain functions or methods. The following example implements a decorator that does just this:

```
from functools import wraps


def log_args(function: Callable[..., Any]) -> Callable[..., Any]:
    @wraps(function)
    def wrapped_function(*args: Any, **kwargs: Any) -> Any:
        print(f"Calling {function.__name__}(*{args}, **{kwargs})")
        result = function(*args, **kwargs)
```

```
        return result

    return wrapped_function
```

This decorator function is very similar to the example we explored earlier. In the earlier examples, the decorator took a socket-like object and created a socket-like object. This time, our decorator takes a function object and returns a new function object. We've provided a type hint of `Callable[..., Any]` to state that any function will work here. This code comprises three separate tasks:

- A function, `log_args()`, that accepts another function, `function`, as a parameter value
- This function defines (internally) a new function, named `wrapped_function`, that does some extra work before calling the original function and returning the results from the original function
- The new inner function, `wrapped_function()`, is returned from the decorator function

Because we're using `@wraps(function)`, the new function will have a copy of the original function's name and docstring. This avoids having all of the functions we decorate wind up named `wrapped_function`.

Here's a sample function to demonstrate the decorator in use:

```
def test1(a: int, b: int, c: int) -> float:
    return sum(range(a, b + 1)) / c
```

This function can be decorated and used like this:

```
>>> test1 = log_args(test1)
>>> test1(1, 9, 2)
Calling test1(*(1, 9, 2), **{})
22.5
```

This syntax allows us to build decorated function objects dynamically, just as we did with the socket example. If we don't use assignment to assign the new object to the old name, we can even keep the decorated and the non-decorated versions for different situations. We could use a statement like `test1_log = log_args(test1)` to create a second, decorated version of the `test1()` function, named `test1_log()`.

Typically, these decorators are general modifications that are applied permanently to different functions. In this situation, Python supports a special syntax to apply the decorator at the time the function is defined. We've already seen this syntax in several places — starting in *Chapter 5* — now, let's understand how it works.

Instead of applying the decorator function after the method definition, we can use the `@decorator` syntax to do it all at once:

```
>>> @log_args
... def test1(a: int, b: int, c: int) -> float:
...     return sum(range(a, b + 1)) / c
>>> test1(1, 9, 2)
Calling test1(*(1, 9, 2), **{})
22.5
```

The primary benefit of this syntax is that we can easily see that the function has been decorated whenever we read the function definition. If the decorator is applied later, someone reading the code may miss that the function has been altered at all. Answering a question like *Why is my program logging function calls to the console?* can become much more difficult if the decoration isn't obvious! However, the syntax can only be applied to functions we define, since we don't have access to the source code of other modules. If we need to decorate functions that are part of somebody else's third-party library, we have to use the `function = decorator(function)` syntax.

Python's decorators permit parameters, also. One of the most useful decorators in the standard library is `functools.lru_cache`. The idea of a cache is to save computed results of a function to avoid recomputing them. Rather than save all of the parameters and results, we can keep the cache small by discarding the **least recently used** (**LRU**) values. For example, here's a function that involves a potentially expensive computation:

```
>>> from math import factorial
>>> def binom(n: int, k: int) -> int:
...     return factorial(n) // (factorial(k) * factorial(n - k))

>>> f"6-card deals: {binom(52, 6):,d}"
'6-card deals: 20,358,520'
```

We can use the `lru_cache` decorator to avoid doing this computation once the answer is known. Here's the small change required:

```
>>> from math import factorial
>>> from functools import lru_cache

>>> @lru_cache(64)
... def binom(n: int, k: int) -> int:
...     return factorial(n) // (factorial(k) * factorial(n - k))

>>> f"6-card deals: {binom(52, 6):,d}"
'6-card deals: 20,358,520'
```

The parameterized decorator, `@lru_cache(64)`, used to create this second version of the `binom()` function means it will save the most recent 64 results to avoid recomputing values when they've already been computed once. No change is needed elsewhere in the application. Sometimes, the speedup from this small change can be dramatic. We can, of course, fine-tune the size of the cache based on the data and the number of computations that are being performed.

Parameterized decorators like this involve a two-step dance. First, we customize the decorator with the parameter – 64 in the preceding example. Then we apply that customized decorator to a function definition. These two separate steps follow a design pattern that's a bit like a callable object: they can be initialized with the `__init__()` method, and the resulting object can be called, like a function, via their `__call__()` method.

Here's an example of a configurable logging decorator, `NamedLogger`:

```
class NamedLogger:
    def __init__(self, logger_name: str) -> None:
        self.logger = logging.getLogger(logger_name)

    def __call__(self, function: Callable[..., Any]) -> Callable[..., Any]:
        @wraps(function)
        def wrapped_function(*args: Any, **kwargs: Any) -> Any:
            start = time.perf_counter()
            try:
                result = function(*args, **kwargs)
                μs = (time.perf_counter() - start) * 1_000_000
                self.logger.info(f"{function.__name__}, {μs:.1f}μs")
                return result
            except Exception as ex:
                μs = (time.perf_counter() - start) * 1_000_000
                self.logger.error(f"{ex}, {function.__name__}, {μs:.1f}μs")
```

```
                raise

        return wrapped_function
```

The `__init__()` method makes sure we can use code like `NamedLogger("log4")` to create a decorator; this decorator will make sure the function that follows uses a specific logger.

The `__call__()` method follows the preceding pattern. We define a new function, `wrapped_function()`, that does the work, and return that newly minted function. We can use it like this:

```
>>> @NamedLogger("log4")
... def test4(median: float, sample: float) -> float:
...     return abs(sample - median)
```

We've created an instance of the `NamedLogger` class. Then we applied this instance to the `test4()` function definition. The `__call__()` method is invoked, and will create a new function, the decorated version of the `test4()` function.

There are a few more use cases for the decorator syntax. For example, when a decorator is a method of a class, it can also save information about the decorated function, creating a registry of decorated functions. Further, classes can also be decorated; in that case, the decorator returns a new class instead of a new function. In all of these more advanced cases, we're using ordinary object-oriented design with the simpler-looking syntax of `@decorator`.

The Observer pattern

The Observer pattern is useful for state monitoring and event handling situations. This pattern allows a given object to be monitored by an unknown and dynamic group of *observer* objects. The core object being observed needs to implement an interface that makes it *observable*.

Whenever a value on the core object changes, it lets all the observer objects know that a change has occurred, by calling a method announcing there's been a change of state. This is used widely in GUIs to make sure that any state change in the underlying model is reflected in the views of the model. It's common to have detail and summary views; a change to the details must also update the widgets that display the details and update any summaries that are displayed, also. Sometimes a large change in mode may lead to a number of items being changed. For instance, clicking a "lock" icon may alter a number of displayed items to reflect their status as locked. This can be

implemented as a number of observers attached to the observable display widget.

In Python, the observer can be notified via the `__call__()` method, making each observer behave like a function or other callable object. Each observer may be responsible for different tasks whenever the core object changes; the core object doesn't know or care what those tasks are, and the observers don't typically know or care what other observers are doing.

This allows tremendous flexibility by decoupling the response to a state change from the change itself.

Here is a depiction of the Observer design pattern in UML:

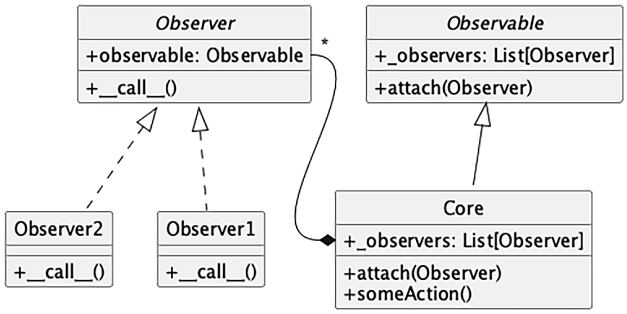


Figure 11.3: Observer pattern in UML

We've shown the `Core` object as containing a list of observer objects. To be observable, the `Core` class must adhere to a common understanding of observability; specifically, it must provide a list of observers and a way to attach new observers.

We've shown the `Observer` subclasses as having a `__call__()` method. This will be used by the observable to notify each observer of a state change. As with the Decorator pattern, we don't need to formalize the relationships with formally defined abstract superclasses. In most cases, we can rely on duck typing rules; as long as the observers have the right interface, they can be used in the defined role in this pattern. If they lack the proper interface, a tool like **mypy** may catch the conflict, and a unit test should catch the problem.

An Observer example

Outside a GUI, the Observer pattern is useful for saving intermediate states of objects. Using observer objects can be handy in systems where a rigorous audit of changes is required. It's also handy in a system where chaos reigns and components are unreliable.

Complex, cloud-based applications can suffer from chaos issues due to unreliable connections. We can use observers to record state changes, making recovery and restart easier.

For this example, we'll define a core object to maintain a collection of important values, and then have one or more observers create serialized copies of that object. These copies might be stored in a database, on a remote host, or in a local file, for example. Because we can have a number of observers, it's easy to modify the design to use different data caches. For this example, we're thinking of a dice game called Zonk or Zilch or Ten Thousand, where a player will roll six dice, score some points for triples and runs, and possibly roll again, leading to a sequence of dice rolls.

(The rules are a bit more complex than this glib summary.)

We'll start with a few overheads to help make our intention clear:

```
from typing import Protocol


class Observer(Protocol):
    def __call__(self) -> None:
        ...


class Observable:
    def __init__(self) -> None:
        self._observers: list[Observer] = []

    def attach(self, observer: Observer) -> None:
        self._observers.append(observer)

    def detach(self, observer: Observer) -> None:
        self._observers.remove(observer)

    def _notify_observers(self) -> None:
        for observer in self._observers:
            observer()
```

The `Observer` class is a protocol, an abstract superclass for our observers. We didn't formalize it as an `abc.ABC` abstract class; we're not relying on the runtime error offered by the `abc` module. When defining a `Protocol`, we're relying on tools like **mypy** to confirm that all observers actually implement the required method.

The `Observable` class defines the `_observers` instance variable and three methods that are purely

part of this protocol definition. An observable object can append an observer, remove an observer, and — most important — notify all the observers of a state change. The only thing the core class needs to do that's special or different is to make calls to the `_notify_observers()` method when there's a state change. Appropriate notification is an important piece of the design for an observable object.

Here's part of the Zonk game we care about. This class keeps a player's hands:

```
type Hand = list[int]


class ZonkHandHistory(Observable):
    def __init__(self, player: str, dice_set: Dice) -> None:
        super().__init__()
        self.player = player
        self.dice_set = dice_set
        self.rolls: list[Hand]

    def start(self) -> Hand:
        self.dice_set.roll()
        self.rolls = [self.dice_set.dice]
        self._notify_observers()  # State change
        return self.dice_set.dice

    def roll(self) -> Hand:
        self.dice_set.roll()
        self.rolls.append(self.dice_set.dice)
        self._notify_observers()  # State change
        return self.dice_set.dice
```

This class makes calls to `self._notify_observers()` on important state changes. This will notify all the observer instances. The observers might cache copies of the hand, send details over a network, update widgets on a GUI — any number of things. The `_notify_observers()` method inherited from `Observable` iterates over all registered observers and lets each know that the state of the hand has changed.

Now let's implement a simple observer object; this one will print out some state to the console:

```
class SaveZonkHand(Observer):
    def __init__(self, hand: ZonkHandHistory) -> None:
```

```
        self.hand = hand
        self.count = 0

    def __call__(self) -> None:
        self.count += 1
        message = {
            "player": self.hand.player,
            "sequence": self.count,
            "hands": json.dumps(self.hand.rolls),
            "time": time.time(),
        }
        print(f"SaveZonkHand {message}")
```

There's nothing terribly exciting here; the observed object is set up in the initializer, and when the observer is called, we do *something* – in this example, printing a line. Note that the superclass, `Observer`, isn't actually needed here. The context in which this class is used is sufficient for **mypy** to confirm this class matches the required `Observer` protocol. While we don't need to state that it's an `Observer`, it can help readers to see that this class implements the `Observer` protocol.

We can test the `SaveZonkHand` observer in an interactive console:

```
>>> d = Dice.from_text("6d6")
>>> player = ZonkHandHistory("Bo", d)

>>> save_history = SaveZonkHand(player)
>>> player.attach(save_history)
>>> r1 = player.start()
SaveZonkHand {'player': 'Bo', 'sequence': 1, 'hands': '[[1, 1, 2, 3, 6,
6]]', 'time': ...}
>>> r1
[1, 1, 2, 3, 6, 6]

>>> r2 = player.roll()
SaveZonkHand {'player': 'Bo', 'sequence': 2, 'hands': '[[1, 1, 2, 3, 6, 6],
[1, 2, 2, 6, 6, 6]]', 'time': ...}
>>> r2
[1, 2, 2, 6, 6, 6]
```

(Note that the time is replaced with `...`. The exact value varies every time this is run, making it difficult to use doctest and confirm that output matches our expectations.)

After attaching the observer to the `Inventory` object, whenever we change one of the two observed properties, the observer is called and its action is invoked. Note that our observer tracks a sequence number and includes a timestamp. These are outside the game definition, and are kept separate from the essential game processing by being part of the `SaveZonkHand` observer class.

We can add multiple observers of a variety of classes. Let's add a second observer that has a limited job to check for three pairs and announce it:

```
class ThreePairZonkHand:
    """Observer of ZonkHandHistory"""

    def __init__(self, hand: ZonkHandHistory) -> None:
        self.hand = hand
        self.zonked = False

    def __call__(self) -> None:
        last_roll = self.hand.rolls[-1]
        distinct_values = set(last_roll)
        self.zonked = len(distinct_values) == 3 and all(
            last_roll.count(v) == 2 for v in distinct_values
        )
        if self.zonked:
            print("3 Pair Zonk!")
```

For this example, we omitted naming `Observer` as a superclass. We can trust a tool like **mypy** to note how this class is used and what protocols it must implement. Introducing this new `ThreePairZonkHand` observer means that when we change the state of the hand, there may be two sets of output, one for each observer. The key idea here is that we can easily add totally different types of observers to do different kinds of things – in this case, copying the data as well as checking for a special case in the data.

The Observer pattern detaches the code being observed from the code doing the observing. If we were not using this pattern, we would have had to put code in the `ZonkHandHistory` class to handle the different cases that might come up: logging to the console, updating a database or file, checking for special cases, and so on. The code for each of these tasks would all be mixed in with the core class definition. Maintaining it would be a nightmare and adding new monitoring functionality at a later date would be painful.

The Strategy pattern

The Strategy pattern is a common demonstration of abstraction in object-oriented programming. The pattern implements different solutions to a single problem, each in a different object. The core class can then choose the most appropriate implementation dynamically at runtime.

Typically, different algorithms have different trade-offs. One might be faster than another, but uses a lot more memory, while a third algorithm may be most suitable when multiple CPUs are present or a distributed system is provided.

Here is the Strategy pattern in UML:

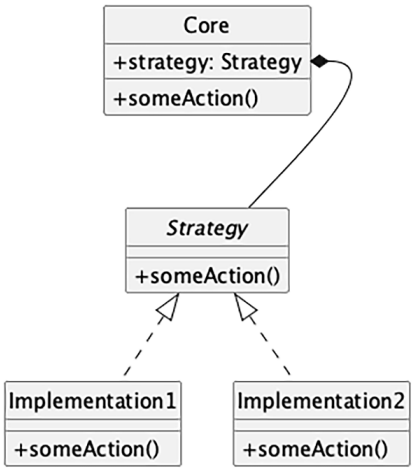


Figure 11.4: Strategy pattern in UML

The **Core** code connecting to the **Strategy** abstraction simply needs to know that it is dealing with some kind of class that fits the Strategy interface for this particular action. Each of the implementations should perform the same task, but in different ways. The implementation interfaces need to be identical, and it's often helpful to leverage an abstract base class to make sure the implementations match.

This idea of a plug-in strategy is also an aspect of the Observer pattern. Indeed, the idea of strategy objects is an important aspect of many of the patterns covered in this chapter. The common idea is to use a separate object to isolate conditional or replaceable processing and delegate the work to the separate object. This works for observables, decorations, and — as we'll see — commands and states, also.

A Strategy example

One common example of the Strategy pattern is sort routines. Over the years, numerous algorithms have been invented for sorting a collection of objects. Quick sort, merge sort, and heap sort are all algorithms with different features, each useful in its own right, depending on the size and type of inputs, how out of order they are, and the requirements of the system.

If we have client code that needs to sort a collection, we could pass it to an object with a `sort()` method. This object may be a `QuickSorter` or `MergeSorter` object, but the result will be the same in either case: a sorted list. The strategy used to do the sorting is abstracted from the calling code, making it modular and replaceable.

Of course, in Python, we typically just call the `sorted()` function or `list.sort()` method and trust that it will do the sorting quickly enough that the details of the TimSort algorithm don't really matter. For details on how amazingly fast TimSort is, see `https://bugs.python.org/file4451/timsort.txt`. While sorting is a helpful concept, it's not the most practical example, so let's look at something different.

As a simpler example of the Strategy design pattern, consider a desktop wallpaper manager. When an image is displayed on a desktop background, it can be adjusted to the screen size in different ways. For example, assuming the image is smaller than the screen, it can be tiled across the screen, centered on it, or scaled to fit. There are other, more complicated strategies that can be used as well, such as scaling to the maximum height or width, combining it with a solid, semi-transparent, or gradient background color, or other manipulations. While we may want to add these strategies later, let's start with a few basic ones.

You'll need to install the `pillow` module from the **PIL** project. If you're using **conda** to manage your virtual environments, use `conda install pillow` to install the library. For tools like **uv**, a command like `uv add pillow` is needed. If you're not using a tool to manage the virtual environment, use `python -m pip install pillow`.

Our Strategy objects need to take two inputs: the image to be displayed, and a tuple of the width and height of the screen. They each return a new image the size of the screen, with the image manipulated to fit according to the given strategy.

Here are some preliminary definitions, including an abstract superclass for all of the strategy variants:

```
import abc
from pathlib import Path
import PIL.Image as image_module
from PIL.Image import Image  # Image class from Image module

type Size = tuple[int, int]


class FillAlgorithm(abc.ABC):
    @abc.abstractmethod
    def make_background(self, img_file: Path, desktop_size: Size) -> Image:
        pass
```

Is this abstraction necessary? This sits right on the fence between too simple to require an abstraction and complex enough that the superclass helps clarify the design. The function signature is kind of complex, with a special type hint to describe the size tuple. For this reason, the abstraction can help check each implementation to be sure all the types match.

Here's our first concrete strategy; this is a fill algorithm that tiles the images:

```
class TiledStrategy(FillAlgorithm):
    def make_background(self, img_file: Path, desktop_size: Size) -> Image:
        in_img = image_module.open(img_file)
        out_img = image_module.new("RGB", desktop_size)
        num_tiles = [o // i + 1 for o, i in zip(out_img.size, in_img.size)]
        for x in range(num_tiles[0]):
            for y in range(num_tiles[1]):
                out_img.paste(
                    in_img,
                    (
                        in_img.size[0] * x,
                        in_img.size[1] * y,
                        in_img.size[0] * (x + 1),
                        in_img.size[1] * (y + 1),
                    ),
                )
        return out_img
```

This works by dividing the output height and width by the input image height and width. The `num_tiles` sequence is a way of doing the same computation to widths and heights. It's a two-tuple computed via a list comprehension to be sure both width and height are processed the same way.

Here's a fill algorithm that centers the image without re-scaling it:

```
class CenteredStrategy(FillAlgorithm):
    def make_background(self, img_file: Path, desktop_size: Size) -> Image:
        in_img = image_module.open(img_file)
        out_img = image_module.new("RGB", desktop_size)
        left = (out_img.size[0] - in_img.size[0]) // 2
        top = (out_img.size[1] - in_img.size[1]) // 2
        out_img.paste(
            in_img,
            (left, top, left + in_img.size[0], top + in_img.size[1]),
        )
        return out_img
```

Finally, here's a fill algorithm that scales the image up to fill the entire screen:

```
class ScaledStrategy(FillAlgorithm):
    def make_background(self, img_file: Path, desktop_size: Size) -> Image:
        in_img = image_module.open(img_file)
        out_img = in_img.resize(desktop_size)
        return out_img
```

Here, we have three strategy subclasses, each using the `PIL.Image` module to perform their task. All the strategy implementations have a `make_background()` method that accepts the same set of parameters. Once selected, the appropriate Strategy object can be called to create a correctly sized version of the desktop image. The `TiledStrategy` class computes the number of input image tiles that would fit in the width and height of the display screen and copies the image into each tile location, repeatedly, without rescaling, so it may not fill the entire space. The `CenteredStrategy` class figures out how much space needs to be left on the four edges of the image to center the image. The `ScaledStrategy` forces the image to the output size, without preserving the original aspect ratio.

Here's an overall object that does resizing, using one of these Strategy classes. The `algorithm` instance variable is filled in when a `Resizer` instance is created:

```
class Resizer:
    def __init__(self, algorithm: FillAlgorithm) -> None:
        self.algorithm = algorithm
```

```
    def resize(self, image_file: Path, size: Size) -> Image:
        result = self.algorithm.make_background(image_file, size)
        return result
```

To be complete, here's a `main` function that builds an instance of the `Resizer` class and applies one of the available Strategy classes:

```
def main() -> None:
    image_file = Path.cwd() / "boat.png"
    tiled_desktop = Resizer(TiledStrategy())
    tiled_image = tiled_desktop.resize(image_file, (1920, 1080))
    tiled_image.show()
```

What's important is the binding of the Strategy instance happens as late as possible in the processing. The decision can be made (and unmade) at any point in the processing. In this example, we've made sure any of the available strategy objects can be plugged into a `Resizer` object at any time. The use of inheritance to provide alternatives and composition to inject any of those alternatives is the essence of the Strategy pattern.

Consider how switching between these options would be implemented without the Strategy pattern. We'd need to put all the code inside one great big method and use an awkward `if` statement to select the expected one. Every time we wanted to add a new strategy, we'd have to make the method even more ungainly.

Strategy in Python

The preceding canonical implementation of the Strategy pattern, while very common in most object-oriented libraries, isn't always ideal in Python. It involves some overheads that aren't really necessary.

These strategy classes each define objects that do nothing but provide a single method. We could just as easily call that function `__call__` and make the object callable directly. Since there is no other data associated with the object, we need do no more than create a set of top-level functions and pass them around as our strategies instead.

Instead of the overheads of an abstract class, we could summarize these strategies with a type hint that describes them as a union of class definitions:

```
type FillAlgorithm_T = TiledStrategy | CenteredStrategy | ScaledStrategy
```

(We put a `_T` suffix on the name to distinguish the base class from the type alias. This kind of naming convention is not recommended at all; it's only required to keep the two declarations from colliding in this book's example files.)

When we do this, we can eliminate all of the references to `FillAlgorithm` as base classes in the class definitions. For example, we'd change `class CenteredStrategy(FillAlgorithm):` to `class CenteredStrategy:`. This means adding a new algorithm will also require updating the `FillAlgorithm_T` type hint.

Because we have a choice between an abstract class and a type hint, the Strategy design pattern seems superfluous. This leads to an odd conversation, starting with *"Because Python has first-class functions, the Strategy pattern is actually unnecessary."* In truth, Python's first-class functions allow us to implement the Strategy pattern in a more straightforward way, without the overhead of class definitions. The pattern is more than the implementation details handed down from other, statically-compiled languages. Knowing the pattern can help us choose a good design for our program, and implement it using the most readable syntax. The Strategy pattern, whether a class or a function, should be used when we need to allow client code or the end user to select from multiple implementations of the same interface at runtime.

There's a bright line separating mixin class definitions from plug-in strategy objects. As we saw in *Chapter 6*, mixin class definitions are created in the source code, and cannot easily be tweaked at runtime. A plug-in strategy object, however, is filled in at runtime, allowing late binding of the strategy. The code tends to be very similar between them, and it helps to write clear docstrings on each class to explain how the various classes fit together.

The Command pattern

When we think about class responsibilities, we can sometimes distinguish "passive" classes that hold objects and maintain an internal state, but don't initiate very much, and "active" classes that reach out into other objects to take action and do things. This is not always a crisp distinction, but it can help separate the relatively passive Observer and the more active Command design patterns. An Observer is notified that something changed. A Command, on the other hand, will be active, making state changes in other objects. We can combine the two aspects, and that's one of the beauties of talking about a software architecture by describing the various patterns that apply to a

class or a relationship among classes.

The Command pattern generally involves a hierarchy of classes that each do something. A Core class can create a command (or a sequence of commands) to carry out actions.

In a way, it's a kind of meta-programming: by creating Command objects that contain a bunch of statements, the design has a higher-level "language" of Command objects.

Here's a UML diagram showing a Core object and a collection of **Commands**:

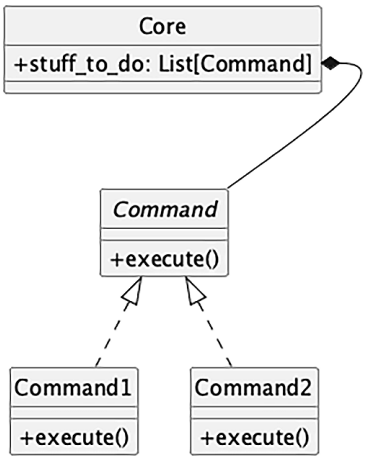


Figure 11.5: Command pattern in UML

This looks similar to the diagrams for the Strategy and Observer patterns because all these patterns rely on delegating work from a Core object to a plug-in object — in this case, a sequence of individual plug-in objects that represent a sequence of commands to perform.

A Command example

As an example, we'lllook at the fancy dice rolling that was omitted from the Decorator pattern example earlier in this chapter. In the earlier example, we had a function, `dice_roller()`, that computed a sequence of random numbers:

```
import random


def dice_roller_ex(request: bytes) -> bytes:
    request_text = request.decode("utf-8")
```

```
    numbers = [random.randint(1, 6) for _ in range(6)]
    response = f"{request_text} = {numbers}"
    return response.encode("utf-8")
```

This isn't very clever; we'd rather handle something a little more sophisticated. We want to be able to make requests using strings like `'3d6'` to mean three six-sided dice, `'3d6+2'` to mean three six-sided dice plus a bonus of two more, and something a little more obscure like `'4d6d1'` to mean "roll four six-sided dice and drop the lowest die." We might want to combine things and write `'4d6d1+2'`, also, to combine dropping the lowest and adding two to the result.

These `d1` and `+2` options at the end can be viewed as a series of commands. There are four common varieties: "drop," "keep," "add," and "subtract." There can be a lot more, of course, to reflect a wide variety of game mechanics and desired statistical distributions, but we'll look at these four commands to modify a batch of dice.

Here's the regular expression we're going to implement:

```
dice_pattern = re.compile(r"(?P<n>\d*)d(?P<d>\d+)(?P<a>(?:[dk+-]\d+)*)")
```

This regular expression can be a little daunting. Some people find the *railroad diagrams* at `https://www.debuggex.com` to be helpful. Here's a depiction as a UML state diagram:

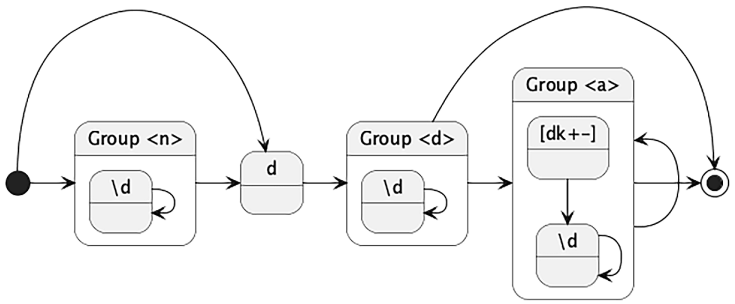


Figure 11.6: Dice-parsing regular expression

This pattern has four parts:

1. The first named group, `(?P<n>*)`, captures a batch of digits for the number of dice, saving this as a group named `n`. This is optional, allowing us to write `d6` instead of `1d6`.
2. The letter "d", which must be present, but isn't captured.

3. The next named group, `(?P<d>+)`, captures the digits for the number of faces on each die, saving this as a group named `d`. If we were very fussy, we might try to limit this to `(4|6|8|10|12|20|100)` to define an acceptable list of regular polyhedral dice (and two common irregular polyhedrons). We didn't provide this short list; instead, we'll accept any sequence of digits.

4. The final named group, `(?P<a>(?:[dk+-]+)*)`, defines a repeating series of adjustments. Each one has a prefix and a sequence of digits – for example, `d1` or `k3` or `+1` or `-2`. We'll capture the whole sequence of adjustments as group `a`, and decompose the parts separately. Each of these parts will become a command, following the Command design pattern.

We will design dice rolling as a sequence of commands. The first command rolls the dice, and then subsequent commands adjust the value of the dice: keeping, dropping, adding, or subtracting.

For example, `3d6+2` means roll three dice (maybe we get ⚃⚃⚂), and add 2 to get 13 in total.

The class, overall, looks like this:

```
import re

class Dice:
    def __init__(self, n: int, d: int, *adj: Adjustment) -> None:
        self.adjustments = [Roll(n, d)] + list(adj)
        self.dice: list[int]
        self.modifier: int

    def roll(self) -> int:
        for a in self.adjustments:
            a.apply(self)
        return sum(self.dice) + self.modifier
```

When we want a new roll of the dice, a `Dice` object applies the individual `Adjustment` objects to create a new roll. We can see one of the kinds of `Adjustment` objects in the `__init__()` method: a `Roll` object. This is put first into a sequence of adjustments; after that, any additional adjustments are processed in order. Each adjustment is another kind of command.

Here are the kinds of adjustment commands that change the state of a `Dice` object:

```
import abc
import random
```

```
class Adjustment(abc.ABC):
    def __init__(self, amount: int) -> None:
        self.amount = amount

    @abc.abstractmethod
    def apply(self, dice: "Dice") -> None:
        ...


class Roll(Adjustment):
    def __init__(self, n: int, d: int) -> None:
        self.n = n
        self.d = d

    def apply(self, dice: "Dice") -> None:
        dice.dice = sorted(random.randint(1, self.d) for _ in range(self.n))
        dice.modifier = 0
```

```
class Drop(Adjustment):
    def apply(self, dice: "Dice") -> None:
        dice.dice = dice.dice[self.amount :]


class Keep(Adjustment):
    def apply(self, dice: "Dice") -> None:
        dice.dice = dice.dice[: self.amount]


class Plus(Adjustment):
    def apply(self, dice: "Dice") -> None:
        dice.modifier += self.amount


class Minus(Adjustment):
    def apply(self, dice: "Dice") -> None:
        dice.modifier -= self.amount
```

An instance of the `Roll()` class sets the values of the dice and the modifier attribute of a `Dice` instance. The other `Adjustment` objects either remove some dice or change the modifier. The operations depend on the dice being sorted. That makes it easy to drop the worst or keep the best

via slice operations. Because each adjustment is a command, they make adjustments to the overall state of the dice that were rolled.

The missing piece is translating the string dice expression into a sequence of `Adjustment` objects. We've made this a `@classmethod` of the `Dice` class. This lets us use `Dice.from_text()` to create a new `Dice` instance. It also provides the subclass as the first parameter value, `cls`, making sure that each subclass creates proper instances of itself, not this parent class.

Here's the definition of this method:

```
    @classmethod
    def from_text(cls, dice_text: str) -> "Dice":
        dice_pattern =
        re.compile(r"(?P<n>\d*)d(?P<d>\d+)(?P<a>(?:[dk+-]\d+)*)")
        adjustment_pattern = re.compile(r"([dk+-])(\d+)")
        adj_class: dict[str, type[Adjustment]] = {
            "d": Drop,
            "k": Keep,
            "+": Plus,
            "-": Minus,
        }

        if (dice_match := dice_pattern.match(dice_text)) is None:
            raise ValueError(f"Error in {dice_text!r}")

        n = int(dice_match.group("n")) if dice_match.group("n") else 1
        d = int(dice_match.group("d"))
        adjustment_matches =
        adjustment_pattern.finditer(dice_match.group("a") or "")
        adjustments = [
            adj_class[a.group(1)](int(a.group(2))) for a in
            adjustment_matches
        ]
        return cls(n, d, *adjustments)
```

The overall `dice_pattern` is applied first and the result is assigned to the `dice_match` variable. If the result is a `None` object, the pattern didn't match, and we can't do much more than raise a `ValueError` exception and give up. The `adjustment_pattern` is used to decompose the string of adjustments in the suffix of the dice expression. A list comprehension is used to create a list of objects from the `Adjustment` class definitions.

Each `Adjustment` class is a separate command. The `Dice` class will inject an additional command, `Roll`, that starts the processing by simulating a roll of the dice. Then the adjustment commands can apply their individual changes to the initial roll.

This design allows us to manually create an instance like this:

```
>>> d = Dice(4, D6, Keep(3), Plus(2))
>>> d.roll()
10
```

The first two positional parameters define the special `Roll` command. The remaining parameters can include any number of further adjustments. In this case, there are two: a `Keep(3)` command and a `Plus(2)` command. The alternative is to parse text, like this: `dice.Dice.from_text("4d6k3+2")`. This will build the `Roll` command and the other `Adjustment` commands. Each time we want a new roll of the dice, the sequence of commands is executed, rolling the dice and then adjusting that roll to give a final outcome.

The State pattern

The State pattern is structurally similar to the Strategy pattern, but its intent and purpose are very different. The goal of the State pattern is to represent state transition systems: systems where an object's behavior is constrained by the state it's in, and some of the behavior includes transitions from state to state.

One way to make this work is to define a manager or context class that provides an interface for the various states. Internally, this class is a container for an object that represents the current state. A state object can change the state of the manager, transition to a different state object.

In the next page, we've shown how it looks in UML.

The State pattern decomposes the problem into two types of classes: the **Core** class and multiple State classes. The **Core** class maintains the current state, and forwards actions to a current state object. The State objects are typically hidden from any other objects that are calling the **Core** object; they act like a black boxes that happen to perform state management internally.

A State example

One of the mostcompelling state-specific processing examples is parsing text. When we write a regular expression, we're detailing a series of alternative state changes used to match a pattern

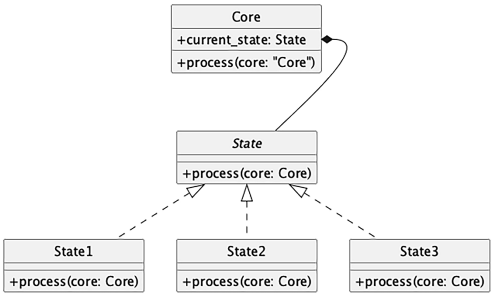


Figure 11.7: State pattern in UML

against a sample piece of text. At a higher level, parsing the text of a programming language or a markup language is also highly stateful work. Markup languages like XML, HTML, YAML, TOML, or even reStructuredText and Markdown all have stateful rules for what is allowed next and what is not allowed next.

We'll look at a relatively simple language that crops up when solving **Internet of Things** (**IoT**) problems. The data stream from a GPS receiver is an interesting problem. Parsing statements in this language is an example of the State design pattern. The language used by GPS devices to report positions is the NMEA 0183 language from the National Marine Electronics Association.

The output from a GPS antenna is a stream of bytes that form a sequence of "sentences." Each sentence starts with `$`, includes printable characters in the ASCII encoding, and ends with a carriage return and a newline character. A GPS device's output includes a number of different kinds of sentences, including the following:

- GPRMC – recommended minimum data
- GPGGA – global position
- GPGLL – latitude and longitude
- GPGSV – satellites in view
- GPGSA – active satellites

There are many, many more messages available, and they come out of the antenna device at a pace that can be bewildering. They all have a common format, however, making them easy to validate and filter so we can use the good ones, and ignore the ones that aren't providing useful information for our specific application. We can also reject messages that are incomplete when our IoT device is first powered on, or suffer from some other problem.

A typical message looks like this:

```
$GPGLL,3723.2475,N,12158.3416,W,161229.487,A,A*41
```

This sentence has the structure shown in *Table 11.1.*

`$`	Starts the sentence
`GPGLL`	The "talker," GP, and the type of message, GLL
`3723.2475`	Latitude, 37°23.2475
`N`	North of the equator
`12158.3416`	Longitude, 121°58.3416
`W`	West of the 0° meridian
`161229.487`	The timestamp in UTC: 16:12:29.487
`A`	Status, `A`=valid, `V`=not valid
`A`	Mode, `A`=Autonomous, `D`=DGPS, `E`=DR
`*`	Ends the sentence, starts the checksum
`41`	Hexadecimal checksum of the text (excluding the `$` and `*` characters)

Table 11.1: Example GPS Sentence

With a few exceptions, all the messages from a GPS will have a similar pattern. The exceptional messages will start with `!`, and our design should quietly ignore them.

When building IoT devices, we need to be aware of two complicating factors:

1. Things aren't very reliable, meaning our software must be prepared for broken or incomplete messages.
2. The devices are tiny and some common Python techniques that work on a large, general-purpose laptop computer won't work well in a tiny Circuit Playground Express chip with only 32K of memory.

What we need to do, then, is to read and validate the message as the bytes arrive. This saves time (and memory) when ingesting data. Because there's a defined upper bound of 82 bytes for these GPS messages, we can use Python `bytearray` structures as a place to process the bytes of a message.

The process for reading a message has a number of distinct states. The following state transition diagram shows the available state changes:

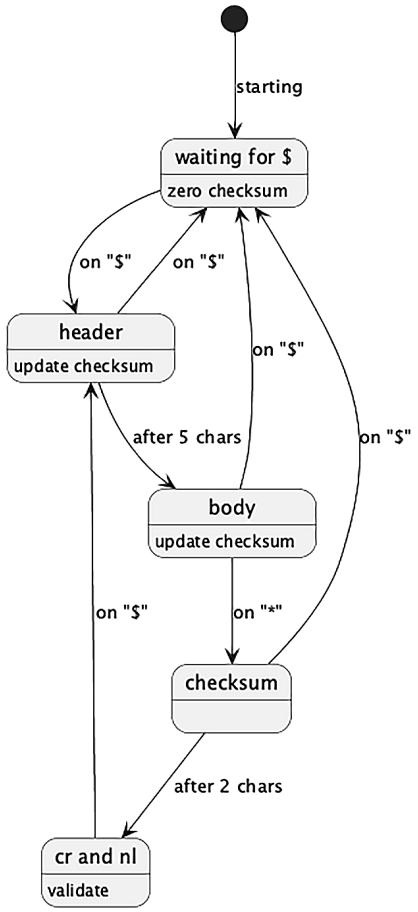


Figure 11.8: State transitions to parse NMEA sentences

We start in a state of waiting for the next `$`. As noted previously, it's best to assume that IoT devices have loose wires and power problems. (Some people can solder really well, so unreliability may not be as common for them as it is for the authors.)

Once we've received the `$`, we'll transition to a state of reading the five-character header. If, at any time, we get another `$`, it means we lost some bytes somewhere; the message we were working on was incomplete, and we need to start over again. Once we have all five characters with the message name, we can transition to reading the message body. This will have up to 73 more bytes. When we receive a `*`, it tells us we're at the end of the body. Again, if we see a `$` along the way, it means something's wrong and we should restart.

The final two bytes (after the *) represent a hexadecimal value that should equal the computed checksum of the preceding message (header and body). If the checksum is good, the message can be used by the application. There will be one or more "whitespace" characters — usually the carriage return and newline characters — at the end of the message.

We can imagine each of these states as an extension of the following class:

```
class NMEA_State:
    def __init__(self, message: "Message") -> None:
        self.message = message

    def feed_byte(self, input: int) -> "NMEA_State":
        return self

    def valid(self) -> bool:
        return False

    def __repr__(self) -> str:
        return f"{self.__class__.__name__}({self.message})"
```

We've defined each state to work with a `Message` object. Some reader object will feed a byte to the current state, which will do something with the byte (usually save it) and return the next state. The exact behavior depends on the byte received; for example, most states will reset the message buffer to empty and transition to the `Header` state when they receive a `$`. Most states will return `False` for the `valid()` function. One state, however, will validate a complete message, and possibly return `True` for the `valid()` function. This only happens when the checksum is correct.

For the purists, the class name doesn't strictly follow PEP-8. It's challenging to include abbreviations or acronyms and keep a properly camel-cased name. It seems like `NmeaState` isn't as clear. While a compromise class name might be `NMEAState`, the clash between abbreviations and class name seems confusing. We prefer to cite "*A foolish consistency is the hobgoblin of little minds…*" in this specific case. Keeping the class hierarchy internally consistent is more important than the full PEP-8 level of consistency.

The `Message` object is a wrapper around two `bytearray` structures where we accumulate the content of the message:

```
class Message:
    def __init__(self) -> None:
        self.body = bytearray(80)
        self.checksum_source = bytearray(2)
        self.body_len = 0
        self.checksum_len = 0
        self.checksum_computed = 0

    def reset(self) -> None:
        self.body_len = 0
        self.checksum_len = 0
        self.checksum_computed = 0

    def body_append(self, input: int) -> int:
        self.body[self.body_len] = input
        self.body_len += 1
        self.checksum_computed ^= input
        return self.body_len

    def checksum_append(self, input: int) -> int:
        self.checksum_source[self.checksum_len] = input
        self.checksum_len += 1
        return self.checksum_len

    @property
    def valid(self) -> bool:
        return (
            self.checksum_len == 2
            and int(self.checksum_source, 16) == self.checksum_computed
        )
```

This definition of the `Message` class encapsulates much of what's important about each sentence that comes from the GPS device. We defined a method, `body_append()`, for accumulating bytes in the body, and accumulating a checksum of those bytes. In this case, the `^` operator is used to compute the checksum. This is a real Python operator; it's the bit-wise exclusive OR. An exclusive OR, XOR, means "one or the other but not both." You can see it in action with an expression like `bin(ord(b'a') ^ ord(b'z'))`. The bits in `b'a'` are `0b1100001`. The bits in `b'z'` are `0b1111010`. Applying "one or the other but not both" to the bits, the XOR is `0b0011011`.

Here's the reader that builds valid `Message` objects by undergoing a number of state changes as bytes are received:

```
from typing import Iterable, Iterator, cast


class Reader:
    def __init__(self) -> None:
        self.buffer = Message()
        self.state: NMEA_State = Waiting(self.buffer)

    def read(self, source: Iterable[bytes]) -> Iterator[Message]:
        for byte in source:
            self.state = self.state.feed_byte(cast(int, byte))
            if self.buffer.valid:
                yield self.buffer
                self.buffer = Message()
                self.state = Waiting(self.buffer)
```

The initial state is an instance of the `Waiting` class, a subclass of `NMEA_State`. The `read()` method consumes one byte from the input, and then hands it to the current `NMEA_State` object for processing. The state object may save the byte or may discard it, the state object may transition to another state, or it may return the current state. If the state's `valid()` method is `True`, the message is complete, and we can yield it for further processing by our application.

Note that we're reusing a `Message` object's byte arrays until it's complete and valid. This avoids allocating and freeing a lot of objects while ignoring incomplete messages on a noisy line. This is not typical for Python programs on large computers. In some applications, we don't need to save the original message, but only need to save the values of a few fields, further reducing the amount of memory used.

To reuse the buffers in the `Message` object, we need to make sure it's not part of any specific `State` object. We've made the current `Message` object part of the overall `Reader`, and provided the working `Message` object to each `State` as an argument value.

Now that we've seen the context, here are the classes to implement the various states for an incomplete message. We'll start with the state of waiting for the initial `$` to begin a message. When a `$` is seen, the parser transitions to a new state, `Header`. Otherwise, it ignores the character. The class definition looks like this:

```
class Waiting(NMEA_State):
    def feed_byte(self, input: int) -> NMEA_State:
        if input == ord(b"$"):
            return Header(self.message)
        return self
```

When we're in the `Header` state, we've seen the `$`, and we're waiting for the five characters that identify the "talker" (`GP`) and the sentence type (`GLL`). We'll accumulate bytes until we get five of them, and then transition to the `Body` state. The presence of a `$` in the input tells us bytes were lost, and we need to begin again with a new message's header. It looks like this:

```
class Header(NMEA_State):
    def __init__(self, message: "Message") -> None:
        self.message = message
        self.message.reset()

    def feed_byte(self, input: int) -> NMEA_State:
        if input == ord(b"$"):
            return Header(self.message)
        size = self.message.body_append(input)
        if size == 5:
            return Body(self.message)
        return self
```

The `Body` state is where we accumulate the bulk of the message. For some applications, we may want to apply additional processing on the header and transition back to waiting for headers when we receive a message type we don't want. This can shave off a little bit of processing time when dealing with devices that produce a lot of data.

When the `*` arrives, the body is complete, and the next two bytes must be part of the checksum. This means transitioning to the `Checksum` state:

```
class Body(NMEA_State):
    def feed_byte(self, input: int) -> NMEA_State:
        if input == ord(b"$"):
            return Header(self.message)
        if input == ord(b"*"):
            return Checksum(self.message)
        self.message.body_append(input)
```

```
        return self
```

The `Checksum` state is similar to accumulating bytes in the `Header` state: we're waiting for a specific number of input bytes. After the checksum, most messages are followed by ASCII `r` and `n` characters. If we receive either of these, we transition to an `End` state where we can gracefully ignore these excess characters:

```
class Checksum(NMEA_State):
    def feed_byte(self, input: int) -> NMEA_State:
        if input == ord(b"$"):
            return Header(self.message)
        if input in {ord(b"\n"), ord(b"\r")}:
            # Incomplete checksum... Will be invalid.
            return End(self.message)
        size = self.message.checksum_append(input)
        if size == 2:
            return End(self.message)
        return self
```

The `End` state has an additional feature: it overrides the default `valid()` method. For all other states, the `valid()` method is `False`. Once we've received a complete message, this state's class definition changes the validity rule: we now depend on the `Message` class to compare the computed checksum with the final checksum bytes to tell us if the message is valid:

```
class End(NMEA_State):
    def feed_byte(self, input: int) -> NMEA_State:
        if input == ord(b"$"):
            return Header(self.message)
        elif input not in {ord(b"\n"), ord(b"\r")}:
            return Waiting(self.message)
        return self

    def valid(self) -> bool:
        return self.message.valid
```

This state-oriented change in behavior is one of the best reasons for using this design pattern. Instead of a complex set of `if` conditions to decide if we have a complete message and the message has all the right parts and punctuation marks, we've refactored the complexity into a number of

individual states and the rules for transition from state to state. This leads us to only checking validity when we've received `$`, five characters, a body, `*`, two more characters, and confirmed the checksum is correct.

Here's a test case to show how this works:

```
>>> message = b'''
... $GPGGA,161229.487,3723.2475,N,12158.3416,W,1,07,1.0,9.0,M,,,,0000*18
... $GPGLL,3723.2475,N,12158.3416,W,161229.487,A,A*41
... '''
>>> rdr = Reader()
>>> result = list(rdr.read(message))
>>> result
[Message(bytearray(b'GPGGA,161229.487,3723.2475,N,12158.3416,W,1,07,1.0,9.0,M,,,,0000'),
bytearray(b'18'), computed=18),
Message(bytearray(b'GPGLL,3723.2475,N,12158.3416,W,161229.487,A,A'),
bytearray(b'41'), computed=41)]
>>> result[0].message()
b'$GPGGA,161229.487,3723.2475,N,12158.3416,W,1,07,1.0,9.0,M,,,,0000*18'
>>> result[1].message()
b'$GPGLL,3723.2475,N,12158.3416,W,161229.487,A,A*41'
```

We've copied two example sentences from the SiRF NMEA Reference Manual, revision 1.3, to be sure our parsing was correct. See `https://www.sparkfun.com/products/13750` for more information on GPS IoT devices. See `http://aprs.gids.nl/nmea/` for additional examples and details. Feel free to try sentences with bad headers, and bad checksums.

It's often helpful to use state transitions when parsing complex messages because we can refactor the validation into individual state definitions and state transition rules.

State versus Strategy

TheState pattern looks very similar to the Strategy pattern; indeed, the UML diagrams for the two are identical. The implementation, too, is identical. We could even have written our states as first-class functions instead of wrapping them in objects, as was suggested in the section on the Strategy pattern earlier in this chapter.

These two patterns are similar because they both delegate work to other objects. This decomposes a complex problem into several closely related but simpler problems. The result is a composition of distinct objects.

The Strategy pattern is used to choose an algorithm at runtime; generally, only one of those algorithms is going to be chosen for a particular use case. The idea here is to provide an implementation choice at runtime, as late in the design process as possible. Strategy class definitions are rarely aware of other implementations; each strategy generally stands alone.

The State pattern, on the other hand, is designed to allow switching between different states dynamically, as some process evolves. In our example, the state changed as bytes were consumed and an evolving set of validity conditions were satisfied. State definitions are generally defined as a group with an ability to switch among the various state objects.

To an extent, the `End` state used to parse an NMEA message has both State pattern features and Strategy pattern features. Because the implementation of the `valid()` method is different from other states, this reflects a different strategy for determining the validity of a sentence.

The Singleton pattern

The Singleton pattern is a source of some controversy; many have accused it of being an *anti-pattern*, a pattern that should be avoided, not promoted. In Python, if someone is using the Singleton pattern, they're almost certainly doing something wrong, probably because they're coming from a more restrictive programming language.

So, why discuss it at all? The basic idea behind the Singleton pattern is to allow exactly one instance of a certain object to exist. A Python example of this is the `None` object, the one — and only — instance of the `NoneType`.

Generally, when a singleton class is used, each collaborator requests an instance of the class. The class makes sure that the one-and-only instance is always returned. The UML diagram doesn't fully describe it, but here it is for completeness:

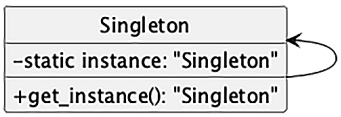


Figure 11.9: Singleton pattern in UML

In some OO programming environments, singletons are enforced by making the constructor private (so no one can create additional instances of it), and then providing a static method to retrieve the single instance. This method creates a new instance the first time it is called, and then returns that

same instance for all subsequent calls.

Singleton implementation

Python doesn't have private constructors, but for this purpose, we can try to use the `__new__()` class method to ensure that only one instance is ever created:

```
from typing import Any


class OneOnly:
    _singleton = None
    def __new__(cls, *args: Any, **kwargs: Any) -> "OneOnly":
        if not cls._singleton:
            cls._singleton = super().__new__(cls, *args, **kwargs)
        return cls._singleton
```

When `__new__()` is called, it normally constructs a new instance of the requested class. This replacement method first checks whether our singleton instance has been created; if not, we create it using a `super` call. Note that there are potential issues with thread safety here, and we're assuming a single-threaded application. Whenever the constructor on `OneOnly` is called, it always returns the exact same instance:

```
>>> o1 = OneOnly()
>>> o2 = OneOnly()
>>> o1 == o2
True
>>> id(o1) == id(o2)
True
```

The two objects have the same value for the `id()` function; thus, they are the same object. This particular implementation isn't very transparent, since it's not obvious that the special method is being used to create a singleton object.

We don't actually need this kind of class. Python provides two built-in Singleton patterns we can leverage. Rather than inventsomething hard to read, there are two built-in choices:

- A Python module *is* a singleton. One `import` will create a module. All subsequent attempts to import the module return the one-and-only singleton instance of the module. In places where an application-wide configuration file or cache is required, make this part of a distinct

module. Library modules like `logging`, `random`, and even `re` have module-level singleton caches. Because this offers the advantage of thread-safe operation, we'll look at using module-level variables next.

- A Python class definition can also be pressed into service as a singleton. A class can only be created once in a given namespace. Consider using a class with class-level attributes as a singleton object. This means defining methods with the `@staticmethod` decorator because there will never be an instance created, and there's no `self` variable.

To use module-level variables instead of a complex Singleton pattern, we instantiate the class after we've defined it. We can improve our State pattern implementation from earlier on to use singleton objects for each of the states. Instead of creating a new object every time we change states, we can create a collection of module-level variables that are always accessible.

We'll make a small but very important design change, also. In the preceding examples, each state has a reference to the `Message` object that is being accumulated. This required us to provide the `Message` object as part of constructing a new `NMEA_State` object. We used code like `return Body(self.message)` to switch to a new state, defined by the `Body` class, while working on the same `Message` instance.

If we don't want to create (and recreate) state objects, we need to provide `Message` as an argument to the relevant methods.

Here's the revised `NMEA_State` class:

```
import abc

class NMEA_State(abc.ABC):
    @staticmethod
    def enter(message: "Message") -> None:
        pass

    @staticmethod
    @abc.abstractmethod
    def feed_byte(message: "Message", input: int) -> "type[NMEA_State]":
        ...

    @staticmethod
    def valid(message: "Message") -> bool:
        return False
```

This variant on the `NMEA_State` class doesn't have any instance variables. All the methods work with argument values passed in by a client; this makes them static methods. The class is a bundle of related methods, and a singleton. Since static methods don't process the internal state of an object, there's no `self` parameter, either. Parts of the reader will work with `type[NMEA_State]` — the class object itself — instead of working with individual `NMEA_State` instances. Here are the individual state definitions:

```
class Waiting(NMEA_State):
    @staticmethod
    def feed_byte(message: "Message", input: int) -> type[NMEA_State]:
        if input == ord(b"$"):
            return Header
        return Waiting


class Header(NMEA_State):
    @staticmethod
    def enter(message: "Message") -> None:
        message.reset()

    @staticmethod
    def feed_byte(message: "Message", input: int) -> type[NMEA_State]:
        if input == ord(b"$"):
            return Header
        size = message.body_append(input)
        if size == 5:
            return Body
        return Header


class Body(NMEA_State):
    @staticmethod
    def feed_byte(message: "Message", input: int) -> type[NMEA_State]:
        if input == ord(b"$"):
            return Header
        if input == ord(b"*"):
            return Checksum
        message.body_append(input)
        return Body


class Checksum(NMEA_State):
```

```
    @staticmethod
    def feed_byte(message: "Message", input: int) -> type[NMEA_State]:
        if input == ord(b"$"):
            return Header
        if input in {ord(b"\n"), ord(b"\r")}:
            # Incomplete checksum... Will be invalid.
            return End
        size = message.checksum_append(input)
        if size == 2:
            return End
        return Checksum


class End(NMEA_State):
    @staticmethod
    def feed_byte(message: "Message", input: int) -> type[NMEA_State]:
        if input == ord(b"$"):
            return Header
        elif input not in {ord(b"\n"), ord(b"\r")}:
            return Waiting
        return End

    @staticmethod
    def valid(message: "Message") -> bool:
        return message.valid
```

The important feature here is that each state transition is simply the object class, a subclass of `NMEA_State`, not an instance of the subclass. The type `type[NMEA_State]`.

Note how the special message reset processing was refactored out of the `Header` class. In the version where each state has an `__init__()`, we can explicitly evaluate `Message.reset()` when entering the `Header` state. Since we're not creating new state objects in this design, we need a way to handle the special case of entering a new state, and performing an `enter()` method one time only to do initialization or setup. This requirement leads to a small change in the `Reader` class:

```
from typing import Iterable, Iterator, cast


class Reader:
    def __init__(self) -> None:
        self.buffer = Message()
```

```
        self.state: type[NMEA_State] = Waiting

    def read(self, source: Iterable[bytes]) -> Iterator[Message]:
        for byte in source:
            new_state = self.state.feed_byte(self.buffer, cast(int, byte))
            if self.buffer.valid:
                yield self.buffer
                # Ready for the next...
                self.buffer = Message()
                new_state = Waiting
            if new_state != self.state:
                # self.state.exit()  # A common extension
                new_state.enter(self.buffer)
                self.state = new_state
```

We don't trivially replace the value of the `self.state` instance variable with the result of the `self.state.feed_byte()` evaluation. Instead, we compare the previous value of `self.state` with the next value, `new_state`, to see if there was a state change. When there is a state change, then we need to evaluate `enter()` on the new state, to allow the state change to do any required one-time initialization.

In this example, we aren't wasting memory creating a bunch of new instances of each state object that must later be garbage collected. Instead, we are reusing the Singleton state objects for each piece of the incoming data stream. Even if multiple parsers are running at once, only these state objects need to be used. The stateful message data is kept separate from the state processing rules in each state object.

We've combined two patterns, each with different purposes. The State pattern covers how processing is completed. The Singleton pattern covers how object instances are managed. Many software designs involve numbers of overlapping and complementary patterns.

Recall

The world of software design is full of good ideas. The really good ideas get repeated and form repeatable patterns. Knowing — and using — these patterns of software design can save the developer from burning a lot of brain calories trying to reinvent something that's been developed already. In this chapter, we looked at a few of the most common patterns:

- The Decorator pattern is used in the Python language to add features to functions or classes. We can define decorator functions and apply them directly, or use the `@` syntax to apply a decorator to another function.
- The Observer pattern can simplify writing GUI applications. It can also be used in non-GUI applications to formalize relationships between objects that change state, and objects that display or summarize or otherwise use the state information.
- The Strategy pattern is central to a lot of object-oriented programming. We can decompose large problems into containers with the data and strategy objects that help with processing the data. The Strategy object is a kind of "plug-in" to another object. This gives us ways to adapt, extend, and improve processing without breaking all the code we wrote when we make a change.
- The Command pattern is a handy way to summarize a collection of changes that are applied to other objects. It's really helpful in a web services context where external commands arrive from web clients.
- The State pattern is a way to define processing where there's a change in state and a change in behavior. We can often push unique or special-case processing into state-specific objects, leveraging the Strategy pattern to plug in state-specific behavior.
- The Singleton pattern is used in the rare cases where we need to be sure there is one and only one of a specific kind of object. It's common, for example, to limit an application to exactly one connection to a central database.

These design patterns help us organize complex collections of objects. Knowing a number of patterns can help the developer visualize a collection of cooperating classes, and allocate their responsibilities. It can also help developers talk about a design: when they've both read the same books on design patterns, they can refer to the patterns by name and skip over long descriptions.

Exercises

While writing the examples for this chapter, the authors discovered that it can be very difficult, and extremely educational, to come up with good examples where specific design patterns *should* be used. Instead of going over current or old projects to see where you can apply these patterns, as we've suggested in previous chapters, it's time to start thinking about the patterns and different situations where they might come up. Try to think outside your own experiences. If your current projects are in the banking business, consider how you'd apply these design patterns in a retail or

point-of-sale application. If you normally write web applications, think about using design patterns while writing a compiler.

Look at the Decorator pattern and come up with some good examples of when to apply it. Focus on the pattern itself, not the Python syntax we discussed. It's a bit more general than the actual pattern. The special syntax for decorators is, however, something you may want to look for places to apply in existing projects too.

What are some good areas to use the Observer pattern? Why? Think about not only how you'd apply the pattern, but how you would implement the same task without using Observer. What do you gain, or lose, by choosing to use it?

Consider the difference between the Stategy and State patterns. Implementation-wise, they look very similar, yet they have different purposes. Can you think of cases where the patterns could be interchanged? Would it be reasonable to redesign a State-based system to use Strategy instead, or vice versa? How different would the design actually be?

In the image filler example, we suggested that the different fill algorithms are — essentially — functions. Each is a `Callable[[Image, Size], Image]` function. This pulls one redundant line of code out of the definition of the `make_background()` function: the line that reads an image file from a Path. The rest of the `make_background()` function can be renamed to be a function that transforms and Image and size to another Image. After making this change, the `resizer()` class also needs some changes. How do these two implementations compare? Which seems more clear? Which seems easier to extend and customize?

In the dice-rolling example, we parsed a simple expression to create a few commands. There are more options possible. See *Roll20 Dice Reference* (`https://help.roll20.net/hc/en-us/articles/360037773133-Dice-Reference#DiceReference-Roll20DiceSpecif`) for some really sophisticated syntax for describing dice and dice games. To implement this, there are two changes that need to be made. First, design the command hierarchy for all of these options. After that, write a regular expression to parse a more complex dice-rolling expression and execute all of the commands present.

We've noted that Singleton objects can be built using Python module variables. It's sometimes helpful to compare the performance of the two different NMEA message processors. If you don't have a GPS chip with a USB interface laying around, you can search the internet for NMEA example messages to parse. `http://aprs.gids.nl/nmea/` is a good source of examples. There's a trade-off question between the potential confusion of module variables and the performance of the application.

It's helpful to have performance data to support the lessons you've learned.

Summary

This chapter discussed several common design patterns in detail, with examples and UML diagrams. The Decorator pattern is often implemented using Python's more generic decorator syntax. The Observer pattern is a useful way to decouple events from actions taken on those events. The Strategy pattern allows different algorithms to be chosen to accomplish the same task. The Command pattern helps us design active classes that share a common interface but carry out distinct actions. The State pattern looks similar to the Strategy pattern but is used instead to represent systems that can move between different states using well-defined actions. The Singleton pattern, popular in some statically typed languages, is almost always an anti-pattern in Python.

In the next chapter, we'll wrap up our discussion of design patterns.

12

Advanced Design Patterns

In this chapter, we will introduce several more design patterns. Once again, we'll cover the canonical examples as well as any common alternative implementations in Python. We'll be discussing the following:

- The Adapter pattern
- The Façade pattern
- Lazy initialization and the Flyweight pattern
- The Abstract Factory pattern
- The Composite pattern
- The Template pattern

Consistent with the practice in *Design Patterns: Elements of Reusable Object-Oriented Software*, we'll capitalize the pattern names.

We'll begin with the Adapter pattern. This is often used to provide a needed interface around an object with a design that doesn't — quite — fit our needs.

The Adapter pattern

Unlike most of the patterns we reviewed in the previous chapter, the Adapter pattern is designed to interact with existing code. We would not design a brand new set of objects that implement the Adapter pattern. Adapters are used to allow two preexisting objects to work together, even if their interfaces are not compatible. Like the display adapters that allow you to plug your Micro USB charging cable into a USB-C phone, an adapter object sits between two different interfaces, translating between them. The adapter object's sole purpose is to perform this translation. Adapting may entail a variety of tasks, such as converting arguments to a different format, rearranging the order of arguments, calling a differently named method, or supplying default arguments.

In structure, the Adapter pattern is similar to a simplified Decorator pattern. A decoration for a class typically provides the same interface for the class it wraps, whereas adapters map between two different interfaces. This is depicted in UML form in the following diagram:

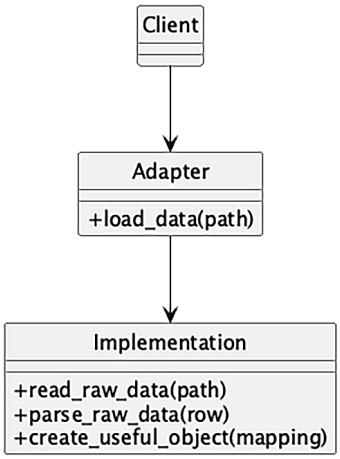


Figure 12.1: Adapter pattern

Here, a client object, an instance of **Client**, needs to collaborate with another class to do something useful. In this example, we're using `load_data()` as a concrete example of a method that requires an adapter.

We already have this perfect class, named **Implementation**, that does everything we want (and to avoid duplication, we don't want to rewrite it!). This perfect class has one problem: it requires a complex sequence of operations using methods called `read_raw_data()`, `parse_raw_data()`, and `create_useful_object()`. The **Adapter** class implements an easy-to-use `load_data()` interface

that hides the complexity of the existing interface provided by the **Implementation** class.

The advantage of this design is that the code that maps from the hoped-for interface to the actual interface is all in one place, the Adapter class. The alternative would require putting the code into the client, cluttering it up with possibly irrelevant implementation details. If we had multiple kinds of clients, we'd have to perform the complex `load_data()` processing in multiple places whenever any of those clients needed to access the `Implementation` class.

An Adapter example

Imagine we have the following pre-existing class, which takes string timestamps in the format `HHMMSS.S` and calculates useful floating-point intervals from those strings:

```
class TimeSince:
    """Expects time as six digits, no punctuation."""

    def parse_time(self, time: str) -> tuple[float, float, float]:
        return (
            float(time[0:2]),
            float(time[2:4]),
            float(time[4:]),
        )

    def __init__(self, starting_time: str) -> None:
        self.hr, self.min, self.sec = self.parse_time(starting_time)
        self.start_seconds = ((self.hr * 60) + self.min) * 60 + self.sec

    def interval(self, log_time: str) -> float:
        log_hr, log_min, log_sec = self.parse_time(log_time)
        log_seconds = ((log_hr * 60) + log_min) * 60 + log_sec
        return log_seconds - self.start_seconds
```

This class handles string to time-interval conversion. Since we have this class in the application already, it has unit test cases and works nicely.

Here's an example showing how this class works:

```
>>> ts = TimeSince("000123")  # Log started at 00:01:23
>>> ts.interval("020304")
7301.0
>>> ts.interval("030405")
```

```
10962.0
```

Working with these unformatted times is a little awkward, but a number of **Internet of Things** (**IoT**) devices provide these kinds of time strings, separated from the rest of the date. For example, look at the NMEA 0183 format messages from a GPS device, where dates and times are unformatted strings of digits.

We have an old log from one of these devices, apparently created years ago. We want to analyze this log for the sequence of messages that occur after each ERROR message. We'd like the exact times, relative to the ERROR, as part of our root cause problem analysis.

Here's some of the log data we're using for testing:

```
>>> data = [
...     ("000123", "INFO", "Gila Flats 1959-08-20"),
...     ("000142", "INFO", "test block 15"),
...     ("004201", "ERROR", "intrinsic field chamber door locked"),
...     ("004210.11", "INFO", "generator power active"),
...     ("004232.33", "WARNING", "extra mass detected")
... ]
```

It's difficult to compute the time interval between the ERROR and the WARNING message. It's not impossible; many of us have enough fingers to do the computation. But it would be better to show the log with relative times instead of absolute times. Here's an outline of the log formatter we'd like to use. This code, however, has a problem that we've marked with `???`:

```
class LogProcessor:
    def __init__(self, log_entries: list[tuple[str, str, str]]) -> None:
        self.log_entries = log_entries

    def report(self) -> None:
        first_time, first_sev, first_msg = self.log_entries[0]
        for log_time, severity, message in self.log_entries:
            if severity == "ERROR":
                first_time = log_time
            interval = ### Need to compute an interval ???
            print(f"{interval:8.2f} | {severity:7s} {message}")
```

This `LogProcessor` class seems like the right thing to do. It iterates through the log entries, resetting

the `first_time` variable on each occurrence of an ERROR line. This makes sure that the log shows offsets from the error, saving us from having to do a lot of math to work out exactly what happened and when.

But, we have a problem. We'd really like to reuse the `TimeSince` class. However, it doesn't simply compute an interval between two values. We have several options to address this scenario:

- We could rewrite the `TimeSince` class to work with a pair of time strings. This runs a small risk of breaking something else in our application. We sometimes call this additional breakage the **splash radius** of a change — how many other things get wet when we drop a boulder into the swimming pool? The Open/Closed design principle (one of the SOLID principles; see *Clean Code in Python* (`https://subscription.packtpub.com/book/application_development/9781788835831/4`) for more background) suggests a class should be open to extension but closed to this kind of modification. If this class was downloaded from PyPI, we may not want to change its internal structure because then we wouldn't be able to use any subsequent releases. We need an alternative to tinkering inside another class.
- We could use theclass as it is, and whenever we want to calculate the intervals between an ERROR and subsequent log lines, we create a new `TimeSince` object. This is a lot of object creation. Imagine we have several log analysis applications, each looking at different aspects of the log messages. Making a change across all these applications means having to go back and fix all of the places where these `TimeSince` objects were created. Cluttering up the `LogProcessor` class with details of how the `TimeSince` class works violates the Single Responsibility design principle. Another principle, **Don't Repeat Yourself** (**DRY**), seems to apply in this case, also.
- Instead, we can add an adapter that connects the needs of the `LogProcessor` class with the methods available from the `TimeSince` class.

The Adapter solution introduces a new class that offers the interface required by the `LogProcessor` class. It consumes the interface offered by the `TimeSince` class. It allows for independent evolution of the two classes, leaving them closed to modification, but open to extension. It looks like this:

```
class IntervalAdapter:
    def __init__(self) -> None:
        self.ts: TimeSince | None = None

    def time_offset(self, start: str, now: str) -> float:
```

```
        if self.ts is None:
            self.ts = TimeSince(start)
        else:
            h_m_s = self.ts.parse_time(start)
            if h_m_s != (self.ts.hr, self.ts.min, self.ts.sec):
                self.ts = TimeSince(start)
        return self.ts.interval(now)
```

This adapter creates a `TimeSince` object when it's needed. If there is no `TimeSince`, it has to create one. If there is an existing `TimeSince` object, and it uses the already established start time, the `TimeSince` instance can be reused. When the `LogProcessor` class needs to shift the focus of the analysis to a new error message, then a new `TimeSince` needs to be created.

Here's the final design for the `LogProcessor` class, using the `IntervalAdapter` class:

```
class LogProcessor:
    def __init__(self, log_entries: list[tuple[str, str, str]]) -> None:
        self.log_entries = log_entries
        self.time_convert = IntervalAdapter()

    def report(self) -> None:
        first_time, first_sev, first_msg = self.log_entries[0]
        for log_time, severity, message in self.log_entries:
            if severity == "ERROR":
                first_time = log_time
            interval = self.time_convert.time_offset(first_time, log_time)
            print(f"{interval:8.2f} | {severity:7s} {message}")
```

We created an `IntervalAdapter()` instance during initialization. Then we used this object to compute each time offset. This lets us reuse the existing `TimeSince` class without any modification to the original class, and it leaves the `LogProcessor` uncluttered by details of how `TimeSince` works.

We can also tackle this kind of design through inheritance. We could extend `TimeSince` to add the needed method to it. This inheritance alternative isn't a bad idea, and it illustrates the common situation where there's no single "right" answer. In some cases, we need to write out the inheritance solution and compare it with the adapter solution to see which one is easier to explain.

It is often possible to use a function as an adapter. While this doesn't obviously fit the traditional design of the Adapter class design pattern, it's a distinction with little practical impact: an instance of a class with the `__call__()` method is a callable object, indistinguishable from a function. A

function can be a perfectly good Adapter; Python doesn't require everything to be defined in classes.

The distinction between Adapter and Decorator is small but important. An Adapter often extends, modifies, or combines more than one method from the class(es) being adapted. A Decorator, however, generally avoids profound changes, keeping a similar interface for a given method, adding features incrementally. As we saw in *Chapter 11*, a Decorator should be viewed as a specialized kind of Adapter.

Using an Adapter class is a lot like using a Strategy class; the idea is that we might make changes and need a different Adapter someday. The principal difference is that Strategies are often chosen at runtime, whereas an Adapter is a design-time choice and changes very slowly.

The next pattern we'll look at is similar to an Adapter, as it also wraps functionality inside a new container. The difference is the complexity of what is being wrapped. A Façade often contains considerably more complex structures.

The Façade pattern

The Façade pattern is designed to provide a simple interface to a complex system of components. It allows us to define a new class that encapsulates a typical usage of the system, thereby avoiding a design that exposes the many implementation details hiding in the object collaboration. Any time we want access to common or typical functionality, we can use a single object's simplified interface. If another part of the project needs access to more complete functionality, it is still able to interact with the components and individual methods directly.

The UML diagram for the Façade pattern is really dependent on the subsystem, shown as a package, `Big System`, but in a cloudy way it looks like this:

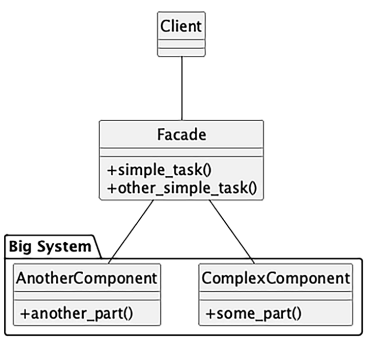


Figure 12.2: The Façade pattern

The Façade pattern is, in many ways, like the Adapter pattern. The primary difference is that a Façade tries to abstract a simpler interface out of a complex one, while an Adapter only tries to map one existing interface to another.

A Façade example

The imagesfor this book were made with *PlantUML* (`https://plantuml.com`). Each diagram starts as a text file and needs to be converted to the PNG file that's part of the text. This is a two-step process and we use the Façade pattern to combine the two processes.

The first part is locating all of the UML files. This is a walk through the directory tree, finding all files with names ending in `.uml`. We also look inside the file to see if there are multiple diagrams named inside the file.

```
import re
from pathlib import Path
from typing import Iterator


class FindUML:
    def __init__(self, base: Path) -> None:
        self.base = base
        self.start_pattern = re.compile(r"@startuml *(.*)")

    def uml_file_iter(self) -> Iterator[tuple[Path, Path]]:
        for source in self.base.glob("**/*.uml"):
            if any(n.startswith(".") for n in source.parts):
                continue
            body = source.read_text()
            for output_name in self.start_pattern.findall(body):
                if output_name:
                    target = source.parent / output_name
                else:
                    target = source.with_suffix(".png")
                yield (source.relative_to(self.base),
                target.relative_to(self.base))
```

The `FindUML` class requires a base directory. The `uml_file_iter()` method walks the entire directory tree, using the `Path.glob()` method. It skips over any directories with names that start with `.`; these are often used by tools like **tox**, **mypy**, or **git**, and we don't want to look inside these directories. The remaining fileswill have `@startuml` lines in them. Some will have a line that names multiple

output files. Most of the UML files don't create multiple files. The `self.start_pattern` regular expression will capture the name, if one is provided. The iterator yields tuples with two paths.

Separately, we have a class that runs the PlantUML application program as a subprocess. When Python is running, it's an operating system process. We can, using the `subprocess` module, start child processes that run other binary applications or shell scripts. We'll break this into two parts because the initialization involves a file-system search. Here's the first part:

```
from pathlib import Path
import subprocess


class PlantUML:
    local_dir = Path.cwd()

    def __init__(
        self,
        plantjar: str | Path = "plantuml-1.2024.7.jar",
        venv_name: str = ".venv"
    ) -> None:
        def find_first(name: str | Path) -> Path:
            places = [
                self.local_dir,
                self.local_dir / venv_name,
            ]
            places += Path.cwd().parents
            for place in places:
                if (path := place / name).exists():
                    return path
            raise FileNotFoundError(f"could not find {plantjar}")

        match plantjar:
            case Path() as path if path.is_absolute():
                self.plantjar = path
            case Path() as path if not path.is_absolute():
                self.plantjar = find_first(path)
            case str() as name:
                self.plantjar = find_first(name)
```

Here's the more interesting part, running the **PlantUML** application:

```
    def process(self, source: Path) -> None:
        env: dict[str, str] = {
            # Rarely needed...
            # "GRAPHVIZ_DOT": str(Path("/")/"to"/"graphviz"/"dot"),
        }
        command = ["java", "-jar", str(self.plantjar), "-progress",
        str(source)]
        subprocess.run(command, env=env, check=True)
```

This `PlantUML` class depends on finding the needed Java JAR file. This means — as preparation — we need to download the `plantuml.jar` file, and place it somewhere. The `__init__()` method searches all the usual locations for a JAR file, including a conda virtual environment, a virtual environment defined in your home directory, a virtual environment defined in the project directory, and then, all of the parent directories for the current working directory.

(Searching all the parents can seem a bit silly, but it's often appropriate. The book is 14 separate projects, all combined into one Git project repository. Since we don't want the JAR file in the Git repository, it's in the parent of the parent of all 14 projects.)

The `subprocess.run()` function accepts the command-line arguments and any special environment variables that need to be set. It will run the given command, with the given environment, and it will check the resulting return code to be sure the program ran properly.

Separately, we can use these steps to find all the UML files and create the diagrams. Because the interface between file finding and subprocess running is a bit awkward, a class that follows the Façade pattern helps combine the two features.

```
class GenerateImages:
    def __init__(self, base: Path, verbose: int = 0) -> None:
        self.finder = FindUML(base)
        self.painter = PlantUML()
        self.verbose = verbose

    def make_all_images(self) -> None:
        for source, target in self.finder.uml_file_iter():
            if not target.exists() or source.stat().st_mtime >
            target.stat().st_mtime:
                print(f"Processing {source} -> {target}")
                self.painter.process(source)
            else:
```

```
                if self.verbose > 0:
                    print(f"Skipping {source} -> {target}")
```

The `GenerateImages` class is a façade that combines features of the `FindUML` and the `PlantUML` classes. It uses the `FindUML.uml_file_iter()` method to locate source files and output image files. It checks the modification times of these files to avoid processing them if the image is newer than the source. (The `stat().st_mtime` is pretty obscure; it turns out the `stat()` method of a `Path` provides a lot of file status information, and the modification time is only one of many things we can find about a file.)

If the `.uml` file is newer, it means one of the authors changed it, and the images need to be regenerated. The main script to do this is now delightfully simple:

```
def main() -> None:
    g = GenerateImages(Path.cwd())
    g.make_all_images()


if __name__ == "__main__":
    main()
```

This example shows one of the important ways Python can be used to automate things. We broke the process into steps that we could implement in a few lines of code. Then we combined those steps, wrapping them in a Façade. Another, more complex application can use the Façade without worrying deeply about how it's implemented. What's essential here is that almost *any* instance of a class actiing as a wrapper to conceal more complex processing is a Façade over that complex processing.

Although it is rarely mentioned by name in the Python community, the Façade pattern is an integral part of the Python ecosystem. Because Python emphasizes language readability, both the language and its libraries tend to provide easy-to-comprehend interfaces for complicated tasks. For example, `for` statements, `list` comprehensions, and generators are all façades for a more complicated iterator protocol. The `defaultdict` implementation is a façade that abstracts away annoying edge cases when a key doesn't exist in a dictionary.

The third-party `requests` or `httpx` libraries are both powerful façades over the complications of HTTP processing. These packages include numerous other patterns; they're not simply examples of

the Façade pattern. The HTTP protocol is a conceptual façade over the underlying `socket` protocol for making requests and handling responses.

A Façade pattern conceals complexity. Sometimes, we want to avoid duplicating data. The next design pattern can help optimize storage when working with large volumes of data. It's particularly helpful on very small computers, typical for Internet of Things applications.

The Flyweight pattern

The Flyweight pattern is a memory optimization pattern. Novice Python programmers tend to ignore memory optimization, assuming the built-in garbage collector will take care of it. Relying on the built-in memory management is the best way to start. Indeed, for the most part, Python lets us ignore memory management.

There are a few cases where memory limitations surface. One example is very large data science applications. In these applications, memory constraints can become barriers, and more active measures need to be taken. The other common example is very small Internet of Things devices; for these, memory management can also be helpful.

The Flyweight pattern ensures that objects that share a state can use the same memory for their shared state. It is normally implemented only after a program has demonstrated memory problems. Bear in mind that premature optimization is the most effective way to create a program that is too complicated to maintain. First, build it the obvious way; if memory problems surface, then start refactoring.

In some languages, a Flyweight design requires careful sharing of object references, avoiding accidental object copying, and careful tracking of object ownership to ensure that objects aren't deleted prematurely. There are a lot of considerations to make sure memory is used safely. In Python, everything is an object, and all objects work through consistent references. A Flyweight design in Python is generally somewhat simpler than in other languages.

Let's have a look at the following UML diagram for the Flyweight pattern in the next page.

Each **Flyweight** object has no specific state of its own. Any time it needs to perform an operation on **SpecificState**, that state needs to be passed into the **Flyweight** by the calling code as an argument value. Traditionally, the factory that returns an instance of a `Flyweight` class is a separate object; its purpose is to return individual Flyweight objects, perhaps organized by a key or index of some kind. It works like the Singleton pattern we discussed in *Chapter 11*; if the Flyweight object exists, we return it; otherwise, we create a new one. In many languages, the factory is implemented, not

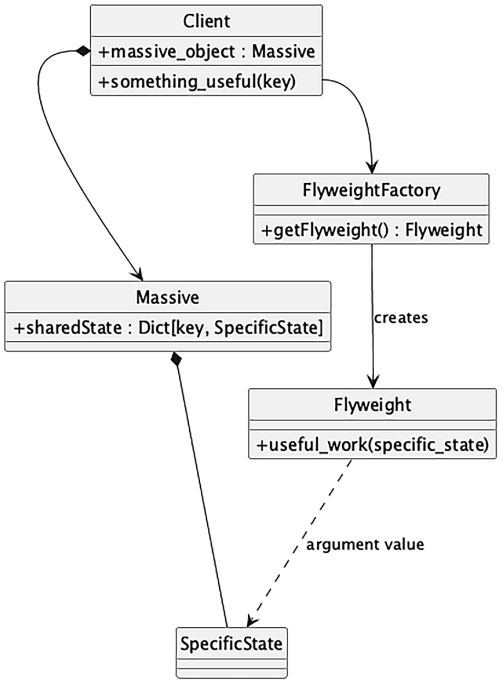


Figure 12.3: The Flyweight pattern

as a separate object, but as a static method on the `Flyweight` class itself.

We can liken this to the way the World Wide Web has replaced a computer loaded up with data. In the olden days, we would be forced to collect and index documents and files, filling up our local computer with copies of source material. This used to involve transfers of physical media like floppy disks and CDs. Now, we can – via a website – have a reference to the original data without making a bulky, space-consuming copy. Because we are working with a reference to the source data, we can read it easily on a mobile device. The Flyweight principle of working with a reference to data has been a profound change in our access to information.

Unlike the Singleton design pattern, which only needs to return one instance of a class, a Flyweight design may have multiple instances of the Flyweight classes. One approach is to store the items in a dictionary and provide values to Flyweight objects based on the dictionary key. Another common approach in some IoT applications is to leverage a buffer of items. On a large computer, allocating and deallocating objects is relatively low-cost. On a small IoT computer, we need to minimize object creation, which means leveraging Flyweight designs where a buffer is shared by objects.

A Flyweight example in Python

We'll start with some concrete classes for an IoT device that works with GPS messages. We don't want to create a lot of individual `Message` objects with duplicate values taken from a source buffer; instead, we want Flyweight objects to help save memory. This leverages two important features:

- The Flyweight objects reuse bytes in a single buffer. This avoids data duplication in a small computer.

- The Flyweight classes can have unique processing for the various message types. In particular, the GPGGA, GPGLL, and GPRMC messages all have latitude and longitude information. Even though the details vary by message, we don't want to create distinct Python objects. It's a fair amount of overhead to handle the case when the only real processing distinction is the location of the relevant bytes within a buffer.

Here's the UML diagram:

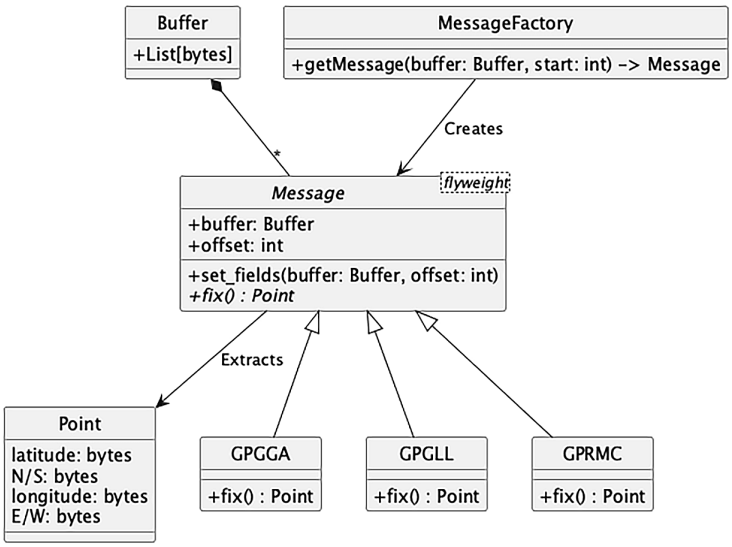


Figure 12.4: GPS messages UML diagram

Given a `Buffer` object with bytes read from the GPS, we can apply a `MessageFactory` to create Flyweight instances of the various `Message` subclasses. Each subclass has access to the shared `Buffer` object and can produce a `Point` object, but they have unique implementations reflecting the

distinct structure of each message.

There's an additional complication that is unique to Python. We can get into trouble when we have multiple references to an instance of the `Buffer` object. After working with a number of messages, we'll have local, temporary data in each of the `Message` subclasses, including a reference to the `Buffer` instance.

The situation might look as shown in the following diagram, which has the concrete objects and their references:

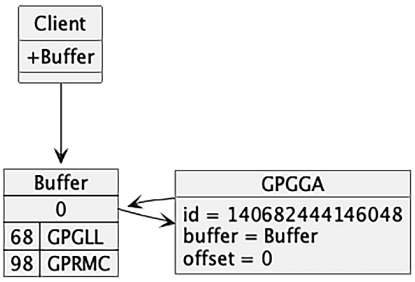


Figure 12.5: Circular references

Some client application, shown as a `Client` object, has a reference to a `Buffer` instance. It read and processed a bunch of GPS traffic using this buffer. Additionally, a specific `GPGGA` message instance also has a reference to the `Buffer` object because offset `0` in the buffer had a GPGGA message. Offsets `68` and `98` have other messages; these will also have references back to the `Buffer` instance.

Because the `Buffer` has a reference to a GPGGA `Message` object, and the `Message` also has a reference back to the `Buffer`, we have a circular pair of references. When the client stops using the `Buffer` instance, the reference count goes from four references to three. We cannot easily remove the `Buffer` because it's still used by `Message` objects.

We can solve this problem by taking advantage of Python's `weakref` module. Unlike ordinary ("strong") references, a weak reference isn't counted for the purposes of memory management. We can have lots of weak references to an object, but once the last ordinary reference is removed, the object can be removed from memory. This permits the client to start working with a new `Buffer` object without having to worry about the old `Buffer` cluttering up memory. The number of strong references goes from one to zero, allowing it to be removed. Similarly, each `Message` object could have one strong reference from the `Buffer`, so removing the `Buffer` will also remove each `Message`.

Weak references are part of the foundation of the Python runtime. Consequently, they are an

important optimization that surfaces in a few special cases. One of these optimizations is that we can't create a weak reference to a `bytes` object. The overhead would be painful.

In a few cases (like this) we need to create an Adapter for the underlying `bytes` object to transform it into an object that can have weak references.

```
from collections.abc import Sequence, Iterator
```

```
class Buffer(Sequence[int]):
    def __init__(self, content: bytes) -> None:
        self.content = content

    def __len__(self) -> int:
        return len(self.content)

    def __iter__(self) -> Iterator[int]:
        return iter(self.content)

    @overload
    def __getitem__(self, index: int) -> int:
        ...

    @overload
    def __getitem__(self, index: slice) -> bytes:
        ...

    def __getitem__(self, index: int | slice) -> int | bytes:
        return self.content[index]
```

This definition of a `Buffer` class doesn't really contain a great deal of new code. We provided three special methods, and all three delegated the work to the underlying `bytes` object. The `Sequence` abstract base type provides a few methods for us, like `index()` and `count()`.

The three definitions of the overloaded `__getitem__()` method is how we tell **mypy** of the important distinction between an expression like `buffer[i]` and `buffer[start: end]`. The first expression gets a single `int` item from the buffer, the second uses a slice and returns a `bytes` object. The final non-overload definition of `__getitem__()` implements the two overloads by delegating the work to the `self.contents` object, which handles this nicely.

Back in *Chapter 11*, we looked at using a state-based design to acquire and compute checksums. This chapter takes a different approach to working with a large volume of rapidly arriving GPS

messages.

Here's a typical GPS sentence:

```
>>> raw =
Buffer(b"$GPGGA,170834,4124.8963,N,08151.6838,W,1,05,1.5,280.2,M,-34.0,M,,*75")
```

The `$` starts the sentence. The `*` ends the sentence. The characters after the `*` are the checksum value. We'll ignore the two checksum bytes in this example, trusting that they are correct. Here's the abstract `Message` class with some common methods to help parse these GPS messages:

```
import abc
import weakref
```

```
class Message(abc.ABC):
    def __init__(self) -> None:
        self.buffer: weakref.ReferenceType[Buffer]
        self.offset: int
        self.end: int | None
        self.commas: list[int]

    def from_buffer(self, buffer: Buffer, offset: int) -> "Message":
        self.buffer = weakref.ref(buffer)
        self.offset = offset
        self.commas = [offset]
        self.end = None
        for index in range(offset, offset + 82):
            if buffer[index] == ord(b","):
                self.commas.append(index)
            elif buffer[index] == ord(b"*"):
                self.commas.append(index)
                self.end = index + 3
                break
        if self.end is None:
            raise GPSError("Incomplete")
        # TODO: confirm checksum.
        return self

    def __getitem__(self, field: int) -> bytes:
        if not hasattr(self, "buffer") or (buffer := self.buffer()) is None:
            raise RuntimeError("Broken reference")
```

```
        start, end = self.commas[field] + 1, self.commas[field + 1]
        return buffer[start:end]
```

The `__init__()` method doesn't actually do anything. We've provided a list of instance variables with their types, but we don't actually set them here. This is a way to alert **mypy** to what instance variables are going to be set elsewhere in the class.

In the `from_buffer()` method, we create a weak reference to a `Buffer` instance using the `weakref.ref()` function. As noted previously, this special reference is not used to avoid tracking how many places a `Buffer` object is used; it allows `Buffer` objects to be removed even if `Message` objects still have old, stale references to them.

The `from_buffer()` method scans the buffer for `","` characters, making it easier to locate where each field is. This can save some time if we need several fields. If we only need one or two fields, this might be excessive overhead.

In the `__getitem__()` method, we de-reference the weak reference to track down the `Buffer` object. Normally, when processing a `Buffer`, it's in memory along with some `Message` objects. Evaluating `self.buffer()` — calling the reference like a function — retrieves the ordinary reference we can use in the body of the method. At the end of the `__getitem__()` method, the buffer variable is no longer used, and the temporary reference vanishes.

A client application may have code like this:

```
while True:
    buffer = Buffer(gps_device.read(1024))
    # process the messages in the buffer.
```

The `buffer` variable has an ordinary reference to a `Buffer` object. Ideally, this is the only reference. Each time we execute this assignment statement, the old `Buffer` object will have zero references and can be removed from memory. After this assignment statement, and before we evaluate the `from_buffer()` method of a `Message`, an attempt to use the `__getitem__()` method of a `Message` object will raise a `RuntimeError` exception.

If our application attempts to use a `Message` object's `__getitem__()` method without having done `set_fields()` first, that's a serious, fatal bug. We've tried to make it obvious by crashing the application. When we get to *Chapter 13*, we can use unit tests to confirm that the methods are used in the proper order. Until then, we have to be sure we use `__getitem__()` correctly.

Here's the rest of the `Message` abstract base class, showing the methods required to extract a fix from a message:

```
    def get_fix(self) -> Point:
        return Point.from_bytes(
            self.latitude(), self.lat_n_s(), self.longitude(),
            self.lon_e_w()
        )

    @abc.abstractmethod
    def latitude(self) -> bytes:
        ...

    @abc.abstractmethod
    def lat_n_s(self) -> bytes:
        ...

    @abc.abstractmethod
    def longitude(self) -> bytes:
        ...

    @abc.abstractmethod
    def lon_e_w(self) -> bytes:
        ...
```

The `get_fix()` method delegates the work to four separate methods, each of which extracts one of the many fields from the GPS message. We can provide subclasses like the following:

```
class GPGLL(Message):
    def latitude(self) -> bytes:
        return self[1]

    def lat_n_s(self) -> bytes:
        return self[2]

    def longitude(self) -> bytes:
        return self[3]

    def lon_e_w(self) -> bytes:
        return self[4]
```

This class will use the `get_field()` method, inherited from the `Message` class, to pick out the bytes

for four specific fields in the overall sequence of bytes. Because the `get_field()` method uses a reference to a `Buffer` object, we don't need to duplicate the entire message's sequence of bytes. Instead, we reach back into the `Buffer` object to get the data, avoiding cluttering up memory.

We haven't shown the `Point` object. It's left as part of the exercises. It needs to convert strings of bytes into useful floating-point numbers.

Here's how we create a suitable Flyweight object, based on the message type in the buffer:

```
def message_factory(header: bytes) -> Message | None:
    # TODO: Add functools.lru_cache to save storage and time
    if header == b"GPGGA":
        return GPGGA()
    elif header == b"GPGLL":
        return GPGLL()
    elif header == b"GPRMC":
        return GPRMC()
    else:
        return None
```

If we're looking at a recognized message, we create an instance of one of our Flyweight classes. We left a comment suggesting another exercise: Use `functools.lru_cache` to avoid creating `Message` objects that are already available. Let's look at how `message_factory()` works in practice:

```
>>> buffer = Buffer(
...     b"$GPGLL,3751.65,S,14507.36,E*77"
... )
>>> flyweight = message_factory(buffer[1 : 6])
>>> flyweight.__class__.__name__
'GPGLL'
>>> flyweight.from_buffer(buffer, 0)
<gps_messages.GPGLL object at ...>

>>> flyweight.get_fix()
Point(latitude=-37.86083333333333, longitude=145.12266666666667)
>>> print(flyweight.get_fix())
(37°51.6500S, 145°07.3600E)
```

We've loaded up a `Buffer` object with some bytes. The message name is a slice of bytes in positions 1 to 6 of the buffer. The slice operation will create a small `bytes` object here. The `message_factory()` function will locate one of our Flyweight class definitions, the `GPGLL` class. We can then use the

`from_buffer()` method so the Flyweight can scan the `Buffer`, starting from offset zero, looking for `","` bytes to locate the starting point and ending point for the various fields.

When we evaluate `get_fix()`, the `GPGLL` flyweight will extract four fields, convert the values to useful degrees and return a `Point` object with two floating-point values. If we want to correlate this with other devices, we might want to show a value that has degrees and minutes separated from each other. It can be more helpful to see `37°51.6500S` than `37.86083333333333`.

Multiple messages in a buffer

Let's stretch this example out a bit, to look at a buffer with a sequence of messages in it. We'll put two GPGLL messages into a sequence of bytes. We'll include explicit end-of-line whitespace characters that some GPS devices include in the data stream.

```
>>> buffer_2 = Buffer(
...     b"$GPGLL,3751.65,S,14507.36,E*77\\r\\n"
...     b"$GPGLL,3723.2475,N,12158.3416,W,161229.487,A,A*41\\r\\n"
... )
>>> start = 0
>>> flyweight = message_factory(buffer_2[start+1 : start+6])
>>> p_1 = flyweight.from_buffer(buffer_2, start).get_fix()
>>> p_1
Point(latitude=-37.86083333333333, longitude=145.12266666666667)
>>> print(p_1)
(37°51.6500S, 145°07.3600E)
```

We've found the first GPGLL message, created a `GPGLL` object, and extracted the fix from the message. The next message begins where the previous message ends. This lets us start at a new offset in the buffer and examine a different region of bytes.

```
>>> flyweight.end
30
>>> next_start = buffer_2.index(ord(b"\$"), flyweight.end)
>>> next_start
32
>>>
>>> p_2 = flyweight.from_buffer(buffer_2, next_start).get_fix()
>>> p_2
Point(latitude=37.387458333333335, longitude=-121.97236)
>>> print(p_2)
```

```
(37°23.2475N, 121°58.3416W)
```

We've used the `message_factory()` function to create a new GPGLL object. Since the data from the message isn't in the object, we can reuse the previous GPGLL object. We can take out the `flyweight =` line of code, and the results are the same. When we use the `from_buffer()` method, we'll locate a new batch of `","` characters. When we use the `get_fix()` method, we'll get values from a new place in the overall collection of bytes.

This implementation creates a few short strings of bytes to create a cacheable object for use by `message_factory()`. It creates new float values when it creates a `Point`. It avoids slinging around large blocks of bytes, however, by making the message processing objects reuse a single `Buffer` instance.

Generally, using the Flyweight pattern in Python is a matter of making sure we have references to the original data. Generally, Python avoids making implicit copies of objects; almost all object creation is obvious, using a class name or perhaps comprehension syntax. One case where object creation is not obvious is taking a slice from a sequence, like a buffer of bytes: when we use `bytes[start: end]`, this makes a copy of the bytes. Too many of these and our IoT device is out of usable memory. A Flyweight design avoids creating new objects, and — in particular — avoids slicing strings and bytes to create copies of the data.

Our example also introduced the `weakref` module. This isn't essential for a Flyweight design, but it can be helpful to identify objects that can be removed from memory. While the two are often seen together, they're not closely related.

The Flyweight pattern can have an enormous impact on memory consumption. It is common for programming solutions that optimize CPU, memory, or disk space to result in more complicated code than their unoptimized brethren. It is therefore important to weigh up the trade-offs when deciding between code maintainability and optimization. When choosing optimization, try to use patterns such as Flyweight to ensure any complexity introduced by optimization is confined to a single (well-documented) section of the code.

Before we look at the Abstract Factory pattern, we'll digress a bit, to look at another memory optimization technique, unique to Python. This is the `__slots__` magic attribute name.

Memory optimization via Python's __slots__

If you have a lot of Python objects in one program, another way to save memory is through the use of `__slots__`. This is a sidebar, since it's not a common design pattern outside the Python language. It is a helpful Python design pattern because it can shave a few bytes off an object that's used widely. Instead of a Flyweight design — where storage is intentionally shared — a slots design creates objects with their own private data, but avoids Python's built-in dictionary. Instead, there is direct mapping from attribute name to a sequence of values, avoiding the rather large hash table that is a part of every Python `dict` object.

Looking back at our previous example in this chapter, we avoided describing the `Point` object that was created as part of the `get_fix()` method of each subclass of `Message`. Here's one possible definition of the `Point` class:

```
from math import radians, floor
```

```
class Point:
    __slots__ = ("latitude", "longitude")

    def __init__(self, latitude: float, longitude: float) -> None:
        self.latitude = latitude
        self.longitude = longitude

    def __repr__(self) -> str:
        return f"Point(latitude={self.latitude},
        longitude={self.longitude})"

    @classmethod
    def from_bytes(
        cls,
        latitude: bytes,
        N_S: bytes,
        longitude: bytes,
        E_W: bytes,
    ) -> "Point":
        lat_deg = float(latitude[:2]) + float(latitude[2:]) / 60
        lat_sign = 1 if N_S.upper() == b"N" else -1
        lon_deg = float(longitude[:3]) + float(longitude[3:]) / 60
        lon_sign = 1 if E_W.upper() == b"E" else -1
        return Point(lat_deg * lat_sign, lon_deg * lon_sign)
```

```
    def __str__(self) -> str:
        lat = abs(self.latitude)
        lat_deg = floor(lat)
        lat_min_sec = 60 * (lat - lat_deg)
        lat_dir = "N" if self.latitude > 0 else "S"
        lon = abs(self.longitude)
        lon_deg = floor(lon)
        lon_min_sec = 60 * (lon - lon_deg)
```

Each instance of a `Point` can have exactly two attributes with the names `latitude` and `longitude`. The `__init__()` method sets these values and provides useful type hints for tools like **mypy**.

In most other respects, this class is the same as a class without `__slots__`. The most notable difference is we cannot add attributes. Here's an example, showing what exception is raised:

```
>>> p2 = Point(latitude=49.274, longitude=-123.185)
>>> p2.extra_attribute = 42
Traceback (most recent call last):
...
AttributeError: 'Point' object has no attribute 'extra_attribute' and no
__dict__ for setting new attributes
```

The extra housekeeping of defining the names of the slots can be helpful when our application creates vast numbers of these objects. In those cases where an application is built on one or a very small number of instances of a class, then the memory-saving from introducing `__slots__` is negligible. This can be a slight performance boost from avoiding the dictionary overhead.

The memory savings can be profound. A list of about 5,600 GPS messages, using a conventional class, occupies about 3.5 Mb of memory. Compare this with a list using `__slots__` that only occupies 0.4 Mb. For this example, the `__slots__` implementation uses about $\frac{1}{8}$ of the storage. (Why 5,600 messages? The benchmark is based on a buffer size of 512 Kb of raw GPS output. This gives a funny-looking number of messages.)

The `@dataclass` decorator can be used with a `slots=True` option to define `__slots__` automatically. In some cases, adding this option can create a profound performance improvement.

It's also possible to improve performance using a `NamedTuple`. This structure can be as effective at saving memory as using `__slots__` in a more general class definition. We looked at these in *Chapter 8*.

We've seen how to manage complexity by wrapping objects in a Façade. We've seen how to manage memory use by using Flyweight objects that have little (or no) internal state. Next, we'll look at how we can create a variety of different kinds of objects using a *factory*.

The Abstract Factory pattern

The Abstract Factory pattern is appropriate when we have multiple possible implementations of a system that depend on some configuration or platform detail. The calling code requests an object from the Abstract Factory, not knowing exactly what class of object will be returned. The underlying implementation returned may depend on a variety of factors, such as the current locale, operating system, or local configuration.

Within the Python standard library, the `pathlib` module has the `Path` class that acts as an abstract factory for concrete `PosixPath` and `WindowsPath` objects that have operating system-specific features.

Other common examples of the Abstract Factory pattern include code for operating-system-independent toolkits, database backends, and country-specific formatters or calculators. An operating-system-independent GUI toolkit might use an Abstract Factory pattern that returns a set of WinForm widgets under Windows, Cocoa widgets under Mac, GTK widgets under Gnome, and QT widgets under KDE. Django provides an abstract factory that returns a set of object-relational classes for interacting with a specific database backend (MySQL, PostgreSQL, SQLite, and others) depending on a configuration setting for the current site. If the application needs to be deployed in multiple places, each one can use a different database backend by changing only one configuration variable. Different countries have different systems for calculating taxes, subtotals, and totals on retail merchandise; an Abstract Factory can return a particular tax calculation object.

There are two central features of an Abstract Factory:

- We need to have multiple implementation choices. Each implementation has a factory class to create objects. A single Abstract Factory defines the interface to the implementation factories.
- We have a number of closely related objects, and the relationships are implemented via multiple methods of each factory.

The following UML class diagram seems like a clutter of relationships:

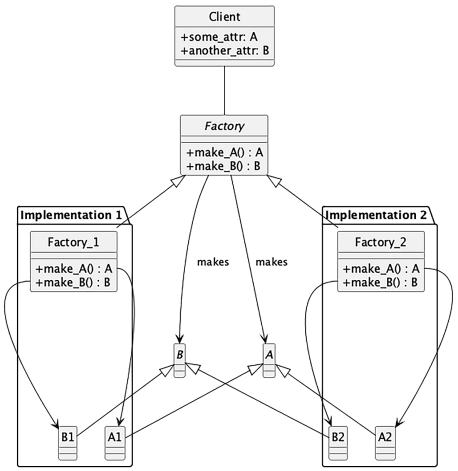


Figure 12.6: Abstract Factory pattern

There's an essential symmetry here that's very important. The client needs instances of class A and class B. To the client, these are abstract class definitions. The `Factory` class is an abstract base class that requires an implementation. Each of the implementation packages, `Implementation 1` and `Implementation 2`, provides concrete `Factory` subclasses that will build the necessary `A` and `B` instances for the client.

An Abstract Factory example

The UML class diagram for the Abstract Factory pattern is hard to understand without a specific example, so let's turn things around and create a concrete example first. Let's look at two card games, Poker and Cribbage. Don't panic, you don't need to know all the rules, only that they're similar in a few fundamental ways but different in the details. This is depicted in *Figure 12.7* on the next page.

The `Game` class requires `Card` objects and `Hand` objects (among several others). We've shown that the abstract `Card` objects are contained within the abstract `Hand` collection. Each implementation provides some unique features. For the most part, the `PokerCard` matches the generic `Card` definition. The `PokerHand` class, however, extends the `Hand` abstract base class with all the unique rules for defining the rank of the hand. Poker players know that there are a very, very large number of Poker game variants. We've shown a hand containing five cards because this seems to be a common feature of many games.

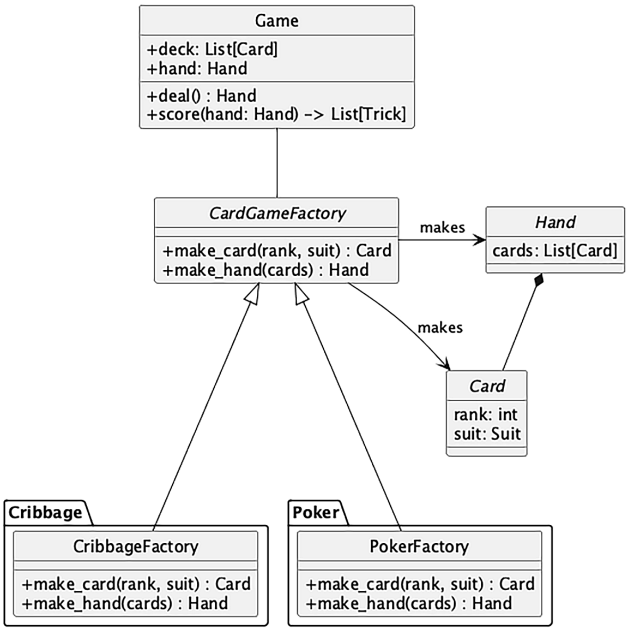


Figure 12.7: Abstract Factory pattern for Cribbage and Poker

The Cribbage implementation introduces a number of types of `CribbageCard` subclasses, each of which has an additional attribute, `points`. The `CribbageFace` cards are all worth 10 points, while for the other kinds of `CribbageCard` classes, the number of points matches the rank. The `CribbageHand` class extends the abstract base class of `Hand` with the unique rules for finding all the scoring combinations in a hand. We can use an Abstract Factory to build `Card` and `Hand` objects that are appropriate for each game.

Here are the core definitions of `Suit`, `Card`, `Trick`, and `Hand`. A hand's score is based on the score of the one (or more) tricks it contains. We didn't make these abstract base classes. Python doesn't require this, and the extra complexity didn't seem helpful.

```
from enum import Enum, auto
from typing import NamedTuple


class Suit(str, Enum):
    Clubs = "\N{Black Club Suit}"
    Diamonds = "\N{Black Diamond Suit}"
```

```
    Hearts = "\N{Black Heart Suit}"
    Spades = "\N{Black Spade Suit}"
```

```
class Card(NamedTuple):
```

```
    rank: int
    suit: Suit

    def __str__(self) -> str:
        return f"{self.rank}{self.suit.value}"


class Trick(int, Enum):
    pass


class Hand(list[Card]):
    def __init__(self, *cards: Card) -> None:
        super().__init__(cards)

    @abc.abstractmethod
    def scoring(self) -> list[Trick]:
        ...
```

These seem to capture the essence of "card" and "hand of cards." They include the suit and rank concept of cards, and also computing the score from a collection of tricks. We'll need to extend these with subclasses that pertain to each game. We'll also need an Abstract Factory that creates cards and hands for us:

```
import abc
```

```
class CardGameFactory(abc.ABC):
    @abc.abstractmethod
    def make_card(self, rank: int, suit: Suit) -> "Card":
        ...

    @abc.abstractmethod
    def make_hand(self, *cards: Card) -> "Hand":
        ...
```

We've made the factory an actual abstract base class. Each individual game needs to provide extensions for the game's unique features of `Hand` and `Card`. Any game application will also provide an implementation of the `CardGameFactory` class that can build the expected classes.

We can define the cards for Cribbage like this:

```
class CribbageCard(Card):
    @property
    def points(self) -> int:
        return self.rank


class CribbageAce(Card):
    @property
    def points(self) -> int:
        return 1


class CribbageFace(Card):
    @property
    def points(self) -> int:
        return 10
```

These extensions to the base `Card` class all have an additional `points` property. In Cribbage, one of the kinds of tricks is any combination of cards worth 15 points. Most cards have points equal to the rank, but the Jack, Queen, and King are all worth 10 points. This also means the Cribbage extension to `Hand` has a rather complex method for scoring, which we'll omit for now.

```
class CribbageHand(Hand):
    starter: Card

    def upcard(self, starter: Card) -> "Hand":
        self.starter = starter
        return self

    def scoring(self) -> list[Trick]:
        """15's. Pairs. Runs. Right Jack."""
```

```
        ### details omitted...
        return tricks
```

To provide some uniformity between the games, we've designated the scoring combinations in Cribbage and the rank of the hand in Poker as a subclass of "Trick." In Cribbage, there's a fairly large number of point-scoring tricks. In Poker, on the other hand, there's a single trick that represents the hand as a whole. Tricks don't seem to be a place where an Abstract Factory is useful.

The computation of the various scoring combinations in Cribbage is a rather sophisticated problem. It involves looking at all possible combinations of cards that total to 15 points, among other things. These details are unrelated to the Abstract Factory design pattern.

The Poker variant has its own unique complication: Aces are a higher rank than the King. This leads to the following class definitions:

```
class PokerCard(Card):
    def __str__(self) -> str:
        if self.rank == 14:
            return f"A{self.suit}"
        return f"{self.rank}{self.suit}"
```

```
class PokerHand(Hand):
    def scoring(self) -> list[Trick]:
```

```
        ### details omitted...
        return [rank]
```

Ranking the various hands in poker is also a rather sophisticated problem, but outside the Abstract Factory realm. Here's the concrete factory that builds hands and cards for Poker:

```
class PokerFactory(CardGameFactory):
    def make_card(self, rank: int, suit: Suit) -> "Card":
        if rank == 1:
            # Aces above kings
            rank = 14
        return PokerCard(rank, suit)

    def make_hand(self, *cards: Card) -> "Hand":
        return PokerHand(*cards)
```

Note the way the `make_card()` method reflects the way aces work in Poker. Having the Ace outrank the King reflects a common complication in a number of card games; we need to reflect the various

ways Aces work.

Here's a test case for how Cribbage works:

```
>>> factory = CribbageFactory()
>>> cards = [
...     factory.make_card(6, Suit.Clubs),
...     factory.make_card(7, Suit.Diamonds),
...     factory.make_card(8, Suit.Hearts),
...     factory.make_card(9, Suit.Spades),
... ]
>>> starter = factory.make_card(5, Suit.Spades)
>>> hand = factory.make_hand(*cards)
>>> score = sorted(hand.upcard(starter).scoring())
>>> [t.name for t in score]
['Fifteen', 'Fifteen', 'Run_5']
```

We've created an instance of the `CribbageFactory` class, a concrete implementation of the abstract `CardGameFactory` class. We can use the factory to create some cards, and we can also use the factory to create a hand of cards. When playing Cribbage, an additional card is flipped, called the "starter." In this case, our hand is four cards in sequence, and the starter happens to fit with that sequence. We can score the hand and see that there are three scoring combinations: there are two ways to make 15 points, plus a five-card run.

This design provides some hints toward what needs to be done when we want to add support for more games. Introducing new rules means creating the new `Hand` and `Card` subclasses and extending the Abstract Factory class definition, also. Of course, inheritance leads to the opportunity for reuse, something we can capitalize on to create families of games with similar rules.

Abstract Factories in Python

The previous example highlights an interesting consequence of the way Python's duck typing works. Do we really need the abstract base class, `CardGameFactory`? It provides a framework used for type checking, but otherwise doesn't have any useful features. Since we don't really need it, we can think of this design as having three parallel modules (see *Figure 12.8* on the next page).

Both of the defined games implement a class, `CardGameFactory`, that defines the unique features of the game. Because these are in separate modules, we can use the same name for each class. This lets us write a Cribbage application that uses `from cribbage import CardGameFactory`. This approach skips past the overhead of a common abstract base class and lets us provide extensions as

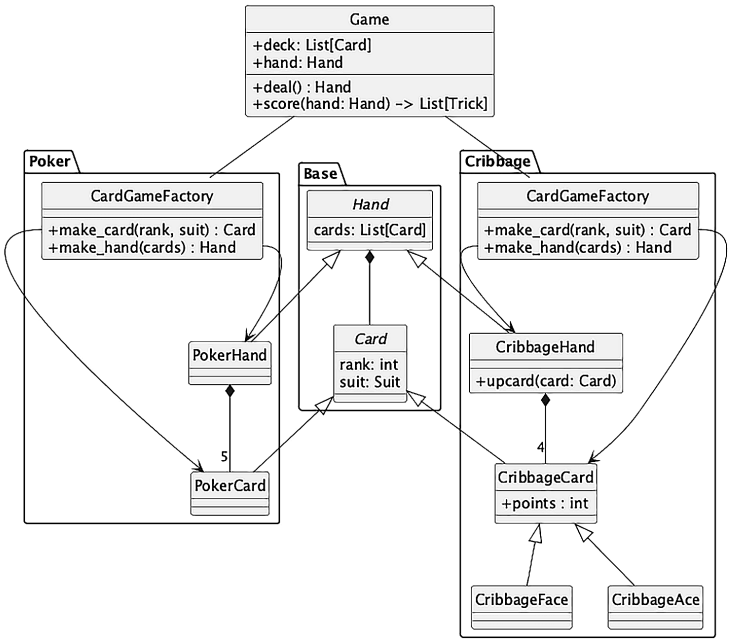


Figure 12.8: Abstract Factory without abstract base classes

separate modules sharing some common base class definitions. Each alternative implementation also provides a common module-level interface: they expose a standard class name that handles the remaining details of creating unique objects.

In this case, the Abstract Factory becomes a concept that helps us design a common module structure; it is not implemented as an actual abstract base class. We'll need to provide adequate documentation in the docstrings for all classes that purport to be `CardGameFactory` implementations. We can clarify our intentions by defining protocols using `typing.Protocol`. It could look like this:

```
class CardGameFactoryProtocol(Protocol):
    def make_card(self, rank: int, suit: Suit) -> "Card":
        ...

    def make_hand(self, *cards: Card) -> "Hand":
        ...
```

This definition allows tools like **mypy** to confirm that a `Game` class can refer to either a `poker.CardGameFactory` or a `cribbage.CardGameFactory` because both implement the same protocol. Unlike the abstract

base class definition, this is not a runtime check. A protocol definition is only used by tools to confirm that the code is likely to work.

The Abstract Factory pattern helps us define related families of objects – for instance, playing cards and hands. A single factory can produce two separate classes of objects that are closely related. In some cases, the relationships aren't a simple collection and an item. Sometimes there are sub-collections in addition to items. These kinds of structures can be handled using the Composite design pattern.

The Composite pattern

The Composite pattern allows complex tree structures to be built from simple components, often called **nodes**. A node with children will behave like a container; a node without children will behave like a single object. A composite object is — generally — a container object, where the content may be another composite object.

Traditionally, each node in a composite object must be either a **leaf** node (that cannot contain other objects) or a **composite** node. and leaf nodes can have the same interface. The following UML diagram shows this elegant parallelism as a `some_action()` method:

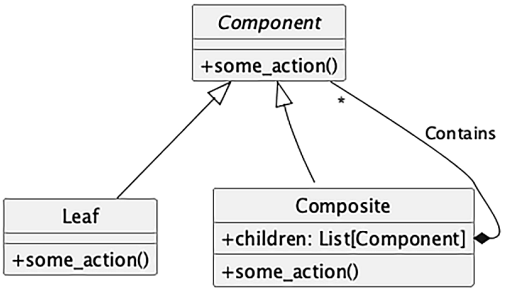


Figure 12.9: The Composite pattern

This simple pattern, however, allows us to create complex arrangements of elements, all of which satisfy the interface of the component object. *Figure 12.10* (next page) depicts a concrete instance of such a complicated arrangement.

The Composite pattern applies to language processing. Both natural languages and artificial languages (like Python) tend to follow rules that are hierarchical and fit nicely with the Composite design pattern. Markup languages, like HTML, XML, RST, and Markdown, tend to reflect some common composite concepts like lists of lists, and headers with sub-headings.

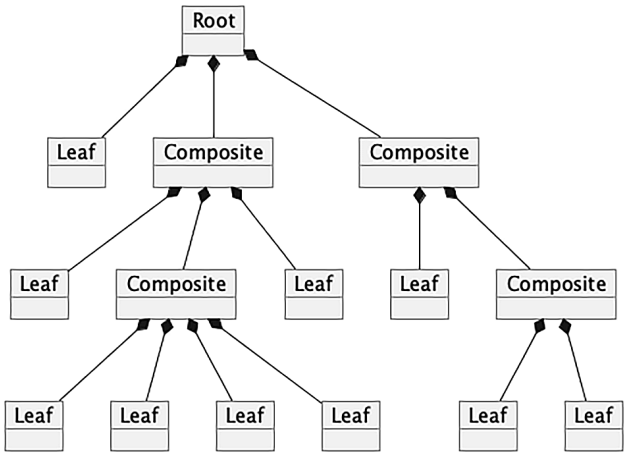


Figure 12.10: A large Composite object collection

A programming language involves recursive tree structures. The Python standard library includes the `ast` module, which provides the classes that define the structure of Python code. We can use this module to examine Python code without resorting to regular expressions or other hard-to-get-correct text processing.

A Composite example

The Composite pattern works best when applied to tree structures with nested collections of children. Examples include the tag structure in HTML documents, and the files and folders of a file system. A programming language with nested statements — like Python — is best described using the Composite design pattern.

We'll look at the file system, since it's relatively simple. Regardless of whether a node in the tree is an ordinary data file or a folder, it is still subject to operations such as moving, copying, or deleting the node. We can create a component interface that supports these operations, and then use a composite object to represent folders, and leaf nodes to represent data files.

This example overlaps with the `pathlib` module in many ways. It's important to look more at the design pattern than the overlaps with other packages.

Of course, in Python, once again, we can take advantage of duck typing to implicitly provide the interface, so we only need to write two classes. Let's sketch out the interfaces first. Here's some

incomplete code to help us start thinking about `Folder` and `File`:

```
class Folder:
    def __init__(
            self,
            name: str,
            children: dict[str, "Node"] | None = None
    ) -> None:
        self.name = name
        self.children = children or {}
        self.parent: "Folder | None" = None
    def __repr__(self) -> str:
        return f"Folder({self.name!r}, {self.children!r})"
    def add_child(self, node: "Node") -> "Node":
        node.parent = self
        return self.children.setdefault(node.name, node)
    def move(self, new_folder: "Folder") -> None:
        pass  # Changes the parent
    def copy(self, new_folder: "Folder") -> None:
        pass  # Whole tree
    def remove(self) -> None:
        pass  # Must be empty

class File:
    def __init__(self, name: str) -> None:
        self.name = name
        self.parent: "Folder | None" = None
    def __repr__(self) -> str:
        return f"File({self.name!r})"
    def move(self, new_path: "File") -> None:
        pass  # Changes the parent
    def copy(self, new_path: "File") -> None:
        pass  # Changes both parents
    def remove(self):
        pass  # Changes the parent
```

For each `Folder`, a composite object, we maintain a dictionary of children. The children may be a mixture of `Folder` and `File` instances. For many composite implementations, a list is sufficient, but in this case, a dictionary will be useful for looking up children by name.

Thinking about the methods involved, there are several patterns:

- The `move()` method has the comment "changes the parent." A file isn't moved in isolation;

the containing folder reflects the move. To do a move, relocating the `Folder` will carry along all the children. Relocating a `File` will turn out to be the same code; we can use an empty dictionary to show there are no children.

- To do a copy of a `Folder`, we'll need to copy all of the children. For a `File` node, we don't need to do anything more.
- For a delete, we should follow the Linux pattern of requiring the children have already been removed before trying to remove a parent.

This design lets us create subclasses with distinct operation implementations. Each subclass implementation could make external requests, or perhaps make OS requests on the local machine.

Also, we need to upgrade our data structure to include the the parent node for a given `File` or `Folder` node. Sketching out the initial design made it clear, the parent `Folder` information was required.

To take advantage of the similar operations, we can extract the common methods into an abstract base class. Let's refactor this to create a base class, `Node`, with the following code:

```
import abc
```

```
class Node(abc.ABC):
    def __init__(
        self,
        name: str,
    ) -> None:
        self.name = name
        self.parent: "Folder | None" = None

    def move(self, new_place: "Folder") -> None:
        previous = self.parent
        new_place.add_child(self)
        if previous:
            del previous.children[self.name]

    @abc.abstractmethod
    def copy(self, new_folder: "Folder") -> None:
        ...

    @abc.abstractmethod
    def remove(self) -> None:
```

```
        ...
```

This abstract `Node` class defines that each node has a string with a reference to a parent. Keeping the parent information around lets us look "up" the tree toward the root node. This makes it possible to move and remove files by making a change to the parent's collection of children.

The `move()` method reassigns a `Folder` or a `File` object to a new parent. It follows up by removing the object from its previous location. For the `move()` method, the target should be an existing `Folder`, or we'll get an error because a `File` instance doesn't have an `add_child()` method. As in many examples in technical books, error handling is woefully absent, to help focus on the principles under consideration. A common practice is to handle any `AttributeError` exceptions by raising a new `TypeError` exception. See *Chapter 4*.

We can then extend this class to provide the unique features of a `Folder` that has children, and a `File`, which is the leaf node of the tree and has no children. Here's the `Folder` class:

```
class Folder(Node):
    def __init__(self, name: str, children: dict[str, "Node"] | None = None)
    -> None:
        super().__init__(name)
        self.children = children or {}

    def __repr__(self) -> str:
        return f"Folder({self.name!r}, {self.children!r})"

    def add_child(self, node: "Node") -> "Node":
        node.parent = self
        return self.children.setdefault(node.name, node)

    def copy(self, new_folder: "Folder") -> None:
        target = cast(Folder, new_folder.add_child(Folder(self.name)))
        for c in self.children:
            self.children[c].copy(target)

    def remove(self) -> None:
        names = list(self.children)
        for c in names:
            self.children[c].remove()
        if self.parent:
            del self.parent.children[self.name]
```

The details of the move, copy, and remove operations are incomplete: they don't make any changes to real files in the OS filesystem. We'll leave those as an exercise.

Here's the `File` class:

```
class File(Node):
    def __repr__(self) -> str:
        return f"File({self.name!r})"

    def copy(self, new_folder: "Folder") -> None:
        new_folder.add_child(File(self.name))

    def remove(self) -> None:
        if self.parent:
            del self.parent.children[self.name]
```

Again, the details of the move, copy, and remove operations are incomplete: they don't make any changes to real files in the OS filesystem. We want to get the composite design right first.

When we add a child to a `Folder`, we'll do two things. First, we tell the child who their new parent is. This makes sure that each `Node` (except the root `Folder` instance) has a parent. Second, we'll drop the new `Node` into the folder's collection of children, if it doesn't already exist.

When we copy `Folder` objects around, we need to make sure all the children are copied. Each child could, in turn, be another `Folder`, with children. This recursive walk involves delegating the `copy()` operation to each sub-`Folder` within a `Folder` instance. The implementation for a `File` object, on the other hand, is simpler.

The recursive design for removal is similar to the recursive copy. A `Folder` instance must first remove all of the children; this may involve removing sub-`Folder` instances. A `File` object, on the other hand, can be directly removed.

Well, that was easy enough. Let's see if our composite file hierarchy is working properly with the following code snippet:

```
>>> tree = Folder("Tree")
>>> tree.add_child(Folder("src"))
Folder('src', {})
>>> tree.children["src"].add_child(File("ex1.py"))
File('ex1.py')
>>> tree.add_child(Folder("src"))
```

```
Folder('src', {'ex1.py': File('ex1.py')})
>>> tree.children["src"].add_child(File("test1.py"))
File('test1.py')
>>> tree
Folder('Tree', {'src': Folder('src', {'ex1.py': File('ex1.py'), 'test1.py':
File('test1.py')})})
```

The value of `tree` can be a little difficult to visualize. Here's a variation on the display that can help.

```
+-- Tree
     +-- src
          +-- ex1.py
          +-- test1.py
```

We didn't cover the algorithm for producing this nested visualization. It's not too difficult to add to the class definitions. We've left this as an exercise, also.

We can see that the parent folder, `Tree`, has a sub-folder, `src`, with two files inside it. We can describe a filesystem operation like this:

```
>>> test1 = tree.children["src"].children["test1.py"]
>>> test1
File('test1.py')
>>> tree.add_child(Folder("tests"))
Folder('tests', {})
>>> test1.move(tree.children["tests"])
>>> tree
Folder('Tree', {'src': Folder('src', {'ex1.py': File('ex1.py')}), 'tests':
Folder('tests', {'test1.py': File('test1.py')})})
```

We've created a new folder, `tests`, and moved the file. Here's another view of the resulting composite objects:

```
+-- Tree
     +-- src
          +-- ex1.py
     +-- tests
          +-- test1.py
```

The Composite pattern is extremely useful for a variety of tree-like structures, including GUI

widget hierarchies, file hierarchies, tree sets, graphs, and the HTML document object model (DOM). Sometimes, if only a shallow tree is being created, we can get away with a list of lists or a dictionary of dictionaries, and do not need to implement custom component, leaf, and composite classes. Indeed, JSON, YAML, and TOML documents often follow the dict-of-dict pattern.

While we often use abstract base classes for building Composite objects, it isn't required; Python's duck typing can make it easy to add other objects to a composite hierarchy, as long as they have the correct interface.

One of the important aspects of the Composite pattern is a common interface for the various subtypes of a node. We needed two implementation variants for `Folder` and `File` classes. In some cases, these operations are similar, and it can help to offer a template implementation of a complex method.

The Template pattern

The Template pattern (sometimes called the Template method) is useful for removing duplicate code; it's intended to support the **Don't Repeat Yourself** design principle we discussed in *Chapter 5*. It is designed for situations where we have several different tasks to accomplish that have some, but not all, steps in common. The common steps are implemented in a base class, and the distinct steps for each subclass are overridden to provide custom behavior. This is built on the Strategy pattern; in a way, it's an application of Strategy to a more specialized need. Here it is in the UML format:

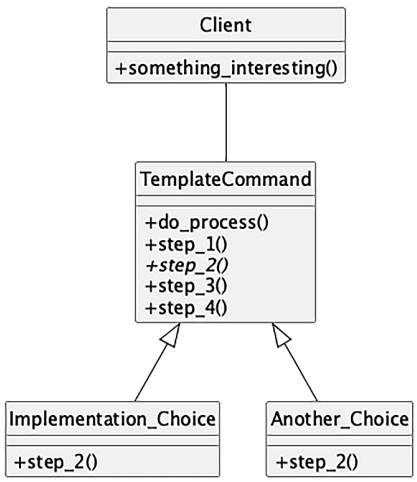


Figure 12.11: The Template pattern

This may seem like a straightforward application of inheritance. That's because it is generally pretty clear what's going on. The interesting spin is the idea that there's a general template for a complicated process, and steps of the process can be overridden in distinct ways.

A Template example

Let's create a car sales reporter as an example. We can store records of sales in an SQLite database table. SQLite is the built-in database engine that allows us to store records using SQL syntax. Python includes SQLite in its standard library, so there are no extra modules to install.

We have two common tasks we need to perform:

- Select all sales of new vehicles and output them to the screen in a comma-delimited format
- Output a comma-delimited list of all salespeople with their gross sales and save it to a file that can be imported to a spreadsheet

These seem like quite different tasks, but they have some common features. In both cases, we need to perform the following steps:

1. Connect to the database.
2. Construct a query for new vehicles or gross sales.
3. Issue the query.
4. Format the results into a comma-delimited string.
5. Output the data to a file or email.

The query construction and output steps are different for the two tasks, but the remaining steps are identical. We can use the Template pattern to put the common steps in a base class, and the varying steps in two subclasses.

Before we start, let's create a database and put some sample data in it, using a few lines of SQL:

```
import sqlite3


def db_preparation(db_name: str = "sales.db") -> sqlite3.Connection:
    conn = sqlite3.connect(db_name)

    conn.execute(
        """
```

```
        CREATE TABLE IF NOT EXISTS Sales (
            salesperson text,
            amt currency,
            year integer,
            model text,
            new boolean
        )
        """
    )
    conn.execute(
        """
        DELETE FROM Sales
        """
    )
    values = [
        ('Tim', 16000, 2010, 'Honda Fit', 'true'),
        ('Tim', 9000, 2006, 'Ford Focus', 'false'),
        ('Hannah', 8000, 2004, 'Dodge Neon', 'false'),
        ('Hannah', 28000, 2009, 'Ford Mustang', 'true'),
        ('Hannah', 50000, 2010, 'Lincoln Navigator', 'true'),
        ('Jason', 20000, 2008, 'Toyota Prius', 'false')
    ]
    conn.executemany(
        """
            INSERT INTO Sales VALUES (?,?,?,?,?)
        """,
        values)
    conn.commit()
    return conn
```

Hopefully, you can see what's going on here even if you don't know SQL. We've created a table named `Sales` to hold the data, and used six `insert` statements to add sales records. The data will be stored in a file named `sales.db`. Now we have a sample database with a table we can work with in developing our Template pattern.

Since we've already outlined the steps that the template has to perform, we can start by defining the base class that contains the steps. Each step gets its own method (to make it easy to selectively override any one step), and we have one more managerial method that calls the steps in turn. Without any method content, here's how the class might look as a first step toward completion:

```
class QueryTemplate:
    def __init__(self, db_name: str = "sales.db") -> None:
        self.db_name = db_name
        self.conn: sqlite3.Connection
        self.results: list[tuple[str, ...]]
        self.query: str
        self.header: list[str]

    def connect(self) -> None:
        self.conn = sqlite3.connect(self.db_name)

    def construct_query(self) -> None:
        raise NotImplementedError("construct_query not implemented")

    def do_query(self) -> None:
        results = self.conn.execute(self.query)
        self.results = results.fetchall()

    def output_context(self) -> ContextManager[TextIO]:
        self.target_file = sys.stdout
        return cast(ContextManager[TextIO], contextlib.nullcontext())

    def output_results(self) -> None:
        writer = csv.writer(self.target_file)
        writer.writerow(self.header)
        writer.writerows(self.results)

    def process_format(self) -> None:
        self.connect()
        self.construct_query()
        self.do_query()
        with self.output_context():
            self.output_results()
```

The `process_format()` method is the primary method to be called by an outside client. It will process the query and format the results. It ensures each step is executed in order, but it does not care whether that step is implemented in this class or in a subclass. For our examples, we expect the `construct_query()` and the `output_context()` methods are likely to change.

In Python, we can formalize our expectation by using an abstract base class. A commonly used alternative is to define a class with default methods. Some do nothing useful. Others, like `construct_query()`, raise a `NotImplementedError` exception to show that a method is missing.

As with the `@abstractmethod` decoration and the `abc.ABC` base class, this is a runtime check.

Looking more deeply, we can see three distinct ways to create what is effectively an abstract base class.

- Use `abc.ABC` as the base class, and decorate abstract methods with `@abstractmethod`. Tools like **mypy** can check for missing implementations.
- Define methods that raise the `NotImplementedError` exception to help the developer understand that the class is meant to be subclassed and the method overridden. It can be described as "smuggling in an abstract base class without being explicit" in the `class` definition. Tools like **mypy** can't discern the intent behind this code; but a person will see it when their unit tests fail.
- Provide "do nothing" default behavior. For example, the `output_context()` method *may* be overridden. There's a default implementation provided that sets the `self.target_file` instance variable and also returns a context value. The default uses `sys.stdout` as the output file and a null context manager.

Now we have a template class that takes care of the details unlikely to change. This makes it flexible enough to allow the execution and formatting of a wide variety of queries. The best part is, if we ever want to change our database engine from SQLite to another database engine (such as `py-postgresql`), we only have to do it here, in this template class, and we don't have to touch the two (or two hundred) subclasses we might have written.

Let's have a look at the concrete classes now:

```
import datetime


class NewVehiclesQuery(QueryTemplate):
    def construct_query(self) -> None:
        self.query = """
            SELECT * FROM Sales WHERE new='true'
        """
        self.header = ["salesperson", "amt", "year", "model", "new"]


class SalesGrossQuery(QueryTemplate):
    def construct_query(self) -> None:
        self.query = """
```

```
            SELECT salesperson, sum(amt) FROM Sales GROUP BY salesperson
        """
        self.header = ["salesperson", "total sales"]

    def output_context(self) -> ContextManager[TextIO]:
        today = datetime.date.today()
        filepath = Path(f"gross_sales_{today:%Y%m%d}.csv")
        self.target_file = filepath.open("w")
        return self.target_file
```

These two classes are actually pretty short, considering what they're doing: connecting to a database, executing a query, formatting the results, and outputting them. The superclass takes care of the repetitive work, but lets us easily specify those steps that vary between tasks. Further, we can also easily change steps that are provided in the base class. For example, if we wanted to output something other than a comma-delimited string (for example, an HTML report to be uploaded to a website), we could still override the `output_results()` method.

Recall

Often, we'll spot really good ideas that are repeated; the repetition can form a recognizable pattern. Exploiting a pattern-based approach to software design can save the developer from wasting time trying to reinvent something already well understood. In this chapter, we looked at a few more advanced design patterns:

- An Adapter class is a way to insert an intermediary so a client can make use of an existing class even when the class is not a perfect match. The software adapter parallels the idea of USB hardware adapters between various kinds of devices with various USB interface connectors.
- The Façade pattern is a way to create a unified interface over a number of objects. The idea parallels the façade of a building that unifies separate floors, rooms, and halls into a single space.
- We can leverage the Flyweight pattern to implement a kind of lazy initialization. Instead of copying objects, we can design Flyweight classes that share a common pool of data, minimizing or avoiding initialization entirely.
- When we have closely related classes of objects, the Abstract Factory pattern can be used to build a class that can emit instances that will work together.

- The Composition pattern is widely used for complex document types. It covers programming languages, natural languages, and markup languages, including XML and HTML. Even something like the filesystem with a hierarchy of directories and files fits this design pattern.
- When we have a number of similar, complex classes, it seems appropriate to create a class following the Template pattern. We can leave gaps or openings in the template, into which we can inject any unique features.

These patterns can help a designer focus on accepted, good design practices. Each problem is, of course, unique, so the patterns must be adapted. It's often better to make an adaptation to a known pattern and avoid trying to invent something completely new.

Exercises

Add the `os` and `pathlib` calls to implement the methods for the `File` and `Folder` classes in the *The Composite pattern* section. The `copy()` method on `File` will need to read and write the bytes of a file. The `copy()` method on `Folder` is quite a bit more complicated, as you first have to duplicate the folder, and then recursively copy each of its children to the new location. The examples we provided update the internal data structure, but don't apply changes to the operating system. Be careful about testing this in isolated directories. You don't want to accidentally destroy important files.

The Composite example has an alternative design, also. Consider how this could be built using a Façade class to contain operations from `pathlib` and `shutil`. Which implementation of Folder-File move, copy, and remove operations seems easier to understand? Which is easier to test?

As in the previous chapter, look at the patterns we've discussed and consider ideal places where you might implement them. You may want to apply the Adapter pattern to existing code, as it is usually applicable when interfacing with existing libraries, rather than new code. How can you use an Adapter to force two interfaces to interact with each other correctly?

Can you think of a system complex enough to justify using the Façade pattern? Consider how façades are used in real-life situations, such as the driver-facing interface of a car, or the control panel in a factory. It is similar in software, except the users of the façade interface are other programmers, rather than people trained to use it. Are there complex systems in your latest project that could benefit from the Façade pattern?

It's possible you don't have any huge, memory-consuming code that would benefit from the Flyweight pattern, but can you think of situations where it might be useful? Anywhere that large

amounts of overlapping data need to be processed, a Flyweight is waiting to be used. Would it be useful in the banking industry? In web applications? At what point does adopting the Flyweight pattern make sense? When is it overkill?

The Abstract Factory pattern, or the somewhat more Pythonic derivatives we discussed, can be very useful for creating one-touch-configurable systems. Can you think of places where such systems are useful?

The Composite pattern applies in a number of places. There are tree-like structures all around us in programming. Some of them, like our file hierarchy example, are blatant; others are fairly subtle. What situations might arise where the Composite pattern would be useful? Can you think of places where you can use it in your own code? What if you adapted the pattern slightly; for example, to contain different types of leaf or composite nodes for different types of objects?

The `ast` module provides a composite tree structure for Python code. A particularly useful thing is to use the `ast` module to locate all of the `import` statements in some code. This can help confirm that a project's list of required modules, often in a `requirements.txt` file, is complete and consistent.

A Template method is helpful when decomposing a complex operation so it is open to extension. When we look back at the Façade examples, where we're finding files and then executing PlantUML to make images from the UML source files, this seems to be a potential for a Template. What if we want both PNG files as well as LaTeXfiles from PlantUML? What is the common processing that should be part of a Template? What is the unique processing for each format?

Summary

In this chapter, we went into detail on several more design patterns, covering their canonical descriptions as well as alternatives for implementing them in Python. We saw how Python is often more flexible and versatile than traditional object-oriented languages. The Adapter pattern is useful for matching interfaces, while the Façade pattern is suited to simplifying them. Flyweight is a complicated pattern and only useful if memory optimization is required. Abstract Factories allow the runtime separation of implementations depending on configuration or system information. The Composite pattern is used universally for tree-like structures. A Template method can be helpful for breaking complex operations into steps to avoid repeating the common features.

This is the last of the truly object-oriented design chapters in this book. In the next chapter, we'll discuss how important it is to test Python programs, and how to do it, focusing on object-oriented principles. Then, in the final chapter, we'll look at the concurrency features of Python and how to exploit them to get work done more quickly.

Join our community Discord space

Join our Python Discord workspace to discuss and know more about the book: `https://packt.link/dHrHU`

13

Testing Object-Oriented Programs

Skilled Python programmers agree that testing is one of the most important aspects of software development. Even though this chapter is placed near the end of the book, it is not an afterthought; everything we have studied so far will help us when writing tests. In this chapter, we'll look at the following topics:

- The importance of unit testing and test-driven development
- The standard library `unittest` module
- The `pytest` tool
- The `mock` module
- Code coverage

We'll start with some of the fundamental reasons why automated software testing is so important.

Why test?

Many programmers already know how important it is to test their code. If you're among them, feel free to skim this section. You'll find the next section — where we actually see how to create tests in Python — much more interesting.

If you're not convinced of the importance of testing, we remind you that without any tests, code will be broken, and no one has any way to know it. Read on!

Some people argue that testing is more important in Python code because of its dynamic nature; compiled languages such as Java and C++ are occasionally thought to be somehow *safer* because they enforce type checking at compile time. However, Python tests rarely check types. Tests check values. They make sure that the right attributes have been set at the right time or that the sequence has the right length, order, and values. These higher-level concepts need to be tested in any language. The real reason Python programmers test more than programmers of other languages is that it is so easy to test in Python!

But why test? Do we really need to test? What if we didn't test? To answer those questions, reflect on the last time you wrote any code. Did it run correctly the first time? Was it free of syntax errors? Was it free of logic problems? It's possible, in principle, to type in code that's perfect once in a while. As a practical matter, the number of obvious syntax errors that had to be corrected is an indicator that perhaps there are more subtle logic errors that also had to be corrected.

We don't need a formal, separate test to make sure our code works. Running the program, as we generally do, and fixing the errors is a crude form of testing. Python's interactive interpreter and near-zero compile times make it easy to write a few lines of code and run the program to make sure those lines are doing what is expected. While acceptable at the beginning of a project, this turns into a liability that grows over time. Attempting to change a few lines of code can affect parts of the program that we didn't realize would be influenced by the changes, and without automated tests, we wouldn't know what we broke. Attempts at redesigns or even small optimization rewrites can be plagued with problems. Furthermore, as a program grows, the number of paths that the interpreter can take through that code also grows, and it quickly becomes impossible or a crude manual test to exercise all the logic paths.

To assure ourselves and to demonstrate to others that our software works, we write automated tests. These are programs that automatically run certain inputs through other programs or parts of programs. We can run these test programs in seconds and cover far more potential input situations than one programmer would think to test every time they change something.

> *Software features that can't be demonstrated by automated tests simply don't exist.*
>
> *— Extreme Programming Explained, Kent Beck*

There are four main reasons to write tests:

- To ensure that code is working the way the developer thinks it should
- To ensure that code continues working when we make changes
- To ensure that the developer understood the requirements
- To ensure that the code we are writing has a maintainable interface

When we have automated tests, we can run them every time we change code, whether it is during initial development or maintenance releases. Testing can confirm that we didn't inadvertently break anything when adding or extending features.

The last two of the preceding points have interesting consequences. When we write tests, it helps us design the API, interface, or pattern that code takes. Thus, if we misunderstood the requirements, writing a test can help highlight the misunderstanding. From the other side, if we're not certain how we want to design a class, we can write a test that interacts with that class so we have an idea of the most natural way to confirm that the interface works. In fact, it is often beneficial to write the tests before we write the code we are testing.

There are some other interesting consequences of focusing on software testing. We'll look at three of these consequences:

- Using tests to drive development
- Managing different objectives for testing
- Having a consistent pattern for test scenarios

Let's start with using tests to drive the development effort.

Test-driven development

Write tests first is the mantra of test-driven development. Test-driven development takes the *untested code is broken code* concept one step further and suggests that only unwritten code should be untested. We don't write any code until after we have written the tests that will prove it works. The first time we run a test, it should fail, since the code hasn't been written. Then, we write the code that ensures the test passes, and then write another test for the next segment of code.

Test-driven development can be fun; it allows us to build little puzzles to solve. Then, we implement the code to solve those puzzles. After that, we make a more complicated puzzle, and we write code that solves the new puzzle without unsolving the previous one.

There are two goals of the test-driven methodology. The first is to ensure that tests really get written.

Secondly, writing tests first forces us to consider exactly how the code will be used. It tells us what methods objects need to have and how attributes will be accessed. It helps us break up the initial problem into smaller, testable problems, and then recombine the tested solutions into larger, also tested, solutions. Writing tests can thus become a part of the design process. Often, when we're writing a test for a new object, we discover anomalies in the design that force us to consider new aspects of the software.

Testing makes software better. Writing tests before we release the software makes it better before the final code is written.

All the code examined in the book has been run through an automated test suite. It's the only way to be absolutely sure that the examples are rock-solid, working code.

Testing objectives

We have a number of distinct objectives for running tests. These are often called types of testing, but the word "type" is heavily overused in the software industry. In this chapter, we'll look at only two of these testing goals:

- **Unit tests** confirm that the software components work in isolation. We'll focus on this first, since Fowler's Test Pyramid seems to suggest that unit testing creates the most value. If the various classes and functions each adhere to their interfaces and produce the expected results, then integrating them is also going to work nicely and have relatively few surprises. It's common to use the **coverage** tool to be sure that all the lines of code are exercised as part of the unit test suite.
- **Integration tests** — unsurprisingly — confirm that software components work when integrated. Integration tests are sometimes called system tests, functional tests, and acceptance tests, among others. When an integration test fails, it often means that an interface wasn't defined properly, or a unit test didn't include some edge case that was exposed through the integration with other components. Integration testing depends on having good unit tests, making it secondary in importance.

Note that "unit" isn't formally defined by the Python language. This is an intentional choice. A unit of code is often a single function, or a single class. It can be a single module, also. The definition gives us a little flexibility to identify and isolate units of code.

While there are many distinct objectives for tests, the techniques used for testing tend to be similar. For additional material, see *Types of Software Testing* (`https://www.softwaretestinghelp.com/types-of-software-testing/`) for a list of over 40 different types of testing objectives; the complete list can be overwhelming, which is why we will only focus on unit tests and integration tests. All tests have a common pattern to them, and we'll look at a general pattern of testing next.

Testing patterns

Writing code is often challenging. We need to figure out what the internal state of the object is and what state changes it undergoes, and determine the other objects it collaborates with. Throughout the book, we've provided a number of common patterns for designing classes.

Tests, in a way, are simpler than class definitions, and all have essentially the same pattern, called a scenario:

```
SCENARIO: use case for some feature
  GIVEN some precondition(s) for a scenario
  WHEN we exercise some method of a class
  THEN some state change(s) or side effect(s) will occur that we can confirm
```

In some cases, the preconditions can be complex, or perhaps the state changes or side effects are complex. They might be so complex that we have to break them into multiple steps. What's important about this three-part pattern is how it disentangles the setup, execution, and expected results from each other. This model applies to a wide variety of tests. If we want to make sure the water's hot enough to make another cup of tea, we'll follow a similar set of steps:

- `GIVEN` a kettle of water on the stove
- `AND` the burner is off
- `WHEN` we flip open the lid on the kettle
- `THEN` we see steam escaping

This pattern is quite handy for making sure we have a clear setup and an observable result.

These three steps are also called **Arrange**, **Act**, and **Assert**. We like the Given-When-Then structure because it's part of the Gherkin language for writing test scenarios.

Let's say we need to write a function to compute an average of a list of numbers, excluding `None` values that might be in the sequence. We might start out like this:

```
def average(data: list[int | None]) -> float:
    """
    GIVEN a list, data = [1, 2, None, 3, 4]
    WHEN we compute m = average(data)
    THEN the result, m, is 2.5
    """
    pass
```

We've roughed out a definition of the function, with a summary of how we think it should behave. The **GIVEN** step defines some data for our test case. The **WHEN** step defines precisely what we're going to be doing. Finally, the **THEN** step describes the expected results. The automated test tool can compare actual results against the stated expectation and report back if the test fails. We can then implement this using the preferred test framework. The ways packages such as `unittest` and `pytest` implement the concept differ slightly, but the core concept remains the same in both frameworks. Once that's done, running the test should reveal an expected failure; we can start implementing the real code, given this test as a clear goal line we want to cross.

Some techniques that can help design test cases are **equivalence partitioning** and **boundary value analysis**. These help us decompose the domain of all possible inputs to a method or function into partitions. A common example is locating two partitions: one with "valid data" and the other with "invalid data." There are often edge cases, special exceptions, and other considerations that create more partitions. Given the partitions, the values at the boundaries of the partitions become interesting values to use in test cases. See What is boundary value analysis and equivalent partitions? at `https://www.softwaretestinghelp.com` for more information.

We'll start by looking at the built-in testing framework, `unittest`. It has the disadvantage of being a bit wordy and complicated-looking. It has the advantage of being built-in and usable immediately; no further installs are required.

Unit testing with unittest

Let's start our exploration with Python's built-in test library. This library provides a common object-oriented interface for *unit tests*. The Python library for this is called, unsurprisingly, `unittest`. It provides several tools for creating and running unit tests, the most important being the `TestCase`

class. (The names follow a Java naming style, so many of the method names don't look very Pythonic.) The `TestCase` class provides a set of methods that allow us to compare values, set up tests, and clean up when they have finished.

When we want to write a set of unit tests for a specific task, we create a subclass of `TestCase` and write individual methods to do the actual testing. These methods must all start with the name `test`. When this convention is followed, the tests automatically run as part of the test process. For simple examples, we can bundle the `GIVEN`, `WHEN`, and `THEN` concepts into the test method. Here's a very simple example:

```
import unittest


class CheckNumbers(unittest.TestCase):
    def test_int_float(self) -> None:
        self.assertEqual(1, 1.0)
```

This code subclasses the `TestCase` class and adds a method that calls the `TestCase.assertEqual()` method. The **GIVEN** step is a pair of values, 1 and 1.0. The **WHEN** step is a kind of degenerate example because there's no new object created and no state change happening. The **THEN** step is the assertion that the two values will test as equal.

When we run the test case, this method will either succeed silently or raise an exception, depending on whether the two parameters are equal. If we run this code, the `main` function from `unittest` will give us the following output:

```
.
----------------------------------------------------------------
Ran 1 test in 0.000s
OK
```

Did you know that floats and integers can be compared as equal?

Let's add a failing test to this test case, as follows:

```
    def test_str_float(self) -> None:
        self.assertEqual(1, "1")
```

The output of this code is more sinister, as integers and strings are not considered equal:

```
.F
======================================================================
FAIL: test_str_float (__main__.CheckNumbers)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "first_unittest.py", line 9, in test_str_float
    self.assertEqual(1, "1")
AssertionError: 1 != '1'
----------------------------------------------------------------------
Ran 2 tests in 0.001s
FAILED (failures=1)
```

The dot on the first line indicates that the first test in the suite passed successfully; the letter `F` after the dot shows that the second test failed. Then, at the end, it gives us an informative summary telling us how and where the test failed, along with a count of the number of failures.

Even the OS-level return code provides a useful summary. The return code is zero if all tests pass and non-zero if any tests fail. This helps when building continuous integration tools: if the `unittest` run fails, the proposed change shouldn't be permitted.

We can have as many test methods on one `TestCase` class as we like. As long as the method name begins with `test`, the test runner will execute each one as a separate, isolated test.

An important design consideration is to keep each test completely independent of other tests.

Tests need to be independent.

Results or calculations from any test should have no impact on any other test.

In order to keep tests isolated from each other, we may have several tests with a common **GIVEN** step, implemented by a common `setUp()` method. This suggests that we'll often have classes that are similar, and we'll need to use inheritance to design the tests so they can share features and still remain completely independent.

The key to writing good unit tests is keeping each test method as short as possible, testing a small unit of code with each test case. If our code does not seem to naturally break up into small, testable units, it's probably a sign that the code needs to be redesigned. The *Imitating objects using mocks* section, later in this chapter, provides a way to isolate objects for testing purposes.

The `unittest` module imposes a requirement to structure tests as a class definition. This is — in

some ways — a bit of overhead. The `pytest` package has slightly more clever test discovery and a slightly more flexible way to construct tests as functions instead of methods of a class. We'll look at `pytest` next.

Unit testing with pytest

We can create unit tests using a library that provides a common framework for the test scenarios, along with a test runner to execute the tests and log results. Unit tests focus on testing the least amount of code possible in any one test. One of the more popular alternatives to the standard library `unittest` is the `pytest` package. This has the advantage of letting us write smaller and clearer test cases. The lack of overhead makes this a desirable alternative.

Since `pytest` is not part of the standard library, you'll need to download and install it. You can get it from the `pytest` home page at `https://docs.pytest.org/en/stable/`. You can install it with any of the installers or environment managers.

In a Terminal window, activate the virtual environment for your current project. (If you're using venv, for example, you might use `source .venv/bin/activate`.) Then, use an OS command like the following:

```
% python -m pip install pytest
```

When using tools such as **uv** or **poetry**, you'll often add this to the development dependencies with a command like this:

```
% uv add pytest --dev
```

The `pytest` tool can use a substantially different test layout from the `unittest` module. It doesn't require test cases to be subclasses of `unittest.TestCase`. Instead, it takes advantage of the fact that Python functions are first-class objects and allows any properly named function to behave like a test. Rather than providing a bunch of custom methods for asserting equality, it uses the `assert` statement to verify results. This makes tests simpler, more readable, and, consequently, easier to maintain.

When we run the `pytest` tool, it starts in the current folder and searches for any modules or sub-packages with names beginning with the characters `test_`. (The module name needs to include the _ character.) If any functions in this module also start with `test` (no _ required), they will

be executed as individual tests. Furthermore, if there are any classes in the module whose name starts with `Test`, any methods on that class that start with `test_` will also be executed in the test environment.

It also searches for tests in a folder named—unsurprisingly—`tests`. Because of this, it's common to have code broken up into two folders: the `src/` directory contains the working module, library, or application, while the `tests/` directory contains all the test cases.

Using the following code, let's port the simple `unittest` example we wrote earlier to `pytest`:

```
def test_int_float() -> None:
    assert 1 == 1.0
```

For the same test, we've reduced six lines of class definition to two lines of more readable code.

However, we are not forbidden from writing class-based tests. Classes can be useful for grouping related tests together or for tests that need to access related attributes or methods on the class. The following example shows an extended class with a passing and a failing test; we'll see that the error output is more comprehensive than that provided by the `unittest` module:

```
class TestNumbers:
    def test_int_float(self) -> None:
        assert 1 == 1.0
```

```
    def test_int_str(self) -> None:
        assert 1 == "1"  # type: ignore [comparison-overlap]
```

Notice that the class doesn't have to extend any special objects to be discovered as a test case. If you want, you can use `unittest.TestCase` as the base class; the `pytest` tool will find and run these, also.

We'll address the `# type: ignore` comment later, first, let's run `python -m pytest tests/test_with_pytest.py`. The output looks as follows:

```
% python -m pytest tests/test_with_pytest.py
======================== test session starts ========================
platform darwin -- Python 3.9.0, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
rootdir: /path/to/ch_13
collected 2 items
```

```
tests/test_with_pytest.py .F                                [100%]
============================ FAILURES ============================
_____________________ TestNumbers.test_int_str _____________________
self = <test_with_pytest.TestNumbers object at 0x7fb557f1a370>
    def test_int_str(self) -> None:
>       assert 1 == "1"
E       AssertionError: assert 1 == "1"
tests/test_with_pytest.py:15: AssertionError
====================== short test summary info ======================
FAILED tests/test_with_pytest.py::TestNumbers::test_int_str - Asse...
=================== 1 failed, 1 passed in 0.07s ====================
```

The output starts with some useful information about the platform and interpreter. This can be useful for sharing or discussing bugs across disparate systems. The third line tells us the name of the file being tested (if there are multiple test modules picked up, they will all be displayed), followed by the familiar `.F` that we saw in the `unittest` module: the `.` character indicates a passing test, while the letter `F` reports a failure.

After all tests have run, the error output for each of them is displayed. The tool presents a summary of local variables (there is only one in this example: the `self` parameter passed into the function), the source code where the error occurred, and a summary of the error message. In addition, if an exception other than `AssertionError` is raised, `pytest` will present us with a complete traceback, including source code references.

By default, `pytest` suppresses output from the `print()` function if the test is successful. This is useful for test debugging; when a test is failing, we can add `print()` statements to the test to check the values of specific variables and attributes as the test runs. If the test fails, these values are output to help with diagnosis. However, once the test is successful, the `print()` output is not displayed, and they are easily ignored. We don't have to clean up the test output by removing the `print()` output. If the tests ever fail again, due to future changes, the debugging output will be immediately available.

Interestingly, this use of the `assert` statement exposes a potential problem to **mypy**. When we use the `assert` statement, **mypy** can scrutinize the types, and will alert us to a potential problem with `assert 1 == "1"`. This code is unlikely to be right. And, of course, it does fail!

We've looked at how `pytest` supports the **WHEN** and **THEN** steps of a test scenario using a function and the `assert` statement. Now, we need to look more closely at how to handle **GIVEN**

steps. There are two ways to establish the **GIVEN** precondition for a test; we'll start with one that works for simple cases.

pytest's setup and teardown functions

The `pytest` tool offers setup and teardown capabilities, similar to the methods used in `unittest`, but it provides even more flexibility. We'll discuss these general functions briefly; `pytest` provides us with a powerful fixtures capability, which we'll discuss in the next section.

If we are writing class-based tests, we can use two methods called `setup_method()` and `teardown_method()`. They are called before and after each test method in the class to perform setup and cleanup duties, respectively.

In addition, `pytest` provides other setup and teardown functions to give us more control over when preparation and cleanup code is executed. The `setup_class()` and `teardown_class()` methods are expected to be class methods; they accept a single argument representing the class in question (there is no `self` argument because there's no instance; instead, the class is provided). These methods are run by `pytest` for the class as a whole rather than on each test within the class.

Finally, we have the `setup_module()` and `teardown_module()` functions, which are run by `pytest` immediately before and after all tests (in functions or classes) in that module. These can be useful for *one-time* setup, such as creating a socket or database connection that will be used by all tests in the module. Be careful with this one, as it can accidentally introduce dependencies between tests if some object state isn't correctly cleaned up between tests.

Let's look at an example that illustrates exactly when it happens. We'll break this into three parts. First is some module-level setup and teardown:

```
from collections.abc import Callable
from typing import Any


def setup_module(module: Any) -> None:
    print(f"setting up MODULE {module.__name__}")


def teardown_module(module: Any) -> None:
    print(f"tearing down MODULE {module.__name__}")
```

```
def test_a_function() -> None:
    print("RUNNING TEST FUNCTION")
```

Here's an abstract base class for test cases that has class-level setup and teardown:

```
class BaseTest:
    @classmethod
    def setup_class(cls: type["BaseTest"]) -> None:
        print(f"setting up CLASS {cls.__name__}")

    @classmethod
    def teardown_class(cls: type["BaseTest"]) -> None:
        print(f"tearing down CLASS {cls.__name__}\n")

    def setup_method(self, method: Callable[[], None]) -> None:
        print(f"setting up METHOD {method.__name__}")

    def teardown_method(self, method: Callable[[], None]) -> None:
        print(f"tearing down METHOD {method.__name__}")
```

Finally, there are some concrete classes that contain ordinary test cases but get their setup and teardown from a base class:

```
class TestClass1(BaseTest):
    def test_method_1(self) -> None:
        print("RUNNING METHOD 1-1")

    def test_method_2(self) -> None:
        print("RUNNING METHOD 1-2")


class TestClass2(BaseTest):
    def test_method_1(self) -> None:
        print("RUNNING METHOD 2-1")

    def test_method_2(self) -> None:
        print("RUNNING METHOD 2-2")
```

The purpose of the `BaseTest` class is to extract four methods that are otherwise identical to the two test classes, and use inheritance to reduce the amount of duplicate code. So, from the point of view of `pytest`, the two subclasses have not only two test methods each but also two setup and

two teardown methods (one at the class level and one at the method level).

If we run these tests using `pytest` with the `print()` function output suppression disabled (by passing the `-s` or `-capture=no` flag), they show us when the various functions are called in relation to the tests themselves:

```
% python -m pytest --capture=no tests/test_setup_teardown.py
======================== test session starts ========================
platform darwin -- Python 3.9.0, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
rootdir: /.../ch_13
collected 5 items
tests/test_setup_teardown.py setting up MODULE test_setup_teardown
RUNNING TEST FUNCTION
.setting up CLASS TestClass1
setting up METHOD test_method_1
RUNNING METHOD 1-1
.tearing down METHOD test_method_1
setting up METHOD test_method_2
RUNNING METHOD 1-2
.tearing down METHOD test_method_2
tearing down CLASS TestClass1
setting up CLASS TestClass2
setting up METHOD test_method_1
RUNNING METHOD 2-1
.tearing down METHOD test_method_1
setting up METHOD test_method_2
RUNNING METHOD 2-2
.tearing down METHOD test_method_2
tearing down CLASS TestClass2
tearing down MODULE test_setup_teardown
======================== 5 passed in 0.01s ==========================
```

The setup and teardown methods for the module as a whole are executed at the beginning and end of the session. Then, the lone module-level test function is run. Next, the setup method for the first class is executed, followed by the two tests for that class. These tests are each individually wrapped in separate `setup_method()` and `teardown_method()` calls. After the tests have executed, the teardown method on the class is called. The same sequence happens for the second class, before the `teardown_module()` method is finally called, exactly once.

While these function names provide a lot of options for testing, we'll often have setup conditions that are shared across multiple test scenarios. These can be reused via a composition-based design;

`pytest` calls these "fixtures." We'll look at fixtures next.

pytest fixtures for setup and teardown

One of the most common uses for the various setup functions is to ensure that the **GIVEN** step of a test is executed. This often involves creating objects and making sure that certain class or module variables have known values before a test method is run.

In addition to a set of special method names for a test class, `pytest` offers a completely different way of doing this, using what are known as **fixtures**. Fixtures are functions to build the **GIVEN** condition, prior to a test's **WHEN** and **THEN** steps.

The `pytest` tool has a number of built-in fixtures. We can define fixtures in a configuration file and reuse them, and we can define unique fixtures as part of our tests. This allows us to separate configuration from the execution of tests, allowing fixtures to be used across multiple classes and modules.

Let's look at a class that does a few computations that we need to test:

```
from collections import defaultdict


class StatsList(list[float | None]):
    """Stats with None objects rejected"""

    def mean(self) -> float:
        clean = list(filter(None, self))
        return sum(clean) / len(clean)

    def median(self) -> float:
        clean = list(filter(None, self))
        if len(clean) % 2:
            return clean[len(clean) // 2]
        else:
            idx = len(clean) // 2
            return (clean[idx] + clean[idx - 1]) / 2

    def mode(self) -> list[float]:
        freqs: defaultdict[float, int] = defaultdict(int)
        for item in filter(None, self):
            freqs[item] += 1
        mode_freq = max(freqs.values())
```

```
        modes = [item for item, value in freqs.items() if value ==
        mode_freq]
        return modes
```

This class extends the built-in `list` class to add three statistical summary methods, `mean()`, `median()`, and `mode()`. For each method, we need to have some set of data we can use; this configuration of a `StatsList` with known data is the fixture we'll be testing.

To use a fixture to create the **GIVEN** precondition, we add the fixture name as a parameter to our test function. When a test runs, the names of the test function's parameters will be located in the collection of fixtures, and those fixture-creating functions will be executed for us automatically.

For example, to test the `StatsList` class, we want to repeatedly provide a list of valid integers. We can write our tests as follows:

```
import pytest
from stats import StatsList


@pytest.fixture
def valid_stats() -> StatsList:
    return StatsList([1, 2, 2, 3, 3, 4])


def test_mean(valid_stats: StatsList) -> None:
    assert valid_stats.mean() == 2.5


def test_median(valid_stats: StatsList) -> None:
    assert valid_stats.median() == 2.5
    valid_stats.append(4)
    assert valid_stats.median() == 3


def test_mode(valid_stats: StatsList) -> None:
    assert valid_stats.mode() == [2, 3]
    valid_stats.remove(2)
    assert valid_stats.mode() == [3]
```

Each of the three test functions accepts a parameter named `valid_stats`; this parameter is created by `pytest` automatically calling the `valid_stats` function for us. The function was decorated with

`@pytest.fixture` so it could be used in this special way by `pytest`.

And yes, the fixture name must match the parameter name. The **pytest** runtime looks for functions with the `@fixture` decorator that match the parameter name.

Fixtures can do a lot more than return simple objects. A `request` object can be passed into the fixture factory to provide extremely useful methods and attributes to modify the fixture's behavior. The `module`, `cls`, and `function` attributes of the `request` object allow us to see exactly which test is requesting the fixture. The `config` attribute of the `request` object allows us to check command-line arguments and a great deal of other configuration data.

If we implement the fixture as a generator, it can also run cleanup code after each test is run. This provides the equivalent of a teardown method on a per-fixture basis. We can use it to clean up files, close connections, empty lists, or reset queues. For unit tests, where items are isolated, a mock object is a better idea than performing a teardown on a stateful object. See the *Imitating objects using mocks* section later in this chapter for a simpler approach that's ideal for unit testing.

For integration tests, we might want to test some code that creates, deletes, or updates files. We'll often use the `pytest` `tmp_path` fixture to create a directory that can be deleted later, saving us from having to do a teardown in a test. While rarely needed for unit testing, a teardown is helpful for stopping subprocesses or removing database changes that are part of an integration test. We'll see this a little later in this section. First, let's look at a small example of a fixture with setup and teardown capabilities.

To get started on the concept of a fixture that does both setup and teardown, here's a little bit of code that makes a backup copy of a file and writes a new file with a checksum of an existing file:

```
from pathlib import Path
import hashlib


def checksum(source: Path, checksum_path: Path) -> None:
    if checksum_path.exists():
        backup = checksum_path.with_stem(f"(old) {checksum_path.stem}")
        backup.write_text(checksum_path.read_text())
    checksum = hashlib.sha256(source.read_bytes())
    checksum_path.write_text(f"{source.name} {checksum.hexdigest()}\n")
```

There are two scenarios:

- The source file exists; a new checksum is added to the directory
- The source file and a checksum file both exist; in this case, the old checksum is copied to a backup location and a new checksum is written

We won't test both scenarios, but we will show how a fixture can create — and then delete — the files required for a test sequence. We'll focus on the second scenario because it's more complex. We'll break the testing into two parts, starting with the fixture:

```
from collections.abc import Iterator
from pathlib import Path
import sys
import pytest
import checksum_writer


@pytest.fixture
def working_directory(tmp_path: Path) -> Iterator[tuple[Path, Path]]:
    working = tmp_path / "some_directory"
    working.mkdir()
    source = working / "data.txt"
    source.write_bytes(b"Hello, world!\n")
    checksum = working / "checksum.txt"
    checksum.write_text("data.txt Old_Checksum")
    yield source, checksum
    checksum.unlink()
    source.unlink()
```

The `yield` statement is the secret for making this work. A fixture is a generator that produces one result and then waits for the next request of a value. The first result that's created is the result of a number of steps: a working directory is created, a source file is created in the working directory, and then an old checksum file is created. The `yield` statement provides two paths to the test and waits for the next request. This work completes the **GIVEN** condition setup for the test.

When the test function finishes, `pytest` will try to get one final item from this fixture. This lets the function unlink the files, removing them. There's no return nor any second `yield` statement: the generator simply exits. In addition to leveraging the generator protocol, the `working_directory` fixture relies on a built-in `pytest` fixture, `tmp_path`, to create a temporary working location for this test.

Here's the test that uses this `working_directory` fixture:

```
@pytest.mark.skipif(sys.version_info < (3, 9), reason="requires python3.9
feature")
def test_checksum(working_directory: tuple[Path, Path]) -> None:
    source_path, old_checksum_path = working_directory
    checksum_writer.checksum(source_path, old_checksum_path)
    backup = old_checksum_path.with_stem(f"(old) {old_checksum_path.stem}")
    assert backup.exists()
    assert old_checksum_path.exists()
    name, checksum = old_checksum_path.read_text().rstrip().split()
    assert name == source_path.name
    assert (
        checksum == "d9014c4624844aa5bac314773d6b689a"
        "d467fa4e1d1a50a1b8a99d5a95f72ff5"
    )
```

The test is marked with a `skipif` condition because this test won't work in Python versions less than 3.9; the `with_stem()` method of a `Path` isn't part of the older `pathlib` implementation. This assures us that the test is counted but noted as inappropriate for a specific Python release. We'll return to this in the *Skipping tests with pytest* section later in this chapter.

The reference to the `working_directory` fixture forces `pytest` to execute the fixture function, providing the test scenario with two paths to be used as part of the **GIVEN** condition prior to testing. The **WHEN** step evaluates the `checksum_writer.checksum()` function with these two paths. The **THEN** steps are a sequence of `assert` statements to make sure that the files are created with the expected values. After the test is run, `pytest` will use `next()` to get another item from the fixture; this action executes the code after the `yield`, resulting in a teardown of two files after the test has completed.

When testing components in isolation, we won't often need to use the teardown feature of a fixture. For integration tests, however, where a number of components are used in concert, it may be necessary to stop processes, remove files, or reset the database state. In the next section, we'll look at a more sophisticated fixture. This kind of fixture can be used for more than a single test scenario.

More sophisticated fixtures

We can pass a `scope` parameter to create a fixture that needs to last longer than one test. This is useful when setting up an expensive operation that can be reused by multiple tests. This works as long as the resource reuse doesn't break the atomic or unit nature of the test: one unit test should not rely on, and should not be impacted by, any other unit test.

As an example, we'll define a server that's part of a client-server application. We want multiple web servers to send their log messages to a single centralized log. In addition to isolated unit tests, we need to have an integration test. This test makes sure the web server and the log collector properly integrate with each other. The integration test will need to start and stop this log collection server.

There are at least three levels to the testing pyramid. Unit tests are the foundation, exercising each component in isolation. Integration tests are the middle of the pyramid, making sure that the components integrate properly with each other. A system test or acceptance test is the top of the pyramid, making sure that the entire suite of software does what it claims.

We'll look at a log collection server that accepts messages and writes them to a single, central file. These messages are defined by the `logging` module's `SocketHandler`. We can depict each message as a block of bytes with a header and a payload. In the following table, we've shown the structure using slices of the block of bytes.

Here's how a message is defined:

Slice [start:stop]	Meaning	Python function for parsing
`[0:4]`	payload size	`struct.unpack(">L", bytes)`
`[4:payload_size+4]`	payload	`pickle.loads(bytes)`

Table 13.1: Structure of a Log Message

The size of the header is shown as a four-byte slice, but the size shown here can be misleading. The header is formally and officially defined by the `">L"` format string used by the `struct` module. The `struct` module has a function, `calcsize()`, to compute the actual length of the format string. Instead of using a literal 4, which is derived from the size of the `">L"` format, our code will derive the size, `size_bytes`, from the size format string, `size_format`. Using one proper source, `size_format`, for both pieces of information follows the design principle of "Don't Repeat Yourself."

Here's an example buffer with a message from the `logging` module embedded in it. The first line is the header with the payload size, a four-byte value. The next lines are the pickled data for a log message:

```
b'\x00\x00\x02d'
b'\}q\x00(X\x04\x00\x00\x00nameq\x01X\x03\x00\x00\x00appq...
...
\x19X\n\x00\x00\x00MainThreadq\x1aX\x0b\x00\x00\x00processNameq\x1bX\x0b...'
```

To read these messages, we'll need to collect the payload size bytes first. Then, we can consume the payload that follows. Here's the socket handler class that reads the headers and the payloads and writes them to a file:

```
import json
from pathlib import Path
import pickle
import socketserver
import struct
import sys
from typing import TextIO


class LogDataCatcher(socketserver.BaseRequestHandler):
    log_file: TextIO
    count: int = 0
    size_format = ">L"
    size_bytes = struct.calcsize(size_format)

    def handle(self) -> None:
        size_header_bytes = self.request.recv(LogDataCatcher.size_bytes)
        while size_header_bytes:
            payload_size = struct.unpack(LogDataCatcher.size_format,
            size_header_bytes)
            print(f"{size_header_bytes=} {payload_size=}", file=sys.stderr)
            payload_bytes = self.request.recv(payload_size[0])
            print(f"{len(payload_bytes)=}", file=sys.stderr)
            payload = pickle.loads(payload_bytes)
            LogDataCatcher.count += 1
            print(f"{self.client_address[0]} {LogDataCatcher.count}
            {payload!r}")
            self.log_file.write(json.dumps(payload) + "\n")
            try:
                size_header_bytes =
                self.request.recv(LogDataCatcher.size_bytes)
            except (ConnectionResetError, BrokenPipeError):
                break
```

The `socketserver.TCPServer` object will listen for connection requests from a client. When a client connects, it will create an instance of the `LogDataCatcher` class and evaluate the `handle()` method of that object to gather data from that client. The `handle()` method decodes the size and payload with a two-step dance: get the size, `payload_size`, then get the payload. `pickle.loads()` will load

a Python object from the payload bytes. This is serialized into JSON notation using `json.dumps()` and written to the open file. Once a message has been handled, we can try to read the next few bytes to see whether there's more data waiting. This server will absorb messages from the client until the connection is dropped, leading to an error in the read and an exit from the `while` statement.

This log collection server can absorb logging messages from an application anywhere in a network. This example implementation is single-threaded, meaning it only handles one client at a time. We can use additional mixins to create a multithreaded server that will accept messages from multiple sources. In this example, we want to focus on testing a single application that depends on this server.

For completeness, here's the main function and script that starts the server running:

```
def main(host: str, port: int, target: Path) -> None:
    with target.open("w") as unified_log:
        LogDataCatcher.log_file = unified_log
        with socketserver.TCPServer((host, port), LogDataCatcher) as server:
            server.serve_forever()


if __name__ == "__main__":
    HOST, PORT = "localhost", 18842
    main(HOST, PORT, Path("one.log"))
```

As a practical matter, we might consider using the `argparse` module and the `os.environ` dictionary to provide these values to the application. For now, we've hardcoded these configuration details as literal values.

Here's the `remote_logging_app.py` application, which transmits log records to the log-catching server:

```
import logging
import logging.handlers
import sys
from math import factorial

logger = logging.getLogger("app")


def work(i: int) -> int:
```

```
    logger.info("Factorial %d", i)
    f = factorial(i)
    logger.info("Factorial(%d) = %d", i, f)
    return f


if __name__ == "__main__":
    HOST, PORT = "localhost", 18842
    socket_handler = logging.handlers.SocketHandler(HOST, PORT)
    stream_handler = logging.StreamHandler(sys.stderr)
    logging.basicConfig(handlers=[socket_handler, stream_handler],
    level=logging.INFO)

    for i in range(10):
        work(i)

    logging.shutdown()
```

This application creates two logging handlers. The `SocketHandler` instance will open a socket on the given server and port number and start writing bytes. The bytes will include headers and payloads. The `StreamHandler` instance will write to the terminal window; this is the default log handler that we would get if we didn't create any special handlers. We configure our logger with both handlers so that each log message goes to both our console and the stream server collecting the messages.

The actual work? That's almost lost in the overheads. We highlighted those two lines in the code block. It does a little bit of math to compute the factorial of some numbers. Each time we run this application, it should blast out 20 log messages.

To test the integration of the client and server, we need to start the server in a separate process. We don't want to start and stop it many times (that would take a while), so we will start it once and use it in multiple tests. We'll break this into two sections, starting with the two fixtures:

```
from collections.abc import Iterator
import logging
from pathlib import Path
import signal
import subprocess
import sys
import time
```

```
import pytest

import remote_logging_app
```

```
@pytest.fixture(scope="session")
def log_catcher() -> Iterator[None]:
    server_path = Path("src") / "log_catcher.py"
    print(f"Starting server {server_path}")
    p = subprocess.Popen(
        [sys.executable, str(server_path)],
        stdout=subprocess.PIPE,
        stderr=subprocess.STDOUT,
        text=True,
    )
    time.sleep(0.25)
    yield
    p.terminate()
    p.wait()
    if p.stdout:
        print(p.stdout.read())
    assert (
        p.returncode == 1 if sys.platform == "win32" else
        -signal.SIGTERM.value
    ), f"Error in watcher, returncode={p.returncode}"
```

```
@pytest.fixture
def logging_config() -> Iterator[None]:
    HOST, PORT = "localhost", 18842
    socket_handler = logging.handlers.SocketHandler(HOST, PORT)
    remote_logging_app.logger.addHandler(socket_handler)
    yield
    socket_handler.close()
    remote_logging_app.logger.removeHandler(socket_handler)
```

The `log_catcher` fixture will start the `log_catcher.py` server as a subprocess. This has a scope set to `"session"` in the `@fixture` decorator, which means it's done once for the whole testing session. The scope can be one of the `"function"`, `"class"`, `"module"`, `"package"`, or `"session"` strings, providing distinct places where the fixture is created and reused. The startup involves a tiny pause (250 ms) to allow the child process to get started and be ready to accept connections.

When this fixture reaches the `yield` statement, this part of the **GIVEN** test setup is done.

The `logging_config` fixture will tweak the log configuration for the `remote_logging_app` module that's being tested. When we look at the `work()` function in the `remote_logging_app.py` module, we can see that it expects a module-level `logger` object. This test fixture creates a `SocketHandler` object, adds this to the `logger`, and then executes the `yield` statement.

Once both of these fixtures have contributed to the **GIVEN** condition, we can define test cases that contain the **WHEN** steps. Here are two examples of two similar scenarios:

```
def test_1(log_catcher: None, logging_config: None) -> None:
    for i in range(10):
        remote_logging_app.work(i)


def test_2(log_catcher: None, logging_config: None) -> None:
    for i in range(1, 10):
        remote_logging_app.work(52 * i)
```

These two test scenarios both require the two fixtures. The `log_catcher` fixture, with a session scope, is prepared once and used for both tests. The `logging_config` fixture, however, has default scope, which means it's prepared for each test function.

The type hint of `None` follows the definition of the fixture as `Iterator[None]`. There's no value returned by the `yield` statement. For these tests, the setup operation is preparing the overall runtime environment by starting a process.

When a test function finishes, the `logging_config` fixture resumes after the `yield` statement. (This fixture is a generator function, and the `next()` function is used to request a second value from it.) This closes and removes the handler, cleanly breaking the network connection with the log catcher process.

When testing finishes overall, the `log_catcher` fixture can then terminate the child process. To help with debugging, we print any output. To be sure the test worked, we check the OS return code. Because the process was terminated (via `p.terminate()`), the return code should be the `signal.SIGTERM` value. Other return code values, particularly a return code of 1, mean that the log catcher crashed and the test failed.

We've omitted a detailed **THEN** check, but it could also be part of the `log_catcher` fixture. The existing `assert` statement makes sure that the log catcher terminated with the expected return

code. Once the catcher in the sky has finished absorbing log messages, this fixture should also read the log file to be sure it contains the expected entries for the two scenarios.

Fixtures can also be parameterized. We can use a decorator such as `@pytest.fixture(params=[some, list, of, values])` to create multiple copies of a fixture, which will lead to multiple tests with each of the given parameter values.

The sophistication of `pytest` fixtures makes them very handy for a wide variety of test setup and teardown requirements. Earlier in this section, we hinted at ways to mark tests as inappropriate for a particular version of Python. In the next section, we'll look at how we can mark tests to be skipped.

Skipping tests with pytest

It is sometimes necessary to skip tests. There can be a similar variety of reasons: the code being tested hasn't been written yet, the test only runs on certain interpreters or OSs, or the test is time-consuming and should only be run when specifically requested. In the previous section, one of our tests would not work in Python 3.8 and needed to be skipped.

One way to skip tests is by using the `pytest.skip()` function. It accepts a single argument: a string describing why it has been skipped. This function can be called anywhere. If we call it inside a test function, the test will be skipped. If we call it at the module level, all the tests in that module will be skipped. If we call it inside a fixture, all tests that reference the fixture will be skipped.

Of course, in all these locations, it is often only desirable to skip tests if certain conditions have or have not been met. Since we can execute the `skip()` function at any place in Python code, we can execute it inside an `if` statement. We may write a test that looks as follows:

```
import sys
import pytest


def test_simple_skip() -> None:
    if sys.platform != "ios":
        pytest.skip("Test works only on Pythonista for ios")

    import location  # type: ignore [import]

    img = location.render_map_snapshot(36.8508, -76.2859)
    assert img is not None
```

This test will skip on most OSs. It should run on the Pythonista port of Python for iOS. It shows how we can skip a scenario conditionally, and since the `if` statement can check any valid conditional, we have a lot of power over when tests are skipped. Often, we check `sys.version_info` to check the Python interpreter version, `sys.platform` to check the OS, or `some_library.__version__` to check whether we have a recent-enough version of a given module.

Since skipping an individual test method or function based on a condition is one of the most common uses of test skipping, `pytest` provides a convenient decorator that allows us to do this in one line. The decorator accepts a single string, which can contain any executable Python code that evaluates to a Boolean value. For example, the following test will only run on Python 3.9 or higher:

```
import sys
import pytest


@pytest.mark.skipif(
    sys.version_info < (3, 9), reason="requires 3.9, Path.removeprefix()"
)
def test_feature_python39() -> None:
    file_name = "(old) myfile.dat"
    assert file_name.removeprefix("(old) ") == "myfile.dat"
```

The `pytest.mark.xfail` decorator marks a test as expected to fail. If the test is successful, it will be recorded as a failure (it failed to fail!). If it fails, it will be reported as expected behavior. In the case of `xfail`, the conditional argument is optional. If it is not supplied, the test will be marked as expected to fail under all conditions.

The `pytest` framework has a ton of other features besides those described here, and the developers are constantly adding innovative new ways to make your testing experience more enjoyable. They have thorough documentation on their website at `https://docs.pytest.org/`.

The `pytest` tool can find and run tests defined using the standard `unittest` library, in addition to its own testing infrastructure. This means that if you want to migrate from `unittest` to `pytest`, you don't have to rewrite all your old tests.

We've looked at using a fixture to set up and tear down a complex environment for testing. This is helpful for some integration tests, but a better approach may be to imitate an expensive object or a risky operation. Additionally, any kind of teardown operation is inappropriate for unit tests. A unit test isolates each software component as a separate unit to be tested. This means we'll often

replace all of the interface objects with imitations, called "mocks," to isolate the unit being tested. Next, we'll turn to creating mock objects to isolate units and imitate expensive resources.

Imitating objects using mocks

Isolated problems are easier to diagnose and solve. Figuring out why a gasoline car won't start can be tricky because there are so many interrelated parts. If a test fails, uncovering all the interrelationships among software components makes diagnosis of the problem difficult. We often want to isolate items by providing simplified imitations. It turns out there are two reasons to replace perfectly good code with imitation (or "mock") objects:

- The most common case is to isolate a unit under test. We want to create collaborating classes and functions so we can test one unknown component in an environment of known, trusted test fixtures.
- Sometimes, we want to test code that requires an object that is either expensive or risky to use. Things such as shared databases, filesystems, and cloud infrastructures can be very expensive to set up and tear down for testing.

In some cases, this may lead to designing an API to have a testable interface. Designing for testability often means designing a more usable interface, too. In particular, we have to expose assumptions about collaborating classes so we can inject a mock object instead of an instance of an actual application class.

For example, imagine we have some code that keeps track of flight statuses in an external key-value store (such as `redis` or `memcached`). We store the timestamp and the most recent status. The implementation will require the `redis` service in order to work. However, this is not needed to write unit tests. The unit tests don't even need the `redis` client library. A mock can be used instead.

To run integration tests, the `redis` client library can be installed. Using a virtual environment such as `venv`, this can be installed with the `python -m pip install redis` command.

Using a virtual environment manager such as `uv` or `poetry` means adding the dependency to the project. The command for `uv` is `uv add redis`.

If you want to run this with a real `redis` server, you'll also need to download and install `redis`. This can be done as follows:

1. Download Docker Desktop to help manage this application. See `https://www.docker.com/products/docker-desktop`.

2. Use the `docker pull redis` command from a Terminal window to download a `redis` server image. This image can be used to build a running Docker container.
3. You can then start the server with `docker run -p 6379:6379 redis`. This will start a container running the `redis` image. Then you can use this for integration testing.

An alternative that avoids **Docker** involves a number of platform-specific steps. See `https://redislabs.com/ebook/appendix-a/` for a number of installation scenarios. The examples that follow will assume **Docker** is being used; the minor changes that are required to switch to a native installation of `redis` are left as an exercise for the reader.

Here's some code that saves the status in a `redis` cache server:

```
import datetime
from enum import Enum
from typing import cast
import redis


class Status(str, Enum):
    CANCELLED = "CANCELLED"
    DELAYED = "DELAYED"
    ON_TIME = "ON TIME"


class FlightStatusTracker:
    def __init__(self) -> None:
        self.redis = redis.Redis(host="127.0.0.1", port=6379, db=0)

    def change_status(self, flight: str, status: Status) -> None:
        if not isinstance(status, Status):
            raise ValueError(f"{status!r} is not a valid Status")
        key = f"flightno:{flight}"
        now = datetime.datetime.now(tz=datetime.timezone.utc)
        value = f"{now.isoformat()}|{status.value}"
        self.redis.set(key, value)

    def get_status(
        self, flight: str
    ) -> tuple[datetime.datetime | None, Status | None]:
        key = f"flightno:{flight}"
        value = cast(str, self.redis.get(key))
        if value:
```

```
            text_timestamp, text_status = value.split("|")
            timestamp = datetime.datetime.fromisoformat(text_timestamp)
            status = Status(text_status)
            return timestamp, status
        return None, None
```

The `Status` class defines an enumeration of four string values. We've provided symbolic names such as `Status.CANCELLED` so that we can have a finite, bounded domain of valid status values. The actual values stored in the database will be strings such as `"CANCELLED"` that — for now — happen to match the symbols we'll be using in the application. In the future, the domain of values may expand or change, but we'd like to keep our application's symbolic names separate from the strings that appear in the database. It's common to use numeric codes with `Enum`, but they can be difficult to remember.

There are a lot of things we ought to test for in the `change_status()` method. We check to be sure that the `status` argument value really is a valid instance of the `Status` enumeration, but we could do more. We should check that it raises the appropriate error if the `flight` argument value isn't sensible. More importantly, we need a test to prove that the key and value have the correct formatting when the `set()` method is called on the `redis` object.

One thing we don't have to check in our unit tests, however, is that the `redis` object is storing the data properly. This is something that absolutely should be tested in integration or application testing. At the unit test level, we need to assume that the `py-redis` developers have tested their code and that this method does what we want it to. As a rule, unit tests should be self-contained; the unit under test should be isolated from outside resources, such as a running Redis instance.

Instead of integrating with a Redis server, we only need to test that the `set()` method was called the appropriate number of times and with the appropriate arguments. We can use `Mock()` objects in our tests to replace the troublesome method with an object we can introspect. The following example illustrates the use of `Mock`:

```
import datetime
from unittest.mock import Mock, patch, call
import pytest

import flight_status_redis
```

```
@pytest.fixture
def mock_redis() -> Mock:
    mock_redis_instance = Mock(set=Mock(return_value=True))
    return mock_redis_instance


@pytest.fixture
def tracker(
        monkeypatch: pytest.MonkeyPatch,
        mock_redis: Mock
) -> flight_status_redis.FlightStatusTracker:
    """Depending on the test scenario, this may require a running REDIS
    server."""
    fst = flight_status_redis.FlightStatusTracker()
    monkeypatch.setattr(fst, "redis", mock_redis)
    return fst
```

```
def test_monkeypatch_class(
        tracker: flight_status_redis.FlightStatusTracker,
        mock_redis: Mock
) -> None:
    # with patch.object(tracker, "redis", mock_redis):
    with pytest.raises(ValueError) as ex:
        tracker.change_status("AC101", "lost")  # type: ignore [arg-type]
    assert ex.value.args[0] == "'lost' is not a valid Status"
    assert mock_redis.set.call_count == 0
```

This test uses the `pytest.raises` context manager to make sure the correct exception is raised when an inappropriate argument is passed in. In addition, it creates a `Mock` object for the `redis` instance that `FlightStatusTracker` will use.

The `Mock` object contains an attribute, `set`, which is a mock method that will always return `True`. The test, however, makes sure that the `redis.set()` method is never called. If it is, it means there is a bug in our exception-handling code.

Note the navigation into the mock object. We use `mock_redis.set` to examine the mocked `set()` method of a `Mock` object created by the `mock_redis` fixture. `call_count` is an attribute that all `Mock` objects maintain.

While we can use code such as `flt.redis = mock_redis` to replace a real object with a `Mock` object during a test, there is potential for problems. Simply replacing a value or even replacing a class method can only work for objects that are destroyed and created for each test function. If we need to patch items at the module level, the module isn't going to be reimported. A much more general solution is to use a patcher to inject a `Mock` object temporarily. In this example, we used the `monkeypatch` fixture of `pytest` to make a temporary change to the `FlightStatusTracker` object. A `monkeypatch` has its own automatic teardown at the end of a test, allowing us to use monkeypatched modules and classes without breaking other tests.

This test case will be flagged by **mypy**. The **mypy** tool will object to using a string argument value for the status parameter of the `change_status()` function; this clearly must be an instance of the `Status` enumeration. A special comment can be added to silence the **mypy** argument type check, `# type: ignore [arg-type]`.

Additional patching techniques

In some cases, we only need to inject a special function or method for the duration of a single test. We may not really be creating a sophisticated `Mock` object that's used in multiple tests. In the case of an isolated mock, we may not need to use all the features of the `monkeypatch` fixture, either. For example, if we want to test the timestamp formatting in the `Mock` method, we need to know exactly what `datetime.datetime.now()` is going to return. However, this value changes from run to run. We need some way to pin it to a specific datetime value so we can test it deterministically.

Temporarily setting a library function to a specific value is one place where patching is essential. In addition to the `monkeypatch` fixture, the `unittest.mock` library provides a `patch` context manager. This context manager allows us to replace attributes on existing libraries with mock objects. When the context manager exits, the original attribute is automatically restored so as not to impact other test cases. Here's an example:

```
def test_patch_class(
        tracker: flight_status_redis.FlightStatusTracker,
        mock_redis: Mock
) -> None:
    fake_now = datetime.datetime(2020, 10, 26, 23, 24, 25)
    utc = datetime.timezone.utc
    # with patch.object(tracker, "redis", mock_redis):
    with patch("flight_status_redis.datetime") as mock_datetime:
```

```
        mock_datetime.datetime = Mock(now=Mock(return_value=fake_now))
        mock_datetime.timezone = Mock(utc=utc)
        tracker.change_status("AC101", flight_status_redis.Status.ON_TIME)
    mock_datetime.datetime.now.assert_called_once_with(tz=utc)
    expected = "2020-10-26T23:24:25|ON TIME"
    mock_redis.set.assert_called_once_with("flightno:AC101", expected)
```

We don't want our test results to depend on the computer's clock, so we built the `fake_now` object with a specific date and time we can expect to see in our test results. This kind of replacement is very common in unit tests.

The `patch()` context manager returns a `Mock` object that was used to replace some other object. In this case, the object being replaced is the entire `datetime` module inside the `flight_status_redis` module. When we assigned `mock_datetime.datetime`, we replaced the `datetime` class inside the mocked `datetime` module with our own `Mock` object; this new `Mock` defines one attribute, `now`. Because the `utcnow` attribute is a `Mock` object that returns a value, it behaves like a method and returns a fixed, known value, `fake_now`. When the interpreter exits the `patch` context manager, the original `datetime` functionality is restored.

After calling our `change_status()` method with known values, we use the `assert_called_once_with()` method of the `Mock` object to ensure that the `now()` function was indeed called exactly once with the expected arguments (no arguments, in this case). We also use the `assert_called_once_with()` method on the `Mock` `redis.set` method to make sure it was called with arguments that were formatted as we expected them to be. In addition to the "called once with" methods, we can also check the exact list of mock calls that were made. This sequence is available in the `mock_calls` attribute of a `Mock` object.

Mocking dates so you can have deterministic test results is a common patching scenario. The technique applies to any stateful object but is particularly important for external resources (such as the clock) that exist outside our application.

For the special case of `datetime` and `time`, packages such as `freezegun` can simplify the monkey-patching required so that a known, fixed date is available.

The patches we made in this example are intentionally sweeping. We replaced the entire `datetime` module with a `Mock` object. This will tend to expose unexpected uses of datetime features; if any method not specifically mocked (like the `now()` method was mocked) gets used, it will return `Mock` objects that are likely to crash code under test.

The previous example also shows how testability needs to guide our API design. The `tracker` fixture has an interesting problem: it creates a `FlightStatusTracker` object, which constructs a Redis connection. After the Redis connection is built, we replace it. When we run tests for this code, however, we will discover that each test will create an unused Redis connection. Some tests may fail if there is no Redis server running. Because this test requires external resources, it's not a proper unit test. There are two possible layers of failure: the code doesn't work or the unit tests don't work because of some hidden external dependency. This can become a nightmare to sort out.

We could solve this problem by mocking the `redis.Redis` class. A `Mock` for this class can return a mock instance in a `setUp` method. A better idea, however, might be to rethink our implementation more fundamentally. Instead of constructing the `redis` instance inside `__init__`, we should allow the collaborating object to pass one in. This reduction in the coupling between the `FlightStatusTracker` class and the persistence mechanism permits easier refactoring as well as better testing. Here's an example:

```
class FlightStatusTracker_Alt:
    def __init__(self, redis_instance: redis.Connection | None = None) ->
    None:
        self.redis = (
            redis_instance
            if redis_instance
            else redis.Redis(host="127.0.0.1", port=6379, db=0)
        )
```

This allows us to pass a connection in when we are testing. The idea for this follows from the Interface Segregation principle: we don't want our status tracker to also have **Redis** initialization buried inside it. This changed design allows any client code that talks to `FlightStatusTracker` to pass in its own `redis` instance. This permits using the Decoration design pattern to create a wrapper around **Redis** to do additional application-specific processing such as auditing or logging. There are a variety of reasons why they might want to do this: they may have already constructed one for other parts of their code; they may have created an optimized implementation of the `redis` API; or perhaps they have one that logs metrics to their internal monitoring systems. By writing a unit test, we've uncovered a use case that makes our API more flexible from the start, rather than waiting for clients to demand that we support their exotic needs.

This has been a brief introduction to the wonders of mocking code. Mock objects have been part of the standard `unittest` library since Python 3.3. As you can see from these examples, they can

also be used with `pytest` and other test frameworks. Mock objects have other, more advanced features that you may need to take advantage of as your code becomes more complicated. For example, you can use the `spec` argument to invite a mock to imitate an existing class, so that it raises an error if code tries to access an attribute that does not exist on the imitated class. You can also construct mock methods that return different arguments each time they are called by passing a list as the `side_effect` argument. The `side_effect` parameter is quite versatile; you can also use it to execute arbitrary functions when the mock is called or to raise an exception.

The point of unit testing is to be sure that each "unit" works in isolation. Often, a unit is an individual class, and we'll need to mock the collaborators. In some cases, there's a composition of classes or a Façade for which a number of application classes can be tested together as a "unit." There's a clear boundary, however. If we need to look inside some external module or class (one we didn't write) to see how to mock its dependencies, we've taken a step too far. Needing to mock internal states of objects we didn't write means we've broken an encapsulation boundary.

Don't examine the implementation details of classes outside your application to see how to mock their collaborators; instead, mock the entire class you depend on.

This often leads to providing a mock for an entire database or external API.

We can extend this idea of imitating objects one step further. There's a specialized fixture we use when we want to ensure data has been left untouched. We'll look at this next.

The sentinel object

In many designs, we'll have a class with attribute values that will be provided as parameters to other objects, without really doing any processing on those objects. For example, we may provide a `Path` object to a class, and the class then provides this `Path` object to an OS function; the class we designed doesn't do any processing to change the internal state of the the `Path` object. From a unit testing perspective, the object is "opaque" to the class we're testing — the class we're writing doesn't look inside the object at state or methods.

The `unittest.mock` module provides a handy object, `sentinel`, that can be used to create opaque objects that we can use in test cases to be sure that the application stored and forwarded some object without any meddling.

Here's a class, `FileChecksum`, that saves an object computed by the `sha256()` function of the `hashlib` module:

```
class FileChecksum:
    def __init__(self, source: Path) -> None:
        self.source = source
        self.checksum = hashlib.sha256(source.read_bytes())
```

We can isolate this code from the other modules for unit testing purposes. We'll create a `Mock` for the `hashlib` module, and we'll use a `sentinel` for the result:

```
from unittest.mock import Mock, sentinel
from typing import Any


@pytest.fixture
def mock_hashlib(monkeypatch: Any) -> Mock:
    mocked_hashlib = Mock(sha256=Mock(return_value=sentinel.checksum))
    monkeypatch.setattr(checksum_writer, "hashlib", mocked_hashlib)
    return mocked_hashlib


def test_file_checksum(mock_hashlib: Mock, tmp_path: Any) -> None:
    source_file = tmp_path / "some_file"
    source_file.write_text("")
    cw = checksum_writer.FileChecksum(source_file)
    assert cw.source == source_file
    assert cw.checksum == sentinel.checksum
```

Our `mocked_hashlib` object provides a method, `sha256`, that returns a unique `sentinel.CHECKSUM` object. This object, created by the `sentinel` object, has very few methods or attributes. These resulting objects are designed for equality checks and nothing else. A `sentinel` in a test case is a way to be sure the `FileChecksum` class doesn't do anything wrong or unexpected with the checksum objects it will be given.

The test case creates a `FileChecksum` object. The test confirms that the file was the provided argument value, `source_file`. The test also confirms that the checksum matched the original `sentinel` object. This confirms that the `FileChecksum` instance stored the checksum results properly and presented the result as the value of the `checksum` attribute.

If we change the implementation of the `FileChecksum` class to — for example — use properties instead of direct access to the attribute, the test will confirm the checksum was treated as an opaque object that came from the `hashlib.sha256()` function and was not processed in any other way.

We've looked at two unit testing frameworks: the built-in `unittest` package and the external `pytest` package. They both provide ways for us to write clear, simple tests that can confirm that our application works. It's important to have a clear objective defining the required amount of testing. Python has an easy-to-use **coverage** package that gives us one objective measure of test quality.

How much testing is enough?

We've already established that untested code is broken code. But how can we tell how well our code is tested? How do we know how much of our code is actually being tested and how much is broken? The first question is the more important one, but it's hard to answer. Even if we know we have tested every line of code in our application, we do not know that we have tested it completely. For example, if we write a `stats` test that only checks what happens when we provide a list of integers, it may still fail spectacularly if used on a list of floats, strings, or self-made objects. The onus of designing complete test suites still lies with the programmer.

The second question — how much of our code is actually being tested — is easy to verify. **Code coverage** is a count of the number of lines of code that are executed by a program. From the number of lines that are in the program as a whole, we know what percentage of the code was really tested or covered. If we additionally have an indicator that tells us which lines were not tested, we can more easily write new tests to ensure that those lines are less likely to harbor problems.

The most popular tool for testing code coverage is called, memorably enough, `coverage.py`. It can be installed like most other third-party libraries, using the `python -m pip install coverage` command, or a command such as `uv add coverage –dev` when using an environment manager such as **uv**.

We don't have space to cover all the details of the coverage API, so we'll just look at a few typical examples. If we have a Python script that runs all our unit tests for us (this could be using `unittest.main`, `unittest discover`, or `pytest`), we can use the following command to perform coverage analysis for a specific unit test file:

```
% export PYTHONPATH="$(pwd)/src:$PYTHONPATH"
% coverage run -m pytest tests/test_coverage.py
```

This command will create a file named `.coverage`, which holds the data from the run.

Windows PowerShell users can do the following:

```
> $ENV:PYTHONPATH = "$pwd\src" + ";" + $PYTHONPATH
> coverage run -m pytest tests/test_coverage.py
```

We can now use the `coverage report` command to get an analysis of the code coverage:

```
% coverage report
```

The resulting output should be as follows:

```
Name                     Stmts   Miss  Cover
------------------------------------------
src/stats.py                19     11    42%
tests/test_coverage.py       7      0   100%
------------------------------------------
TOTAL                       26     11    58%
```

This report lists the files that were executed (our unit test and the module it imported), the number of lines of code in each file, and the number of lines of code that were executed by the test. The two numbers are then combined to show the amount of code coverage. Not surprisingly, the entire test was executed, but only a fraction of the `stats` module was exercised.

If we pass the `-m` option to the `report` command, it will add a column that identifies the lines that are missing from the test execution. The output looks as follows:

```
Name                     Stmts   Miss  Cover   Missing
-----------------------------------------------------
src/stats.py                19     11    42%   18-23, 26-31
tests/test_coverage.py       7      0   100%
-----------------------------------------------------
TOTAL                       26     11    58%
```

The ranges of lines listed here identify the lines in the `stats` module that were not executed during the test run.

The example code uses the same `stats` module we created earlier in this chapter. However, it deliberately uses a single test that fails to test a lot of code in the file. Here's the test:

```
import pytest
from stats import StatsList


@pytest.fixture
def valid_stats() -> StatsList:
    return StatsList([1, 2, 2, 3, 3, 4])


def test_mean(valid_stats: StatsList) -> None:
    assert valid_stats.mean() == 2.5
```

This test doesn't test the median or mode functions, which correspond to the line numbers that the coverage output told us were missing.

It can be helpful to include more than the `src` directory tree in coverage analysis. A large project may have a complex tests directory, including additional tools and supporting libraries. As the project evolves, there may be some test or support code that's obsolete but hasn't been cleaned up yet.

Unfortunately, if we could somehow run a coverage report on this section of the chapter, we'd find that we have not covered most of what there is to know about code coverage! It is possible to use the coverage API to manage code coverage from within our own programs (or test suites). Further, `coverage.py` accepts numerous configuration options that we haven't touched on. We also haven't discussed the difference between statement coverage and branch coverage (the latter is much more useful and is the default in recent versions of `coverage.py`), or other styles of code coverage.

Bear in mind that while 100 percent code coverage is a goal that we should all strive for, 100 percent coverage is not enough! Just because a statement was tested, does not mean that it was tested properly for all possible inputs. The boundary value analysis technique includes looking at five values to bracket the edge cases: a value below the minimum, the minimum, in the middle somewhere, the maximum, and a value above the maximum. For non-numeric types, there may not be a tidy range, but the advice can be adapted to other data structures. For lists and mappings, for example, this advice often suggests testing with empty lists or mapping with unexpected keys.

Looking at other tools, the **Hypothesis** package (`https://pypi.org/project/hypothesis/`) can help with more sophisticated test cases. This is particularly helpful when working with more complicated scientific or statistical computations.

It's difficult to emphasize how important testing is. The test-driven development approach encourages us to describe our software via visible, testable objectives. We have to decompose complex problems into discrete, testable solutions. It's not uncommon to have more lines of test code than actual application code. A short but confusing algorithm is sometimes best explained through examples, and each example should be a test case.

Testing and development

One of the many ways unit tests can help is when debugging application problems. When each unit seems to work in isolation, any remaining problems will often be the result of an improperly used interface between components. When searching for the root cause of a problem, a suite of passing tests acts as a set of signposts, directing the developer into the wilderness of untested features in the borderlands between components.

When a problem is found, the cause is often one of the following:

- Someone writing a new class failed to understand an interface with an existing class and used it incorrectly. This indicates a need for a new unit test to reflect the right way to use the interface. This new test should cause the new code to fail its expanded test suite. An integration test is also helpful but not as important as the new unit test focused on interface details.
- The interface was not spelled out in enough detail, and both parties using the interface need to reach an agreement on how the interface should be used. In this case, both sides of the interface will need additional unit tests to show what the interface should be. Both classes should fail these new unit tests; they can then be fixed. Additionally, an integration test can be used to confirm that the two classes agree.

The idea here is to use test cases to drive the development process. A "bug" or an "incident" needs to be translated into a test case that fails. Once we have a concrete expression of a problem in the form of a test case, we can create or revise software until all the tests pass.

If bugs do occur, we'll often follow a test-driven plan, as follows:

1. Write a test (or multiple tests) that duplicates or proves the bug in question is occurring. This test will, of course, fail. In more complex applications, it may be difficult to find the exact steps to recreate a bug in an isolated unit of code; finding this is valuable work, since it requires knowledge of the software and captures the knowledge as a test scenario.

2. Then, write the code to make the tests stop failing. If the tests were comprehensive, the bug will be fixed, and we will know that we didn't break something new while attempting to fix something.

Another benefit of test-driven development is the value of the test cases for further enhancement. Once the tests have been written to identify the problem, we can improve our code as much as we like and be confident that our changes won't break anything we have been testing for. Furthermore, we know exactly when our refactor is finished: when the tests all pass.

Of course, our tests may not comprehensively test everything we need them to; maintenance or code refactoring can still cause undiagnosed bugs that don't show up in testing. Automated tests are not foolproof. As E. W. Dijkstra said, "Program testing can be used to show the presence of bugs, but never to show their absence!" We need to have good reasons why our algorithm is correct, as well as test cases to show that it doesn't have any problems.

Recall

In this chapter, we've looked at a number of topics related to testing applications written in Python. These topics include the following:

- We described the importance of unit testing and test-driven development as a way to be sure that our software does what is expected.
- We started by using the `unittest` module because it's part of the standard library and readily available. It seems a little wordy but otherwise works well for confirming that our software works.
- The `pytest` tool requires a separate installation, but it seems to produce tests that are slightly simpler than those written with the `unittest` module. More importantly, the sophistication of the fixture concept lets us create tests for a wide variety of scenarios.
- The `mock` module, part of the `unittest` package, lets us create mock objects to better isolate the unit of code being tested. By isolating each piece of code, we can narrow our focus to being sure that it works and has the right interface. This makes it easier to combine components.
- Code coverage is a helpful metric to ensure that our testing is adequate. Simply adhering to a numeric goal is no substitute for thinking, but it can help to confirm that efforts were made to be thorough and careful when creating test scenarios.

We've been looking at several kinds of tests with a variety of tools:

- Unit tests with the `unittest` package or the `pytest` package, often using `Mock` objects to isolate the fixture or unit being tested.
- Integration tests, also with `unittest` and `pytest`, where more complete integrated collections of components are tested.
- Static analysis can use **mypy** to examine the data types to be sure they're used properly. This is a kind of test to ensure that the software is acceptable. There are other kinds of static tests, and tools such as `flake8`, `pylint`, and `pyflakes` can be used for these additional analyses.

Some research will turn up scores of additional types of tests. Each distinct type of test has a distinct objective or approach to confirming that the software works. A performance test, for example, seeks to establish that the software is fast enough and uses an acceptable number of resources.

We can't emphasize enough how important testing is. Without automated tests, software can't be considered complete, or even usable. Starting from test cases lets us define the expected behavior in a way that's specific, measurable, achievable, results-based, and trackable: SMART.

Exercises

Practice test-driven development. That is your first exercise. It's easier to do this if you're starting a new project, but if you have existing code that you need to work on, you can start by writing tests for each new feature you implement. This can become frustrating as you become more enamored with automated tests. The old, untested code will start to feel rigid and tightly coupled, and will become uncomfortable to maintain; you'll start feeling like changes you make are breaking the code and you have no way of knowing, for lack of tests. But if you start small, adding tests to the code base improves it over time. It's not unusual for there to be more test code than application code!

So, to get your feet wet with test-driven development, start a fresh project. Once you've started to appreciate the benefits (you will) and realize that the time spent writing tests is quickly regained in terms of more maintainable code, you'll want to start writing tests for existing code. This is when you should start doing it, not before. Writing tests for code that we *know* works is boring. It is hard to get interested in the project until we realize just how broken the code we thought was working really is.

Try writing the same set of tests using both the built-in `unittest` module and `pytest`. Which do you prefer? `unittest` is more similar to test frameworks in other languages, while `pytest` is arguably

more Pythonic. Both allow us to write object-oriented tests and test object-oriented programs with ease.

Try running a coverage report on the tests you've written. Did you miss testing any lines of code? Even if you have 100 percent coverage, have you tested all the possible inputs? If you're doing test-driven development, 100 percent coverage should follow quite naturally, as you will write a test before the code that satisfies that test. However, if you're writing tests for existing code, it is more likely that there will be edge conditions that go untested.

When creating test cases, it can help to think carefully about the values that are somehow different, such as the following, for example:

- Empty lists when you expect full ones
- Negative numbers, zero, one, or infinity compared to positive integers
- Floats that don't round to an exact decimal place
- Strings when you expected numerals
- Unicode strings when you expected ASCII
- The ubiquitous `None` value when you expected something meaningful

If your tests cover such edge cases, your code will be in good shape.

Summary

We have finally covered the most important topic in Python programming: automated testing. Test-driven development is considered a best practice. The standard library `unittest` module provides a great out-of-the-box solution for testing, while the `pytest` framework has some more Pythonic syntax. Mocks can be used to emulate complex classes in our tests. Code coverage gives us an estimate of how much of our code is being run by our tests, but it does not tell us that we have tested the right things.

In the next chapter, we'll jump into a completely different topic: concurrency.

Join our community Discord space

Join our Python Discord workspace to discuss and know more about the book: `https://packt.link/dHrHU`

14

Concurrency

Concurrency is the art of making a computer do (or appear to do) multiple things at once. Historically, this meant inviting the processor to switch between different tasks many times per second. In modern systems, it can also mean doing two or more things simultaneously on separate processor cores.

Concurrency is not inherently an object-oriented topic, but Python's concurrent systems provide object-oriented interfaces. This chapter will introduce you to the following topics:

- Threads
- Multiprocessing
- Futures
- AsyncIO
- The dining philosophers benchmark

Concurrent processes can become complicated. The basic concepts are fairly simple, but the bugs that can occur are notoriously difficult to track down when the sequence of state changes is unpredictable. However, for many projects, concurrency is the only way to get the performance we need. Imagine if a web server couldn't respond to a user's request until another user's request had

been completed! We'll see how to implement concurrency in Python, and some common pitfalls to avoid.

The Python language explicitly executes statements in order. To consider concurrent execution of statements, we'll need to take a step away from Python.

Background on concurrent processing

Conceptually, it can help to think of concurrent processing by imagining a group of people who can't see each other and are trying to collaborate on a task. Perhaps their vision is impaired or blocked by screens, or their workspace has awkward doorways that they can't quite see through. These people can, however, pass tokens, notes, and work-in-progress to each other.

Imagine a small delicatessen in an old seaside resort city (on the Atlantic coast of the US) with an awkward counter-top layout. The awkward layout of the old building means that the two sandwich chefs can't see or hear each other. While the owner can afford to pay two fine chefs, the owner can't afford more than one serving tray. Due to the awkward complications of the beach boardwalk, the chefs can't even see the tray, either. They're forced to reach down below their counter to be sure the serving tray is in place. Then, assured the tray is there, they carefully place their work of art — complete with pickles and a few of the house-made potato chips — onto the tray. (Yes, they can't see the tray, but they're spectacular chefs who can place a sandwich, pickles, and chips flawlessly.)

The owner, however, can see the chefs. Indeed, passers-by can watch the chefs work. It's a great show. The owner typically deals the order tickets out to each chef in strict alternation. And ordinarily, the one and only serving tray can be placed so the sandwich arrives and is presented at the table with a flourish. The chefs, as we said, have to wait to feel the tray before their next creation warms someone's palate.

Then, one day, one of the chefs (we'll call him Michael, but his friends call him Mo) is nearly done with an order, but has to run to the cooler for more of those dill pickles everyone loves. This delays Mo's prep time, and the owner sees that the other chef, Constantine, looks like he'll finish just a fraction of a second before Mo. Even though Mo has returned with the pickles and is ready with the sandwich, the owner does something embarrassing. Remember the rule: check first, then place the sandwich. Everyone in the shop knows this. The owner moves the tray from the opening below Mo's station to the opening below Constantine's. Mo places his creation — what would have been a delightful Reuben sandwich with extra sauerkraut — into the empty space where a tray should have been, where it splashes onto the delicatessen floor, a horrifying waste of rye bread and corned beef.

How could the foolproof method of checking for the tray then depositing the sandwich have failed to work? It had survived the test of many busy lunch hours, and yet a small disruption in the regular sequence of events leads to a mess. The separation in time between testing for the tray and depositing the sandwich turned into an opportunity for the owner to make a state change.

There's a race between the owner and chefs. Preventing unexpected state changes is the essential design problem for concurrent programming.

One solution could be to use a semaphore — a flag — to prevent unexpected changes to the tray. This is a kind of shared lock. Each chef is forced to seize the flag before plating, and once they have the flag, they can be confident that the owner won't move the tray. Similarly, the owner must seize the flag before moving the tray. And, each participant must return the flag to the little flag-stand when they've finished plating or serving.

Concurrent work requires some method for synchronizing access to shared resources. One essential power of large, modern computers is managing concurrency through operating system features, collectively called the **kernel**.

Older and smaller computers, with a single core in a single CPU, had to interleave everything. The clever coordination made things appear to be working at the same time. Newer multi-core computers (and large multi-processor computers) can actually perform operations concurrently, making the scheduling of work a bit more involved.

We have several ways to achieve concurrent processing:

- The operating system lets us run more than one program at a time. The Python `subprocess` module gives us ready access to these capabilities. The `multiprocessing` module also provides a number of convenient ways to handle multiple processes. This is relatively easy to start, but each process is carefully sequestered from all other processes. How can they share data?
- Some clever software libraries allow a single program to have multiple concurrent threads of operation within a single process. The Python `threading` module gives us access to handle multiple threads. This is more complex to get started, and each thread has complete access to the data in all other threads. How can we coordinate updates to shared data structures?

Additionally, `concurrent.futures` and `asyncio` provide easier-to-use wrappers around the underlying libraries. We'll start this chapter by looking at Python's use of the `threading` library to allow many things to happen concurrently in a single operating system process. This is simple but has

some challenges when working with shared data structures.

Threads

A thread is a sequence of Python byte-code instructions that may be interrupted and resumed. The idea is for a process to execute via separate, concurrent threads. This will allow computation to proceed while the program is waiting for I/O to happen.

For example, a server can start processing a new network request while it waits for data from a previous request to arrive. Or an interactive program might render an animation or perform a calculation while waiting for the user to press a key. Bear in mind that while a person can type more than 500 characters per minute, a computer can perform billions of instructions per second. Thus, a ton of processing can happen between individual key presses, even when typing quickly.

It's theoretically possible to manage all of this switching between activities within your program, but it would be virtually impossible to get right. Instead, we can rely on Python and the operating system to take care of the tricky switching part, while we create objects that appear to be running independently but simultaneously. These objects are called **threads**.

Let's take a look at a basic example. We've decomposed the chef's actions into getting the next order and then preparing the order. Right now, the work involved in getting the next order is minimal, but it's also something that can change in other implementations; this leaves a one-line function, which seems like a lot of overhead for such a small thing. It makes the small thing (getting an order) visible and changeable, which outweighs the overheads. We'll start with an essential definition of the thread's processing, as shown in the following class:

```
class Chef(Thread):
    def __init__(self, name: str) -> None:
        super().__init__(name=name)
        self.total = 0

    def get_order(self) -> None:
        self.order = THE_ORDERS.pop(0)

    def prepare(self) -> None:
        """Simulate doing a lot of work with a BIG computation"""
        start = time.monotonic()
        target = start + 1 + random.random()
        for i in range(1_000_000_000):
```

```
            self.total += math.factorial(i)
            if time.monotonic() >= target:
                break
        print(f"{time.monotonic():.3f} {self.name} made {self.order}")

    def run(self) -> None:
        while True:
            try:
                self.get_order()
                self.prepare()
            except IndexError:
                break  # No more orders
```

Note that the `prepare()` method does a great deal of computation until it reaches some given time. The idea is to occupy the CPU (or at least one core) as fully as possible. In some examples, folks will use the `sleep()` function, but this gives up the CPU so other processes can use it. We want to see the CPU spike during preparation.

A thread in our running application must extend the `Thread` class and implement the `run` method. Any code executed by the `run` method will be a separate thread of processing, scheduled independently. Our thread is relying on a global variable, `THE_ORDERS`, which is a shared object. This could be a queue or some other structure that provides a series of values. Here's how this example defines this shared global object:

```
import math
import random
from threading import Thread
import time

THE_ORDERS = [
    "Reuben",
    "Ham and Cheese",
    "Monte Cristo",
    "Tuna Melt",
    "Cuban",
    "Grilled Cheese",
    "French Dip",
    "BLT",
]
```

In this case, we've defined the orders as a simple, fixed list of values. In a larger application, we might be reading these from a socket or a queue object. Here's the top-level program that starts things running:

```
Mo = Chef("Michael")
Constantine = Chef("Constantine")

if __name__ == "__main__":
    random.seed(42)
    Mo.start()
    Constantine.start()
```

This will create two threads. The new threads don't start running until we call the `start()` method on the object. When the two threads have started, they both pop a value from the list of orders and then commence to perform a large computation and — eventually — report their status.

The output looks like this:

```
1.076 Constantine made Ham and Cheese
1.676 Michael made Reuben
2.351 Constantine made Monte Cristo
2.899 Michael made Tuna Melt
4.094 Constantine made Cuban
4.576 Michael made Grilled Cheese
5.664 Michael made BLT
5.987 Constantine made French Dip
```

Note that the sandwiches aren't completed in the exact order that they were presented in the `THE_ORDERS` list. Each chef works at their own (randomized) pace. Changing the random generator seed will change the times and may adjust the order slightly.

Using `seed()` provides repeatable sequences of random numbers. This is helpful when testing software because it provides results that are difficult to predict but perfectly repeatable.

What's important about this example is that the threads are sharing data structures. The concurrency is an illusion created by clever scheduling of the threads to interleave work from the two chef threads.

The only update to a shared data structure in this small example is to pop from a list. If we were to create our own class and implement more complex state changes, we could uncover a number of interesting and confusing issues with using threads.

The many problems with threads

Threads can be useful if appropriate care is taken to manage shared memory, but modern Python programmers tend to avoid them for several reasons. As we'll see, there are other ways to code concurrent programming that are receiving more attention from the Python community. Let's discuss some of the pitfalls before moving on to alternatives to multithreaded applications.

Shared memory

The main problem with threads is also their primary advantage. Threads have access to all the process memory and thus all the objects in memory. A disregard for the shared state can too easily cause inconsistencies.

Have you ever encountered a room where a single light has two switches and two different people turn them on at the same time? Each person — each a separate thread — expects their action to set the the lamp (a variable) state to on, but the resulting value (the lamp) remains off, which is inconsistent with those expectations. Now imagine if those two threads were transferring funds between bank accounts or managing the cruise control for a vehicle.

The solution to this problem in threaded programming is to *synchronize* access to any code that reads or (especially) writes a shared variable. Python's `threading` library offers the `Lock` class, which can be used via the `with` statement to create a context where a single thread has access to update shared objects.

The synchronization solution works in general, but it is way too easy to forget to apply it to shared data in a specific application. Worse, bugs due to inappropriate use of synchronization are really hard to track down because the order in which threads perform operations is inconsistent. We can't easily reproduce the error. Usually, it is safest to force communication between threads to happen using a lightweight data structure that already uses locks appropriately. Python offers the `queue.Queue` class to do this; a number of threads can write to a queue, where a single thread consumes the results. This gives us a tidy, reusable, proven technique for having multiple threads sharing a data structure. The `multiprocessing.Queue` class is nearly identical; we will discuss this in the *Multiprocessing* section of this chapter.

In some cases, these disadvantages might be outweighed by the one advantage of allowing shared memory: it's fast. If multiple threads need access to a huge data structure, shared memory can provide that access quickly. However, this advantage is usually nullified by the fact that, in Python, it is impossible for two threads running on different CPU cores to be performing calculations at exactly the same time. This brings us to our second problem with threads.

The Global Interpreter Lock (GIL)

In order to efficiently manage memory, garbage collection, and calls to machine code in native libraries, Python has a **Global Interpreter Lock**, or **GIL**. Prior to Python 3.13, It was impossible to turn off. The GIL constrains thread schedule by preventing any two threads from doing computations at the exact same time; the work is interleaved artificially. When a thread makes an operating system request — for example, to access the disk or network — the GIL is released as soon as the thread starts waiting for the operating system request to complete.

The GIL makes it very easy to write programs with multi-step state changes to complicated data structures. We can remove an item from a list, knowing all subsequent items have to be shuffled forward in the list, without an explicit lock. It relieves us from the burden of having to sweat the details of each state change.

The GIL is disparaged, mostly by people who don't understand what it is or the benefits it brings to Python. While it can interfere with multithreaded compute-intensive programming, the impact on other kinds of workloads is often minimal. When confronted with a compute-intensive algorithm, it may help to switch to using the `dask` package to manage the processing. See `https://dask.org` for more information on this alternative. The book *Scalable Data Analysis in Python with Dask* can be informative, also.

The GIL can be selectively disabled in IronPython. See *The IronPython Cookbook* for details on how to release the GIL for compute-intensive processing in IronPython.

Python 3.13 introduced code changes that permit a developer to recompile Python without the GIL. See *PEP 703* (`https://peps.python.org/pep-0703/`). Note that — for now — this is experimental and requires building a separate Python with the GIL removed. The "free threading" version of Python requires modules that are designed to support a GIL-free Python. This can mean that your favorite library extension may not actually work properly without the GIL, and will not even load. See *C API Extension Support for Free Threading* (`https://docs.python.org/3/howto/free-threading-extensions.html#freethreading-extensions-howto`) in the Python HOWTO documents.

The notion is that after two or three releases — perhaps as early as Python 3.15 or 3.16 — the GIL would be controlled with a runtime flag instead of a separate build. Removing the GIL requires some profound changes to the CPython implementation. These changes include how reference counting works to locate objects in use and no longer referenced, some low-level details of memory management, thread-safety for Python container objects, as well as how locking and atomic APIs need to work. While we rarely see these things directly, we see them indirectly and we see their effects.

Something that seems as simple as passing a mutable object to a function requires some careful coding in the CPython implementation. Inside the function body, Python tracks two references to the mutable container: the calling function and the called function both have references. When the function body finishes, the reference count is decremented by 1. The object doesn't mysteriously vanish, though, because the reference count is still at least one.

Building a free-threading Python is way outside the scope of this book. We'll turn to looking at some of the overheads involved in programming with threads.

Thread overhead

One additional limitation of threads, as compared to other asynchronous approaches we will be discussing later, is the cost of maintaining each thread. Each thread takes up a certain amount of memory (both in the Python process and the operating system kernel) to record the state of that thread. Switching between the threads also uses a (small) amount of CPU time. This work happens seamlessly without any extra coding (we just have to call `start()` and the rest is taken care of), but the work still has to happen somewhere.

What this means is that simply throwing more threads at a problem has a diminishing return. Doubling the threads also increases the overhead, and performance improvements are always less than double.

These costs can be amortized over a larger workload by reusing threads to perform multiple jobs. Python provides a `ThreadPool` feature to handle this. It behaves identically to `ProcessPool`, which we will discuss shortly, so let's defer that discussion until later in this chapter.

In the next section, we'll look at the principal alternative to multi-threading. The `multiprocessing` module lets us work with operating system-level subprocesses.

Multiprocessing

Threads exist within a single OS process; that's why they can share access to objects. We can do concurrent computing at the process level, too. Unlike threads, separate processes cannot directly access variables set up by other processes. This independence is helpful because each process has its own GIL and its own private pool of resources. On a modern multi-core processor, a process may have its own core, permitting concurrent work with other cores.

The `multiprocessing` API was originally designed to mimic the `threading` API. However, the `multiprocessing` interface has evolved, and in recent versions of Python, it supports more features more robustly. The `multiprocessing` library is designed for when CPU-intensive jobs need to happen in parallel and multiple cores are available. Multiprocessing is not as useful when the processes spend a majority of their time waiting on I/O (for example, network, disk, database, or keyboard), but it is the way to go for parallel computation.

The `multiprocessing` module spins up new operating system processes to do the work. This means there is an entirely separate copy of the Python interpreter running for each process. Let's try to parallelize a compute-heavy operation using similar constructs to those provided by the `threading` API, as follows:

```
from multiprocessing import Process, cpu_count
from threading import Thread
import time
import os


class MuchCPU(Process):
    def run(self) -> None:
        print(f"OS PID {os.getpid()}")

        _ = sum(2 * i + 1 for i in range(100_000_000))
```

```
if __name__ == "__main__":
    workers = [MuchCPU() for f in range(cpu_count())]
```

```
    t = time.perf_counter()
    for p in workers:
        p.start()
    for p in workers:
```

```
        p.join()
    print(f"work took {time.perf_counter() - t:.3f} seconds")
```

This example just ties up the CPU computing the sum of 100 million odd numbers. You may not consider this to be useful work, but it can warm up your laptop on a chilly day!

The API should be familiar; we implement a subclass of `Process` (instead of `Thread`) and implement a `run` method. This method prints out the operating system **process ID** (**PID**), a unique number assigned to each process on the machine, before doing some intense (if misguided) work.

Pay special attention to the `if __name__ == "__main__":` guard around the module-level code. This prevents the application from taking off and running when the module is being imported. This is a good practice in general, but when using the `multiprocessing` module, it is absolutely essential.

Always sequester all top-level processing inside the `if __name__ == "__main__":` guard.

A module used for multiprocessing must avoid any processing outside function and class bodies.

Behind the scenes, the `multiprocessing` module will import our application module inside each of the new processes in order to create the class and execute the `run()` method. If we allowed the entire module to execute at that point, it would start creating new processes recursively until the operating system ran out of resources, crashing our computer.

This demo constructs one process for each processor core on our machine, then starts and joins each of those processes. On a 2020-era MacBook Pro with a 2 GHz Quad-Core Intel Core i5, the output looks as follows:

```
% python src/processes_1.py
OS PID 15492
OS PID 15493
OS PID 15494
OS PID 15495
OS PID 15497
OS PID 15496
OS PID 15498
OS PID 15499
```

```
work took 20.711 seconds
```

The first eight lines are the PID that was printed inside each `MuchCPU` instance. The last line shows that the 100 million summations can run in about 20 seconds. During those 20 seconds, all eight cores were running at 100 percent. On some laptops, the fans might start buzzing away trying to dissipate the heat.

If we subclass `threading.Thread` instead of `multiprocessing.Process` in `MuchCPU`, the output looks as follows:

```
% python src/processes_1.py
OS PID 15772
OS PID 15772
OS PID 15772
OS PID 15772
OS PID 15772
OS PID 15772
OS PID 15772
OS PID 15772
work took 69.316 seconds
```

This time, the threads are running inside the same operating system process and take over three times as long to run. The display showed that no core was particularly busy, suggesting that the work was being shunted around among the various cores. This is mostly because there are no I/O or other operating system services during this huge computation. The point of multiple threads is to allow a thread to proceed while others are waiting for operating system requests. Without any operating system requests, the threads are all demanding to be scheduled in one lonely core.

We might expect a single process version to take at least eight times as long as the eight-process version. The lack of a simple multiplier suggests that there are a number of factors involved in how the low-level instructions are processed by Python, the operating system schedulers, and even the hardware itself. This suggests that timing predictions are difficult, and it's best to plan on running multiple performance tests with multiple software architectures.

Starting and stopping individual `Process` instances involves a lot of overhead. The most common use case, therefore, is to create a pool of workers and assign tasks to them. We'll look at this next.

Multiprocessing pools

Because the OS keeps each process separate from all others, interprocess communication becomes an important consideration. We need to pass data between these separate processes. One really common example is having one process write a file that another process can read. When the two processes are reading and writing a file, and running concurrently, we have to be sure that the reader is waiting for the writer to produce data. The operating system *pipe* structure can accomplish this for us. Within the shell, we can write `ps -ef | grep python` and pass output from the `ps` command to the `grep` command. The two commands run concurrently. For Windows PowerShell users, there are similar kinds of pipeline processing, using different command names. (See *Pipelines* (`https://docs.microsoft.com/en-us/powershell/scripting/learn/ps101/04-pipelines?view=powershell-7.1`) in the PowerShell documentation for examples.)

The `multiprocessing` package provides some additional ways to implement interprocess communication. Pools can seamlessly hide the way data is moved between processes. Using a pool looks much like a function call: you pass data into a function, it is executed in another process or processes, and when the work is done, a value is returned. It is important to understand how much work is being done to support this: objects in one process are pickled and passed into an operating system pipe. Then, another process retrieves data from the pipe and unpickles it. The requested work is done in the subprocess and a result is produced. The result is pickled and passed back through the pipe. Eventually, the original process unpickles and returns a Python object. Collectively, we call these pickling, transferring, and unpickling steps *serializing* the data.

The serialization to communicate between processes takes time and memory. We want to get as much useful computation done with the smallest serialization cost. The ideal mix depends on the size and complexity of the objects being exchanged, meaning that different data structure designs will have different performance levels.

Performance predictions are difficult to make. It's essential to profile the application to ensure the concurrency design is effective.

The code to make all this machinery work is surprisingly simple. Let's look at the problem of calculating all the prime factors of a list of random numbers. This is a common part of a variety of cryptography algorithms (not to mention attacks on those algorithms!).

It requires months, possibly years, of processing power to factor the 232-digit numbers used by some encryption algorithms. The following implementation, while readable, is not at all efficient; it would take years to factor even a 100-digit number. That's okay because we want to see it using

lots of CPU time factoring 9-digit numbers:

```
from math import sqrt, ceil
import random
from multiprocessing.pool import Pool


def prime_factors(value: int) -> list[int]:
```

```
    if value in {2, 3}:
        return [value]
    factors: list[int] = []
    for divisor in range(2, ceil(sqrt(value)) + 1):
        quotient, remainder = divmod(value, divisor)
        if not remainder:
            factors.extend(prime_factors(divisor))
            factors.extend(prime_factors(quotient))
            break
    else:
        factors = [value]
    return factors


if __name__ == "__main__":
    to_factor = [random.randint(100_000_000, 1_000_000_000) for i in
    range(40_960)]
    with Pool() as pool:
        results = pool.map(prime_factors, to_factor)
    primes = [
        value for value, factor_list in zip(to_factor, results) if
        len(factor_list) == 1
    ]
    print(f"9-digit primes {primes}")
```

Let's focus on the parallel processing aspects, as the brute-force recursive algorithm for calculating factors is pretty clear. We create the `to_factor` list of 40,960 individual numbers. Then we construct a multiprocessing `pool` instance. By default, this pool creates a separate process for each of the CPU cores on the machine it is running on.

The `map()` method of the pool accepts a function and an iterable. The `map()` method pickles each of the values in the iterable and passes them to an available worker process in the pool. The worker executes the function on the object provided. When that process is finished doing its work, the

`map()` method pickles the available resulting list of factors so it can be returned to the calling process. Meanwhile, if the pool has more work available, the worker takes on the next job.

The pickling of results is an important overhead. For small objects (such as numbers), pickling happens quickly. For complicated objects, it can take more time. In the rare case of a class of objects that can't be pickled, this approach won't work at all.

Once all the workers in the pool are finished processing (which could take some time), a `results` list is available to the original process. This process has been waiting patiently for all this work to complete. The results of `map()` will be in the same order as the requests. This makes it sensible to use `zip()` to match up the original value with the computed prime factors.

It is often more useful to use the similar `map_async()` method, which returns each result object immediately. In that case, the `results` variable is not a list of values, but a contract (or a deal or an obligation) to return a list of values in the future when the client calls `results.get()`. This future object also has methods such as `ready()` and `wait()`, which allow us to check whether all the results are in yet. This is suitable for processing where the completion time is highly variable.

Alternatively, if we don't know all the values we want to get results for in advance, we can use the `apply_async()` method to queue up a single job. If the pool has a process that isn't already working, it will start immediately; otherwise, it will hold onto the task until there is a free worker process available.

Pools can also be `closed`; they refuse to take any further tasks, but continue to process everything currently in the queue. They can also be `terminated`, which goes one step further and refuses to start any jobs still in the queue, although any jobs currently running are still permitted to complete.

There are a number of constraints on how many workers make sense, including the following:

- Only `cpu_count()` processes can be computed simultaneously; any number can be waiting. If the workload is CPU-intensive, a larger pool of workers won't compute any faster. If the workload involves a lot of input/output, however, a large pool might improve the rate at which work is completed.
- For very large data structures, the number of workers in the pool may need to be reduced to make sure memory is used effectively.
- Communication between processes is expensive; easily serialized data is the best policy.

- Creating new processes takes a non-zero amount of time; a pool of a fixed size helps minimize the impact of this cost.

The multiprocessing pool gives us a tremendous amount of computing power with relatively little work on our part. We need to define a function that can perform the parallelized computation, and we need to map arguments to that function using an instance of the `multiprocessing.Pool` class.

In many applications, we need to do more than a mapping from a parameter value to a complex result. For these applications, the simple `pool.map()` may not be enough. For more complicated data flows, we can make use of explicit queues of pending work and computed results. We'll look at creating a network of queues next.

Queues

If we need more control over communication between processes, the queue.Queue data structure is useful. There are several variants offering ways to send messages from one process to one or more other processes. Any picklable object can be sent into a `Queue`, but remember that pickling can be a costly operation, so keep such objects small. To illustrate queues, let's build a little search engine for text content that stores all relevant entries in memory.

This particular search engine scans all files in the current directory in parallel. A process is constructed for each core on the CPU. Each of these is instructed to load some of the files into memory. Let's look at the function that does the loading and searching:

```
from collections.abc import Iterator
from pathlib import Path
from multiprocessing import Queue


type Query_Q = Queue[str | None]
type Result_Q = Queue[list[str]]


def search(paths: list[Path], query_q: Query_Q, results_q: Result_Q) ->
None:
    print(f"PID: {os.getpid()}, paths {len(paths)}")
    lines: list[str] = []
    for path in paths:
        lines.extend(
            line.rstrip()
```

```
            for line in path.read_text().splitlines()
        )

    while True:
        if (query_text := query_q.get()) is None:
            break
        results = [line for line in lines if query_text in line]
        results_q.put(results)
```

Remember, the `search()` function is run in a separate process (in fact, it is run in `cpu_count()` separate processes) from the main process that created the queues. Each of these processes is started with a list of `pathlib.Path` objects, and two `multiprocessing.Queue` objects: one for incoming queries and one to send outgoing results. These queues automatically pickle the data in the queue and pass it into the subprocess over a pipe. These two queues are set up in the main process and passed through the pipes into the search function inside the child processes.

The `search()` function does two separate things:

1. When it starts, it opens and reads all the supplied files in the list of `Path` objects. Each line of text in those files is accumulated into the `lines` list. This preparation is relatively expensive, but it's done exactly once.
2. The `while` statement is the main event-processing loop for search. It uses `query_q.get()` to get a request from its queue. It searches lines. It uses `results_q.put()` to put a response into the results queue.

The `while` statement has the characteristic design pattern for queue-based processing. The process will get a value from a queue of some work to perform, perform the work, and then put the result into another queue. We can decompose very large and complex problems into processing steps and queues so that the work is done concurrently, producing more results in less time. This technique also lets us tailor the processing steps and the number of workers to make best use of a processor.

The main part of the application builds this pool of workers and their queues. We'll follow the **Façade** design pattern (refer back to *Chapter 12* for more information). The idea here is to define a class, `DirectorySearch`, to wrap the queues and the pool of worker processes into a single object.

This object can set up the queues and the workers, and an application can then interact with them by posting a query and consuming the replies:

```
from fnmatch import fnmatch
import os


class DirectorySearch:
    def __init__(self) -> None:
        self.query_queues: list[Query_Q]
        self.results_queue: Result_Q
        self.search_workers: list[Process]

    def setup_search(self, paths: list[Path], cpus: int | None = None) ->
    None:
        if cpus is None:
            cpus = cpu_count()
        worker_paths = [paths[i::cpus] for i in range(cpus)]
        self.query_queues = [Queue() for p in range(cpus)]
        self.results_queue = Queue()

        self.search_workers = [
            Process(target=search, args=(paths, q, self.results_queue))
            for paths, q in zip(worker_paths, self.query_queues)
        ]
        for proc in self.search_workers:
            proc.start()

    def teardown_search(self) -> None:
        # Signal process termination
        for q in self.query_queues:
            q.put(None)

        for proc in self.search_workers:
            proc.join()

    def search(self, target: str) -> Iterator[str]:
        # print(f"search queues={self.query_queues}")
        for q in self.query_queues:
            q.put(target)

        for i in range(len(self.query_queues)):
            for match in self.results_queue.get():
                yield match
```

The `setup_search()` method prepares the worker subprocesses. The `[i::cpus]` slice operation lets us break this list into a number of equally sized parts. If the number of CPUs is 8, the step size will be 8, and we'll use 8 different offset values from 0 to 7. We also construct a list of `Queue` objects to send data into each worker process. Finally, we construct a **single** results queue. This is passed into all of the worker subprocesses. Each of them can put data into the queue and it will be aggregated in the main process.

Once the queues are created and the workers started, the `search()` method provides the target to all the workers at one time. They can then all commence examining their separate collections of data to emit answers.

Since we're searching a fairly large number of directories, we use a generator function, `all_source()`, to locate all the `*.py` `Path` objects under the given `base` directory. Here's the function to find all the source files:

```
def all_source(path: Path, pattern: str) -> Iterator[Path]:
    for root, dirs, files in os.walk(path):
        for skip in {".tox", ".mypy_cache", "__pycache__", ".idea",
        ".venv"}:
            if skip in dirs:
                dirs.remove(skip)
        yield from (Path(root) / f for f in files if fnmatch(f, pattern))
```

The `all_source()` function uses the `os.walk()` function to examine a directory tree, rejecting file directories that are filled with files we don't want to look at. This function uses the `fnmatch` module to match a filename against the kind of wildcard patterns the Linux shell uses. We can use a pattern parameter of `'*.py'`, for example, to find all files with names ending in `.py`. This seeds the `setup_search()` method of the `DirectorySearch` class.

The `teardown_search()` method of the `DirectorySearch` class puts a special termination value into each queue. Remember, each worker is a separate process, executing the `while` statement inside the `search()` function and reading from a queue of requests. When it reads a `None` object, it will break out of the `while` statement and exit the function. We can then use the `join()` to collect all the child processes, cleaning up politely. (If we don't do the `join()`, some Linux distros can leave "zombie processes" — children not properly rejoined with their parent because the parent crashed; these consume system resources and often require a reboot.)

Now let's look at the code that makes a search actually happen:

```
from multiprocessing import Process, Queue, cpu_count
import time

if __name__ == "__main__":
    ds = DirectorySearch()
    base = Path.cwd().parent
    all_paths = list(all_source(base, "*.py"))
    ds.setup_search(all_paths)
    for target in ("import", "class", "def"):
        start = time.perf_counter()
        count = 0
        for line in ds.search(target):
            # print(line)  # If you want to see what's going on
            count += 1
        milliseconds = 1000 * (time.perf_counter() - start)
        print(
            f"Found {count} {target!r} in {len(all_paths)} files "
            f"in {milliseconds:.3f}ms"
        )
    ds.teardown_search()
```

This code creates a `DirectorySearch` object, `ds`, and provides all of the source paths starting from the parent of the current working directory, via `base = Path.cwd().parent`. Once the workers are prepared, the `ds` object performs searches for a few common strings, "`import`", "`class`", and "`def`". Note that we've commented out the `print(line)` statement that shows the useful results. We're interested in performance of the search. Since there's nothing we can do about performance of the operating system graphics kernel updating the console display, we don't want to measure this. The initial file reads take a fraction of a second to get started. Once all the files are read, however, the time to do the search is dramatic. On a MacBook Pro with 134 files of source code, the output looks like this:

```
% python src/directory_search.py
PID: 29387, paths 19
PID: 29389, paths 19
PID: 29388, paths 19
PID: 29390, paths 19
PID: 29391, paths 19
PID: 29392, paths 19
PID: 29393, paths 19
```

```
PID: 29394, paths 18
Found 611 'import' in 151 files in 190.105ms
Found 464 'class' in 151 files in 1.293ms
Found 1036 'def' in 151 files in 1.330ms
```

The search for `"import"` took about 190 milliseconds (0.190 seconds.) Why was this so slow compared to the other two searches? It's because the `search()` function was still reading the files when the first request was put in the queue. The first request's performance reflects the one-time startup cost of loading the file content into memory. The next two requests run in about 1 millisecond each. That's amazing! That's almost 1,000 searches per second on a laptop with only a few lines of Python code.

This example of queues to feed data among workers is a single-host version of what could become a distributed system. Imagine the searches were being sent out to multiple host computers and then recombined. Now imagine you had access to the fleet of computers in Google's data centers and you might understand why they can return search results so quickly!

We won't discuss it here, but the `multiprocessing` module includes a manager class that can take a lot of the boilerplate out of the preceding code. There is even a version of `multiprocessing.Manager` that can manage subprocesses on remote systems to construct a rudimentary distributed application. Check the Python `multiprocessing` documentation if you are interested in pursuing this further.

The problems with multiprocessing

As with threads, multiprocessing also has problems, some of which we have already discussed. Sharing data between processes is costly. As we have discussed, all communication between processes, whether by queues or operating system pipes, requires serializing the objects. Excessive serialization can dominate processing time. Multiprocessing works best when relatively small objects are passed between processes and a tremendous amount of work needs to be done on each object.

Using shared memory can avoid some of the cost of repeated serialization and deserialization. There are numerous limitations on the kinds of Python objects that can be shared. Shared memory can help performance, but can also lead to somewhat more complex-looking Python objects.

The other major problem with multiprocessing is that, like threads, it can be hard to tell which process a variable or method is being accessed in. In multiprocessing, the worker processes inherit a great deal of data from the parent process. This isn't *shared*; it's a one-time copy. A child can be

given a copy of a mapping or a list and mutate the object. The parent won't see the effects of the child's mutation.

A big advantage of multiprocessing is the absolute independence of processes. We don't need to carefully manage locks, because the data is not shared. Additionally, the internal operating system's limits on numbers of open files are allocated at the process level; we can have a large number of resource-intensive processes.

When designing concurrent applications, the focus is on maximizing the use of the CPU to do as much work in as short a time as possible. With so many choices, we always need to examine the problem to figure out which of the many available solutions is the best one for that problem.

The notion of concurrent processing is too broad for there to be one right way to do it. Each distinct problem has a best solution. It's important to write code in a way that permits adjustment, tuning, and optimization.

We've looked at the two principal tools for concurrency in Python: threads and processes. Threads exist within a single operating system process, sharing memory and other resources. Processes are independent of each other, which makes interprocess communication a necessary overhead. Both of these approaches are amenable to the concept of a pool of concurrent workers waiting to work and providing results at some unpredictable time in the future. This abstraction of results becoming available in the future is what shapes the `concurrent.futures` module. We'll look at this next.

Futures

Let's start looking at a more asynchronous way of implementing concurrency. The concept of a "future" or a "promise" is a handy abstraction for describing concurrent work. A **future** is an object that wraps a function call. That function call is run in the *background*, in a thread or a separate process. The `future` object has methods to check whether the computation has completed and — if it has — to get the results. We can think of it as a computation where the results will arrive in the future, and we can do something else while waiting for them.

See the book *Asynchronous Programming with Futures and Promises* (`https://hub.packtpub.com/asynchronous-programming-futures-and-promises/`) from Packt for some additional background.

In Python, the `concurrent.futures` module wraps either `multiprocessing` or `threading` depending on what kind of concurrency we need. A future doesn't completely solve the problem of accidentally altering shared state, but using futures allows us to structure our code such that it can be easier to track down the cause of the problem when we do so.

Futures can help manage boundaries between the different threads or processes. Similar to the multiprocessing pool, they are useful for **call-and-answer** type interactions, in which processing can happen in another thread (or process), and then at some point in the future, you will ask it for the result. It's a wrapper around multiprocessing pools and thread pools, but it provides a cleaner API and encourages nicer code.

Let's see another, more sophisticated file search and analyze an example. In the last section, we implemented a version of the Linux `grep` command. This time, we'll create a simple version of the `find` command that bundles in a clever analysis of Python source code. We'll start with the analytical part since it's central to the work we need to be done concurrently:

```
import ast
from pathlib import Path
from typing import NamedTuple


class ImportResult(NamedTuple):
    path: Path
    imports: set[str]

    @property
    def focus(self) -> bool:
        return "typing" in self.imports


class ImportVisitor(ast.NodeVisitor):
    def __init__(self) -> None:
        self.imports: set[str] = set()

    def visit_Import(self, node: ast.Import) -> None:
        # print(ast.dump(node))
        for alias in node.names:
            self.imports.add(alias.name)

    def visit_ImportFrom(self, node: ast.ImportFrom) -> None:
        # print(ast.dump(node))
        if node.module:
            self.imports.add(node.module)


def find_imports(path: Path) -> ImportResult:
```

```
    tree = ast.parse(path.read_text())
    iv = ImportVisitor()
    iv.visit(tree)
    return ImportResult(path, iv.imports)
```

We've defined a few things here. We started with a named tuple, `ImportResult`, which binds a `Path` object and a set of strings together. It has a property, `focus`, that looks for the specific string, `"typing"`, in the set of strings. We'll see why this string is so important in a moment.

The `ImportVisitor` class is built using the `ast` module in the standard library. An **Abstract Syntax Tree** (**AST**) is the parsed source code, usually from a formal programming language. Python code, after all, is just a bunch of characters; the AST for Python code groups the text into meaningful statements and expressions, variable names, and operators, all of the syntactic components of the language. A visitor class has a method to examine the parsed code. We provided overrides for two methods of the `NodeVisitor` class, so we will look at only the two kinds of `import` statements: `import x` and `from x import y`. The details of how each `node` data structure works are a bit beyond this example, but the `ast` module documentation in the standard library describes the unique structure of each Python language construct.

The `find_imports()` function reads some source, parses the Python code, visits the `import` statements, and then returns an `ImportResult` with the original `Path` and the set of names found by the visitor. This is — in many ways — a lot better than a simple pattern match for `"import"`. For example, using an `ast.NodeVisitor` will skip over comments and ignore the text inside character string literals, two jobs that are hard with regular expressions.

There isn't anything particularly special about the `find_imports()` function, but note how it does not access any global variables. All interaction with the external environment is passed into the function or returned from it. This is not a technical requirement, but it is the best way to keep your brain inside your skull when programming with futures.

We want to process hundreds of files in dozens of directories, though. The best approach is to have lots and lots of these running all at the same time, clogging the cores of our CPU with lots and lots of computing:

```
def main(base: Path = Path.cwd()) -> None:
    print(f"\n{base}")
    start = time.perf_counter()
```

```
    with futures.ThreadPoolExecutor(24) as pool:
        analyzers = [
            pool.submit(find_imports, path) for path in all_source(base,
            "*.py")
        ]
        analyzed = (worker.result() for worker in
        futures.as_completed(analyzers))
    for example in sorted(analyzed):
        print(
            f"{'->' if example.focus else '':2s} "
            f"{example.path.relative_to(base)} {example.imports}"
        )
    end = time.perf_counter()
    rate = 1000 * (end - start) / len(analyzers)
    print(f"Searched {len(analyzers)} files in {base} at {rate:.3f}ms/file")
```

We're leveraging the same `all_source()` function shown in the *Queues* section earlier in this chapter; this needs a base directory to start searching in, and a pattern, such as `"*.py"`, to find all the files with the `.py` extension. We've created a `ThreadPoolExecutor`, assigned to the `pool` variable, with two dozen worker threads, all waiting for something to do. We create a list of `Future` objects in the `analyzers` object. This list is created by a list comprehension applying the `pool.submit()` method to our search function,`find_imports()`, and a `Path` from the output of `all_source()`.

The threads in the pool will immediately start working on the submitted list of tasks. As each thread finishes work, it saves the results in the `Future` object and picks up some more work to do.

Meanwhile, in the foreground, the parent application uses a generator expression to evaluate the `result()` method of each `Future` object. Note that the futures are visited using the `futures.as_completed()` generator. The function starts providing complete `Future` objects as they become available. This means the results may not be in the order that they were originally submitted. There are other ways to visit the futures; we can, for example, wait until all are complete and then visit them in the order they were submitted, in case that's important.

We extract the result from each `Future`. From the type hints, we can see that this will be an `ImportResult` object with a `Path` and a set of strings; these are the names of the imported modules. We can sort the results, so the files show up in some sensible order.

On a MacBook Pro, this takes about 1.689 milliseconds (0.001689 seconds) to process each file. The 24 individual threads easily fit in a single process without stressing the operating system. Increasing

the number of threads doesn't materially affect the elapsed runtime, suggesting any remaining bottleneck is not due to concurrent computation but the initial scan of the directory tree and the creation of the thread pool.

And the `focus` feature of the `ImportResult` class? Why is the `typing` module special? We needed to review each chapter's type hints when a new release of **mypy** came out during the development of this book. It was helpful to separate the modules into those that required careful checking and those that didn't need to be revised.

And that's all that is required to develop a futures-based I/O-bound application. Under the hood, it's using the same thread or process APIs that we've already discussed, but it provides a more understandable interface. It makes it easier to see the boundaries between concurrently running functions.

Accessing global variables without proper synchronization can result in a problem called a **race condition**. For example, imagine two concurrent writes trying to increment an integer counter. They start at the same time and both read the current value of the shared variable as 5. One thread is first in the race; it increments the value and writes 6. The other thread comes in second; it increments what the variable was and also writes 6. But if two processes are trying to increment a variable, the expected result would be that it gets incremented by 2, so the result should be 7.

Modern wisdom is that the easiest way to avoid doing this is to keep as much state as possible private and share them through known-safe constructs, such as queues or futures.

For many applications, the `concurrent.futures` module is the place to start with designing the Python code. The lower-level `threading` and `multiprocessing` modules offer some additional constructs for very complex cases.

Using `run_in_executor()` allows an application to leverage the `concurrent.futures` module's `ProcessPoolExecutor` or `ThreadPoolExecutor` classes to farm work out to multiple processes or multiple threads. This provides a lot of flexibility within a tidy, ergonomic API.

In some cases, we don't really need concurrent processes. In some cases, we simply need to be able to toggle back and forth between waiting for data and computing when data becomes available. The `async` features of Python, including the `asyncio` module, can interleave processing within a single thread. We'll look at this variation on the theme of concurrency next.

AsyncIO

AsyncIO combines the concept of futures and an event loop with coroutines. The result is helpful for writing responsive applications that don't seem to waste time waiting for input.

For the purposes of working with Python's `async` features, a *coroutine* is a function that is waiting for an event, and also can provide events to other coroutines. In Python, we implement coroutines using `async def`. A function with `async` must work in the context of an **event loop**, which switches control of the thread among the coroutines waiting for events. We'll see a few Python constructs using `await` expressions to show where the event loop can switch to another waiting `async` function.

It's crucial to recognize that `async` operations are interleaved, and not — generally — parallel. At most one coroutine is in control and processing, and all the others are waiting for an event. The idea of interleaving is described as **cooperative multitasking**: an application can be processing data while also waiting for the next request message to arrive. As data becomes available, the event loop can transfer control to one of the waiting coroutines.

The AsyncIO implementation has a bias toward network I/O. Most networking applications, especially on the server side, spend a lot of time waiting for data to come in from the network. AsyncIO can be more efficient than handling each client in a separate thread; then some threads can be working while others are waiting. The problem is the threads use up memory and other resources. AsyncIO uses coroutines to interleave processing cycles when the data becomes available.

Thread scheduling depends on operating system requests the thread makes (and, to an extent, the GIL's interleaving of threads). Process scheduling depends on the overall scheduler for the operating system. Both thread and process scheduling are **preemptive** — the thread (or process) can be interrupted to allow a different, higher-priority thread or process to control the CPU. This means thread scheduling is unpredictable, and locks are important if multiple threads are going to update a shared resource. At the operating system level, shared locks are required if two processes want to update a shared operating system resource such as a file. Unlike threads and processes, AsyncIO coroutines are **non-preemptive**; they explicitly hand control to each other at specific points in the processing, removing the need for explicit locking of shared resources.

The `asyncio` library provides a built-in *event loop*: this is the loop that handles interleaving control among the running coroutines. However, the event loop comes with a cost. When we run code in an `async` task on the event loop, that code *must* return immediately, blocking neither on I/O nor on long-running calculations. This is a minor thing when writing our own code, but it means that any standard library or third-party functions that block on I/O must be wrapped with an `async def`

function that can handle the waiting politely.

When working with `asyncio`, we'll write our application as a set of coroutines that use `async` and `await` syntax to interleave control via the event loop. The top-level "main" program's job, then, is simplified to starting the event loop so the coroutines can then hand control back and forth, interleaving waiting and working.

AsyncIO in action

A canonical example of a blocking function is the `time.sleep()` call. We can't call the `time` module's `sleep()` directly, because it would seize control, stalling the event loop and preventing interleaving of coroutines. We'll use the version of `sleep()` in the `asyncio` module. Used in an `await` expression, the event loop can interleave another coroutine while waiting for the `sleep()` to finish. Let's use the asynchronous version of this call to illustrate the basics of an AsyncIO event loop, as follows:

```
import asyncio
import random


async def random_sleep(counter: int) -> None:
    delay = random.random() * 5
    print(f"{counter} sleeps for {delay:.2f} seconds")
    await asyncio.sleep(delay)
    print(f"{counter} awakens, refreshed")


async def sleepers(how_many: int = 5) -> None:
    print(f"Creating {how_many} tasks")
    tasks = [asyncio.create_task(random_sleep(i)) for i in range(how_many)]
    print(f"Waiting for {how_many} tasks")
    await asyncio.gather(*tasks)


if __name__ == "__main__":
    asyncio.run(sleepers(5))
    print("Done with the sleepers")
```

This example covers several features of AsyncIO programming. The overall processing is started by the `asyncio.run()` function. This starts the event loop, executing the `sleepers()` coroutine. Within the `sleepers()` coroutine, we create a handful of individual tasks; these are instances of the `random_sleep()` coroutine with a given argument value. `random_sleep()` uses `asyncio.sleep()`

to simulate a long-running request.

Note that the `print()` function involves some overhead. When setting up performance benchmarking, we're often pulling a few elements out of a larger application or module. The presence of `print()` can help confirm that the processing being benchmarked is working correctly. When gathering performance numbers, the (`print()`) statements will often be commented out, using `#`.

Because this is built using `async def` functions and an `await` expression around `asyncio.sleep()`, execution of the `random_sleep()` functions and the overall `sleepers()` function is interleaved. While the `random_sleep()` requests are started in order of their `counter` parameter value, they finish in a completely different order. Here's an example:

```
python src/async_1.py
Creating 5 tasks
Waiting for 5 tasks
0 sleeps for 4.69 seconds
1 sleeps for 1.59 seconds
2 sleeps for 4.57 seconds
3 sleeps for 3.45 seconds
4 sleeps for 0.77 seconds
4 awakens, refreshed
1 awakens, refreshed
3 awakens, refreshed
2 awakens, refreshed
0 awakens, refreshed
Done with the sleepers
```

We can see the `random_sleep()` function with a `counter` value of `4` had the shortest sleep time, and was given control first when it finished the `await asyncio.sleep()` expression. The order of waking is strictly based on the random sleep interval, and the event loop's ability to hand control from coroutine to coroutine.

As asynchronous programmers, we don't need to know too much about what happens inside that `run()` function, but be aware that it tracks which of the available coroutines are waiting and which should have control at the current moment.

A task, in this context, is an object that `asyncio` knows how to schedule in the event loop. This includes the following:

- Coroutines defined with the `async def` statement

- `asyncio.Future` objects: These are almost identical to the `concurrent.futures` you saw in the previous section, but for use with `asyncio`
- Any awaitable object, that is, one with an `__await__()` function

In this example, all the tasks are coroutines; we'll see some of the others in later examples.

Look a little more closely at that `sleepers()` coroutine. It first constructs instances of the `random_sleep()` coroutine. These are each wrapped in an `asyncio.create_task()` call, which adds these as futures to the loop's task queue so they can execute and start immediately when control is returned to the loop.

Control is returned to the event loop whenever we call `await`. In this case, we call `await asyncio.gather()` to yield control to other coroutines until all the tasks are finished.

Each of the `random_sleep()` coroutines prints a starting message, then sends control back to the event loop for a specific amount of time using its own `await` calls. When the sleep has completed, the event loop passes control back to the relevant `random_sleep()` task, which prints its awakening message before returning.

The `async` keyword acts as documentation notifying the Python interpreter (and people reading the code) that the coroutine contains the `await` calls. It also does some work to prepare the coroutine to run on the event loop. It behaves much like a decorator; in fact, back in Python 3.4, it used to be implemented as an `@asyncio.coroutine` decorator.

Reading an AsyncIO future

An AsyncIO coroutine executes each line of code in its body until it encounters an `await` expression, at which point it returns control to the event loop. The event loop then executes any other tasks that are ready to run, including the one that the original coroutine was waiting on. Whenever that child task completes, the event loop sends the result back into the coroutine so that it can pick up execution until it encounters another `await` expression or returns.

This allows us to write code that executes synchronously until we explicitly need to wait for something. The processing is deterministic. Threads — and processes — on the other hand, depend on the operating system scheduler and the operating system workload, making them non-deterministic. The deterministic execution means we don't need to worry nearly so much about shared state. Think of the operating system schedulers as intentionally and wickedly evil; they will maliciously (somehow) find the worst possible sequence of operations among processes and threads.

Throughout this chapter, we'll hit on one key point fairly often.

It's a good idea to limit shared state.

A *share nothing* philosophy can prevent a ton of difficult bugs stemming from sometimes-hard-to-imagine timelines of interleaved operations.

The real value of AsyncIO is the way it allows us to collect logical sections of code together inside a single coroutine, even if we are waiting for other work elsewhere. As a specific instance, even though the `await asyncio.sleep` call in the `random_sleep()` coroutine is allowing a ton of stuff to happen inside the event loop, the coroutine itself looks like it's doing everything in order. This ability to read related pieces of asynchronous code without worrying about the machinery that waits for tasks to complete is the primary benefit of the AsyncIO module.

AsyncIO for networking

AsyncIO was specifically designed for use with network sockets, so let's implement a server using the `asyncio` module. Looking back at *Chapter 13*, we created a fairly complex server to catch log entries being sent from one process to another process using sockets. At the time, we used it as an example of a complex resource we didn't want to set up and tear down for each test.

We'll rewrite that example, creating an `asyncio`-based server that can handle requests from a (large) number of clients. It can do this by having lots of coroutines, all waiting for log records to arrive. When a record arrives, one coroutine can save the record, doing some computation, while the remaining coroutines wait for something to do.

In *Chapter 13*, we were interested in writing a test for the integration of a log catcher process with separate log-writing client application processes. *Figure 14.1* illustrates the relationships involved.

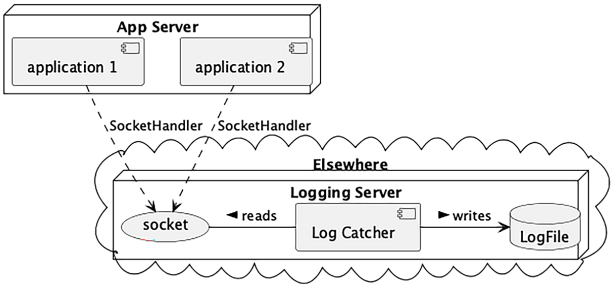


Figure 14.1: The log catcher in the sky

The log catcher process creates a socket server to wait for connections from all client applications. Each of the client applications uses `logging.SocketHandler` to direct log messages to the waiting server. The server collects the messages and writes them to a single, central log file.

This test was based on an example with a weak implementation. To keep things simple, the log server only worked with one application client at a time. We want to revisit the idea of a server that collects log messages. This improved implementation will handle a very large number of concurrent clients because it uses AsyncIO techniques.

The central part of design is a coroutine that reads log entries from a socket. This involves waiting for the bytes that comprise a header, then decoding the header to compute the size of the payload. The coroutine can read the right number of bytes for the log message payload, and then use a separate coroutine to process the payload. Here's the `log_catcher()` function:

```
SIZE_FORMAT = ">L"
SIZE_BYTES = struct.calcsize(SIZE_FORMAT)


async def log_catcher(
    reader: asyncio.StreamReader, writer: asyncio.StreamWriter
) -> None:
    count = 0
    client_socket = writer.get_extra_info("socket")
    size_header = await reader.read(SIZE_BYTES)
    while size_header:
        payload_size = struct.unpack(SIZE_FORMAT, size_header)
        bytes_payload = await reader.read(payload_size[0])
        await log_writer(bytes_payload)
        count += 1
        size_header = await reader.read(SIZE_BYTES)
    print(f"From {client_socket.getpeername()}: {count} lines")
```

This `log_catcher()` function implements the protocol used by the `logging` module's `SocketHandler` class. Each log entry is a block of bytes we can decompose into a header and a payload. We need to read the first few bytes, saved in `size_header`, to get the size of the message that follows. Once we have the size, we can wait for the payload bytes to arrive. Since the two reads are both `await` expressions, other coroutines can work while this function is waiting for the header and payload bytes to arrive.

The `log_catcher()` function is invoked by a server that provides the coroutine with a `StreamReader`

and `StreamWriter`. These two objects wrap the socket pair that is created by the TCP/IP protocol. The stream reader (and the writer) are properly async-aware objects, and we can use `await` when waiting to read bytes from the client.

This `log_catcher()` function waits for socket data, then provides data to another coroutine, `log_writer()`, for conversion and writing. The `log_catcher()` function's job is to do a lot of waiting, and then shuttle the data from reader to writer; it also does an internal computation to count messages from a client. Incrementing a counter is not much, but it is work that can be done while waiting for data to arrive.

Here's a function, `serialize()`, and a coroutine, `log_writer()`, to convert log entries to JSON notation and write them to a file:

```
TARGET: TextIO
LINE_COUNT = 0


def serialize(bytes_payload: bytes) -> str:
    object_payload = pickle.loads(bytes_payload)
    text_message = json.dumps(object_payload)
    TARGET.write(text_message)
    TARGET.write("\n")
    return text_message

async def log_writer(bytes_payload: bytes) -> None:
    global LINE_COUNT
    LINE_COUNT += 1
    await asyncio.to_thread(serialize, bytes_payload)
```

The `serialize()` function needs to have an open file, `TARGET`, to which the log messages are written. The file open (and close) needs to be taken care of elsewhere in the application; we'll look at these operations next. The `serialize()` function is used by the `log_writer()` coroutine. Because `log_writer()` is an `async` coroutine, other coroutines will be waiting to read and decode input messages while this coroutine is writing them.

The `serialize()` function actually does a fair amount of computation. It also harbors a profound problem. The file write operation can be blocked, that is, stuck waiting for the operating system to finish the work. Writing to a disk means handing the work to a disk device and waiting until the device responds that the write operation is complete. While a microsecond to write a 1,000-character line of data may seem fast, it's forever to a CPU. This means all file operations will block their

thread waiting for the operation to complete. To work politely with the other coroutines in the main thread, we assign this blocking work to a separate thread. This is why the `log_writer()` coroutine uses `asyncio.to_thread()` to allocate this work to a separate thread.

Because the `log_writer()` coroutine uses `await` on this separate thread, it returns control to the event loop while the thread waits for the write to complete. This polite `await` allows other coroutines to work while the `log_writer()` coroutine is waiting for `serialize()` to complete.

We've passed two kinds of work to a separate thread:

- A compute-intensive operation. These are the `pickle.loads()` and `json.dumps()` operations.
- A blocking operating system operation. This is `TARGET.write()`. These blocking operations include most operating system requests, including file operations. They do not include the various network streams that are already part of the `asyncio` module. As we saw in the `log_catcher()` function, the streams are already polite users of the event loop.

This technique of passing work to a thread is how we can make sure the event loop is spending as much time waiting as possible. If all the coroutines are waiting for an event, then whatever happens next will be responded to as quickly as possible. This principle of many waiters is the secret to a responsive service.

The `LINE_COUNT` global variable can raise some eyebrows. Recall from previous sections that we raised dire warnings about the consequences of multiple threads updating a shared variable concurrently. With `asyncio`, we don't have preemption among threads. Because each coroutine uses explicit `await` requests to give control to other coroutines via the event loop, we can update this variable in the `log_writer()` coroutine knowing the state change will effectively be atomic — an indivisible update — among all the coroutines.

This doesn't make global variables a good idea. Ordinary object-oriented refactoring is still recommended to make sure the state is properly encapsulated in a class. What `asyncio` does for us is to eliminate the possibility of concurrent threads overwriting the shared, mutable object's value.

To make this example complete, here are the imports:

```
import asyncio
import asyncio.exceptions
import json
from pathlib import Path
```

```
import pickle
import struct
from typing import TextIO
```

Here's the top-level dispatcher that starts this service:

```
server: asyncio.AbstractServer


async def main(host: str, port: int) -> None:
    global server
    server = await asyncio.start_server(
        log_catcher,
        host=host,
        port=port,
    )

    if server.sockets:
        addr = server.sockets[0].getsockname()
        print(f"Serving on {addr}")
    else:
        raise ValueError("Failed to create server")

    async with server:
        await server.serve_forever()
    server.close_clients()
```

The `main()` function contains an elegant way to automatically create new `asyncio.Task` objects for each network connection. The `asyncio.start_server()` function listens at the given host address and port number for incoming socket connections. For each connection, it creates a new `Task` instance using the `log_catcher()` coroutine; this is added to the event loop's collection of coroutines. Once the server is started, the `main()` function lets it provide services forever using the server's `serve_forever()` method.

A small rewrite — from functions to a class — can replace the global `server` variable with one that's a shared attribute of an object. We've left this revision as an exercise for the reader.

The `add_signal_handler()` method of a loop deserves some explanation. For non-Windows operating systems, a process is terminated via a signal from the operating system. The signals have small numeric identifiers and symbolic names. For example, the terminate signal has a numeric

code of 15 and a name of `signal.SIGTERM`. When a parent process terminates a child process, this signal is sent. If we do nothing special, this signal will simply stop the Python interpreter. When we use the **Ctrl + C** sequence on the keyboard, this becomes a `SIGINT` signal, which leads Python to raise a `KeyboardInterrupt` exception.

The `add_signal_handler()` method of the loop lets us examine incoming signals and handle them as part of our AsyncIO processing loop. We don't want to simply stop with an unhandled exception. We want to finish the various coroutines and allow any write threads executing the `serialize()` function to complete normally. To make this happen, we connect the signal to the `server.close()` method. This ends the `serve_forever()` process cleanly, letting all the coroutines finish.

We have one more step after the server is finished. The sockets are left open. This means that a client may try to send data to a socket that will never receive the data. In order to signal the client, the socket must be closed. This is the purpose of the `close_clients()` method of a server. The sockets are closed and the clients can behave appropriately. (For example, the client can avoid trying to write to the closed socket.)

As we saw in the previous AsyncIO example, the main program is also a succinct way to start the event loop:

```
if __name__ == "__main__":
    # These often have command-line or environment overrides
    HOST, PORT = "localhost", 18842

    with Path("one.log").open("w") as TARGET:
        try:
            asyncio.run(main(HOST, PORT))

        except (asyncio.exceptions.CancelledError, KeyboardInterrupt):
            ending = {"lines_collected": LINE_COUNT}
            print(ending)
            TARGET.write(json.dumps(ending) + "\n")
```

This will open a file, setting the global `TARGET` variable used by the `serialize()` function. It uses the `main()` function to create the server that waits for connections. When the `serve_forever()` task is canceled with a `CancelledError` or `KeyboardInterrupt` exception, we can put a final summary line onto the log file. This line confirms that things completed normally, allowing us to verify that no lines were lost.

Any other exceptions will be raised normally. This helps us debug any bugs in the server.

Pragmatically, we might want to use the `argparse` module to parse command-line arguments. We might want to use a more sophisticated file-handling mechanism in `log_writer()` so we can limit the size of log files.

Design considerations

Let's look at some of the features of this design. First, the `log_writer()` coroutine passes bytes into and out of the external thread running the `serialize()` function. This is better than decoding the JSON in a coroutine in the main thread because the (relatively expensive) decoding can happen without stopping the main thread's event loop.

This call to `serialize()` is, in effect, a future. In the *Futures* section, earlier in this chapter, we saw that there are a few lines of boilerplate for using `concurrent.futures`. However, when we use futures with AsyncIO, there are almost none at all! When we use `await asyncio.to_thread()`, the `log_writer()` coroutine wraps the function call in a future and submits it to the internal thread pool executor. Our code can then return to the event loop until the future completes, allowing the main thread to process other connections, tasks, or futures. It is particularly important to put blocking I/O requests into separate threads. When the future is done, the `log_writer()` coroutine can finish waiting and can do any follow-up processing.

The `main()` coroutine used `start_server()`; the server listens for connection requests. It will provide client-specific AsyncIO read and write streams to each task created to handle a distinct connection; the task will wrap the `log_catcher()` coroutine. With the AsyncIO streams, reading from a stream is a potentially blocking call so we can call it with `await`. This means politely returning to the event loop until bytes start arriving.

It can help to consider how the workload grows inside this server. Initially, the `main()` function is the only coroutine. It creates the `server`, and now both `main()` and the `server` are in the event loop's collection of waiting coroutines. When a connection is made, the server creates a new task, and the event loop now contains `main()`, the `server`, and an instance of the `log_catcher()` coroutine. Most of the time, all of these coroutines are waiting for something to do: either a new connection for the server, or a message for the `log_catcher()`. When a message arrives, it's decoded and handed to `log_writer()`, and yet another coroutine is available. No matter what happens next, the application is ready to respond. The number of waiting coroutines is limited by available memory, so a lot of individual coroutines can be patiently waiting for work to do.

Next, we'll take a quick look at a log-writing application that uses this log catcher. The application doesn't do anything useful, but it can tie up a lot of cores for a long period of time. This will show us how responsive AsyncIO applications can be.

A log writing demonstration

To demonstrate how this log catching works, this client application writes a bunch of messages and does an immense amount of computing. To see how responsive the log catcher is, we can start a bunch of copies of this application to stress-test the log catcher.

This client doesn't leverage `asyncio`; it's a contrived example of compute-intensive work with a few I/O requests wrapped around it. Using coroutines to perform the I/O requests concurrently with the computation is — by design — unhelpful in this example.

We've written an application that applies a variation on the bogosort algorithm to some random data. Here's some information on this sorting algorithm: `https://rosettacode.org/wiki/Sorting_algorithms/Bogosort`. This isn't a practical algorithm, but it's simple: it enumerates all possible orderings, searching for one that is the desired, ascending order. Here are the imports and an abstract superclass, `Sorter`, for sorting algorithms:

```
import abc
from collections.abc import Iterable
from itertools import permutations
import logging
import logging.handlers
import os
import random
import time
import sys


logger = logging.getLogger(f"app_{os.getpid()}")


class Sorter(abc.ABC):
    def __init__(self) -> None:
        id = os.getpid()
        self.logger =
        logging.getLogger(f"app_{id}.{self.__class__.__name__}")

    @abc.abstractmethod
```

```
    def sort(self, data: list[float]) -> list[float]:
        ...
```

Next, we'll define a concrete implementation of the abstract `Sorter` class:

```
class BogoSort(Sorter):
    @staticmethod
    def is_ordered(data: tuple[float, ...]) -> bool:
        pairs: Iterable[tuple[float, float]] = zip(data, data[1:])
        return all(a <= b for a, b in pairs)

    def sort(self, data: list[float]) -> list[float]:
        self.logger.info("Sorting %d", len(data))
        start = time.perf_counter()

        ordering: tuple[float, ...] = tuple(data[:])
        permute_iter = permutations(data)
        steps = 0
        while not BogoSort.is_ordered(ordering):
            ordering = next(permute_iter)
            steps += 1

        duration = 1000 * (time.perf_counter() - start)
        self.logger.info(
            "Sorted %d items in %d steps, %.3f ms", len(data), steps,
            duration
        )
        return list(ordering)
```

The `is_ordered()` method of the `BogoSort` class checks to see whether the list of objects has been sorted properly. The `sort()` method generates all permutations of the data, searching for a permutation that satisfies the constraint defined by `is_ordered()`.

Note that a set of n values has $n!$ permutations, so this is a spectacularly inefficient sort algorithm. There are over 6 billion permutations of 13 values; on most computers, this algorithm can take years to sort 13 items into order.

A `main()` function handles the sorting and writes a few log messages. It does a lot of computation, tying up CPU resources doing nothing particularly useful. Here's a `main` program we can use to make log requests while our inefficient sort is grinding up processing time:

```
def main(workload: int = 10, sorter: Sorter = BogoSort()) -> int:
    total = 0
    for i in range(workload):
        samples = random.randint(3, 10)
        data = [random.random() for _ in range(samples)]
        sorter.sort(data)
        total += samples
    return total


if __name__ == "__main__":
    LOG_HOST, LOG_PORT = "localhost", 18842
    socket_handler = logging.handlers.SocketHandler(LOG_HOST, LOG_PORT)
    stream_handler = logging.StreamHandler(sys.stderr)
    logging.basicConfig(handlers=[socket_handler, stream_handler],
    level=logging.INFO)

    start = time.perf_counter()

    workload = 10
    logger.info("sorting %d collections", workload)
    samples = main(workload, GnomeSort())

    end = time.perf_counter()
    logger.info("produced %d entries, taking %f s", workload * 2 + 2, end -
    start)

    logging.shutdown()
```

The top-level script starts by creating a `SocketHandler` instance; this writes log messages to the log catcher service shown previously. A `StreamHandler` instance writes a message to the console. Both of these are provided as handlers for all the defined loggers. Once the logging is configured, the `main()` function is invoked with a random workload. (We've highlighted the three lines of work, to separate it from the clutter of logging configuration.)

On an 8-core MacBook Pro, this was run with 128 workers, all inefficiently sorting random numbers. The internal operating system `time` command describes the workload as using 700% of a core; that is, seven of the eight cores were completely occupied. And yet,there's still plenty of time left over to handle the log messages, edit this document, and play music in the background. Using a faster sort algorithm, we started 256 workers and generated 5,632 log messages in about 4.4 seconds.

This is 1,280 transactions per second and we were still only using 628% of the available 800%. Your performance may vary. For network-intensive workloads, AsyncIO seems to do a marvelous job of allocating precious CPU time to the coroutine with work to be done, and minimizing the time that threads are blocked waiting for something to do.

It's important to observe that AsyncIO is heavily biased toward network resources including sockets, queues, and operating system pipes. The filesystem is not a first-class part of the `asyncio` module, and therefore requires us to use the associated thread pool to handle processing that will be blocked until it's finished by the operating system.

We'll take a diversion to look at AsyncIO to write a client-side application. In this case, we won't be creating a server, but instead leveraging the event loop to make sure a client can process data very quickly.

AsyncIO clients

Because it is capable of handling many simultaneous connections, AsyncIO is very common for implementing servers. However, it is a generic networking library and provides full support for client processes as well. This is pretty important, since many microservices act as clients to other services.

Clients can be much simpler than servers, as they don't have to be set up to wait for incoming connections. We can leverage the `await asyncio.gather()` function to parcel out a lot of work, and wait to process the results when they've completed. This can work well with `asyncio.to_thread()`, which assigns blocking requests to separate threads, permitting the main thread to interleave work among the coroutines.

We can also create individual tasks that can be interleaved by the event loop. This allows the coroutines that implement the tasks to cooperatively schedule reading data along with computing the data that was read.

For this example, we'll use the `httpx` library to provide an AsyncIO-friendly HTTP request. This additional package needs to be installed with `uv add httpx` (if you're using *uv* as a virtual environment manager) or `python -m pip install httpx`.

Here's an application to make requests to the US weather service, implemented using `asyncio`. We'll focus on forecast zones useful for sailors in the Chesapeake Bay area. We'll start with some definitions:

```
import asyncio
import re
import time
from typing import NamedTuple
import httpx


class Zone(NamedTuple):
    zone_name: str
    zone_code: str
    same_code: str  # Special Area Messaging Encoder

    @property
    def forecast_url(self) -> str:
        return (
            f"https://tgftp.nws.noaa.gov/data/forecasts"
            f"/marine/coastal/an/{self.zone_code.lower()}.txt"
        )
```

Given the `Zone` named tuple, we can analyze the directory of marine forecast products, and create a list of `Zone` instances that starts like this:

```
ZONES = [
    Zone("Chesapeake Bay from Pooles Island to Sandy Point, MD", "ANZ531",
    "073531"),
    Zone("Chesapeake Bay from Sandy Point to North Beach, MD", "ANZ532",
    "073532"),
    Zone("Chesapeake Bay from North Beach to Drum Point, MD", "ANZ533",
    "073533"),
```

```
# etc.
```

```
    Zone(
        "Tangier Sound and the Inland Waters surrounding Bloodsworth
        Island",
        "ANZ543",
        "073543",
    ),
]
```

Depending on where you're going to be sailing, you may want additional or different zones.

We need a `MarineWX` class to describe the work to be done. This is an example of a **Command** pattern, where each instance of the class is another thing we wish to do. This class has a `run()` method to gather data from a weather service:

```
class MarineWX:
    advisory_pat = re.compile(r"\n\.\.\.(.*?)\.\.\.\n", re.M | re.S)

    def __init__(self, zone: Zone) -> None:
        super().__init__()
        self.zone = zone
        self.doc = ""

    async def run(self) -> None:
        """
        Blocking IO assigned to a task.
        with urlopen(self.zone.forecast_url) as stream:
            self.doc = stream.read().decode("UTF-8")
        """
        async with httpx.AsyncClient() as client:
            response = await client.get(self.zone.forecast_url)
        self.doc = response.text

    @property
    def advisory(self) -> str:
        if match := self.advisory_pat.search(self.doc):
            return match.group(1).replace("\n", " ")
        return ""

    def __repr__(self) -> str:
        return f"{self.zone.zone_name} {self.advisory}"
```

The `advisory_pat` value defines a regular expression to match the `...some text...` portion of the content. We've use two regular expression flags, `re.M` to do multi-line matching and `re.S` to permit the `.*?` pattern to match all characters, even the end-of-line character. This combination of `re.M | re.S` is the common way to find a big block of text.

In this example, the `run()` method downloads the text document from the weather service via an instance of the `httpx` module's `AsyncClient` class. A separate property, `advisory()`, parses the text, looking for a pattern that marks a marine weather advisory. The sections of the weather service document really are marked by three periods, a block of text, and three periods. The marine forecast system is designed to provide an easy-to-process text-centric format with a tiny document size.

So far, this isn't unique or remarkable. We've defined a repository of zone information, and a class that gathers data for a zone. Here's the important part: a `main()` function that uses the AsyncIO tasks to gather as much data as quickly as possible:

```
async def task_main() -> None:
    start = time.perf_counter()
    forecasts = [MarineWX(z) for z in ZONES]

    await asyncio.gather(
        *(f.run() for f in forecasts)
    )

    for f in forecasts:
        print(f)

    print(
        f"Got {len(forecasts)} forecasts "
        f"in {time.perf_counter() - start:.3f} seconds"
    )


if __name__ == "__main__":
    asyncio.run(task_main())
```

The `task_main()` function, when run in the `asyncio` event loop, will launch a number of tasks, each of which is executing the `MarineWX.run()` method for a different zone. The `gather()` function waits until all of them have finished to return the list of futures. Note the use of `*` to provide the list as individual positional parameters to the `gather()` function.

In this case, we don't really want the future result from the created threads; we want the state changes that have been made to all of the `MarineWX` instances. These will be a collection of `Zone` objects and the forecast details. This client runs pretty quickly — we got 13 forecasts in about 300 milliseconds.

The `httpx` project supports the decomposition of fetching the raw data and processing the data into separate coroutines. This permits waiting for data to be interleaved with processing.

We've hit most of the high points of AsyncIO in this section, and the chapter has covered many other concurrency primitives. Concurrency is a hard problem to solve, and no one solution fits all use cases. The most important part of designing a concurrent system is deciding which of the available tools is the correct one to use for the problem. We have seen the advantages and disadvantages

of several concurrency libraries, and now have some insight into which are the better choices for different types of requirements.

The next topic touches on the question of how "expressive" a concurrency framework or package can be. We'll see how `asyncio` solves a classic computer science problem with a short, clean-looking application program.

The dining philosophers benchmark

The faculty of the College of Philosophy in an old seaside resort city (on the Atlantic coast of the US) has a long-standing tradition of dining together every Sunday night. The food is catered from Mo's Deli, but is always — always — a heaping bowl of spaghetti. No one can remember why, but Mo's a great chef, and each week's spaghetti is a unique experience.

The problem in this section isn't practical. It has a hidden complication with respect to resource sharing. The idea is be able to write clear code that solves the problem.

We serve this morsel as something the reader can then customize to explore distinct ways of building this essential algorithm. For example, we suggest refactoring the various functions into methods to eliminate use of global variables.

The philosophy department is small, having five tenured faculty members. They're also impoverished and can only afford five forks. Because the dining philosophers each require two forks to enjoy their pasta, they sit around a circular table, so each philosopher has access to two nearby forks.

This requirement for two forks to eat leads to an interesting resource contention problem, shown in the following diagram:

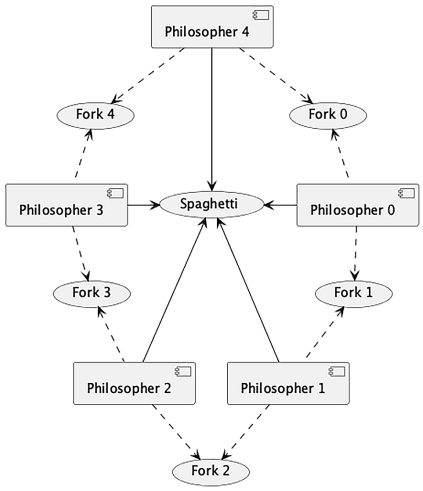


Figure 14.2: The dining philosophers

Ideally, a philosopher, say Philosopher 4, the department chairperson and an ontologist, will acquire the two closest forks, Fork 4 and Fork 0, required to eat. Once they've eaten, they release the forks so they can spend some time on philosophy.

There's a problem waiting to be solved. If each philosopher is right-handed, they will reach out, grab the fork on their right, and — unable to grab another fork — are stopped. The system is **deadlocked** because no philosopher can acquire the resources to eat.

One possible solution is to break the deadlock by using a timeout: if a philosopher can't acquire a second fork in a few seconds, they set their first fork down, wait a few seconds, and try again. If they all proceed at the same tempo, this results in a cycle of each philosopher getting one fork, waiting a few seconds, setting their forks down, and trying again. Funny, but unsatisfying.

A better solution is to permit only four philosophers at a time to sit at the table. This ensures that at least one philosopher can acquire two forks and eat. While that philosopher is philosophizing, the forks are now available to their two neighbors. Additionally, the first to finish philosophizing can leave the table, allowing the fifth to be seated and join the conversation.

How does this look in code? Here's the philosopher, defined as a coroutine:

```
import asyncio
import random

FORKS: list[asyncio.Lock]


async def philosopher(id: int, footman: asyncio.Semaphore) -> tuple[int,
float, float]:
    async with footman:
        async with FORKS[id], FORKS[(id + 1) % len(FORKS)]:
            eat_time = 1 + random.random()
            print(f"{id} eating")
            await asyncio.sleep(eat_time)
        think_time = 1 + random.random()
        print(f"{id} philosophizing")
        await asyncio.sleep(think_time)
    return id, eat_time, think_time
```

Each philosopher needs to know a few things:

- Their own unique identifier. This is their seat at the table, and directs them to the two adjacent forks they're permitted to use.
- A `Semaphore` — the footman — who seats them at the table. It's the footman's job to have an upper bound on how many can be seated, thereby avoiding a deadlock.
- A global collection of forks, represented by a sequence of `Lock` instances, that will be shared by the philosophers.

The philosophers' mealtime is described by acquiring and using resources. This is implemented with the `async with` statements. The sequence of events looks like this:

1. A philosopher acquires a seat at the table from the footman, a `Semaphore`. We can think of the footman as holding a silver tray with four "you may eat" tokens. A philosopher must have a token before they can sit. When leaving the table, a philosopher drops their token on the tray. The fifth philosopher is eagerly waiting for the token drop from the first philosopher who finishes eating.
2. A philosopher acquires the fork with their ID number and the next higher-numbered fork. The modulo operator assures that the counting of "next" wraps around to 0; `(4+1) % 5` is 0.
3. With a seat at the table and with two forks, the philosopher may enjoy their pasta. Mo often

uses Kalamata olives and pickled artichoke hearts; it's delightful. Once a month there might be some anchovies or feta cheese.

4. After eating, a philosopher releases the two fork resources. They're not done with dinner, however. Once they've set the forks down, they then spend time philosophizing about life, the universe, and everything.

5. Finally, they relinquish their seat at the table, returning their "you may eat" token to the footman, in case another philosopher is waiting for it.

Looking at the `philosopher()` function, we can see that the forks are a global resource, but the semaphore is a parameter. There's no compelling technical reason to distinguish between the global collection of `Lock` objects to represent the forks and the `Semaphore` as a parameter. We showed both to illustrate the two common choices for providing data to coroutines.

The overall dining room is organized like this:

```
async def main(faculty: int = 5, servings: int = 5) -> None:
    global FORKS
    FORKS = [asyncio.Lock() for i in range(faculty)]
    footman = asyncio.BoundedSemaphore(faculty - 1)
    for serving in range(servings):
        department = (philosopher(p, footman) for p in range(faculty))
        results = await asyncio.gather(*department)
        print(results)


if __name__ == "__main__":
    asyncio.run(main())
```

The `main()` coroutine creates the collection of forks; these are modeled as `Lock` objects that a philosopher can acquire. The footman is a `BoundedSemaphore` object with a limit one fewer than the size of the faculty; this avoids a deadlock. For each serving, the department is represented by a collection of `philosopher()` coroutines. `asyncio.gather()` waits for all of the department's coroutines to complete their work — the interleaved eating and philosophizing.

The beauty of this benchmark problem is to show how well the processing can be stated in the given programming language and library. The point is not speed, or tricky algorithms, or clever data structures. The point is to achieve some clear, expressive code for a problem that has a number of difficult constraints. With the `asyncio` package, the code is extremely elegant, and seems to be a

succinct and expressive representation of a solution to the problem.

The `concurrent.futures` library can make use of an explicit `ThreadPool`. It can approach this level of clarity but involves a little bit more technical overhead.

The `threading` and `multiprocessing` libraries can also be used directly to provide a similar implementation. Using either of these involves even more technical overhead than the `concurrent.futures` library. If the eating or philosophizing involved real computational work — not simply sleeping — we would see that a `multiprocessing` version would finish the soonest because the computation can be spread among several cores. If the eating or philosophizing was mostly waiting for I/O to complete, it would be more like the implementation shown here, and using `asyncio` or using `concurrent.futures` with a thread pool would work out nicely.

Recall

We've looked closely at a variety of topics related to concurrent processing in Python:

- Threads have an advantage of simplicity for many cases. This has to be balanced against the GIL interfering with compute-intensive multi-threading.
- Multiprocessing has an advantage of making full use of all cores of a processor. This has to be balanced against interprocess communication costs. If shared memory is used, there is the complication of encoding and accessing the shared objects.
- The `concurrent.futures` module defines an abstraction — the future — that can minimize the differences in application programming used for accessing threads or processes. This makes it easy to switch and see which approach is fastest.
- The `async`/`await` features of the Python language are supported by the AsyncIO package. Because these are coroutines, there isn't true parallel processing; control switches among the coroutines allow a single thread to interleave between waiting for I/O and computing.
- The dining philosophers benchmark can be helpful for comparing different kinds of concurrency language features and libraries. It's a relatively simple problem with some interesting complexities.
- Perhaps the most important observation is the lack of a trivial one-size-fits-all solution to concurrent processing. It's essential to create — and measure — a variety of solutions to determine a design that makes best use of the computing hardware.

Exercises

We've covered several different concurrency paradigms in this chapter and still may not have provided a clear idea of when each one is useful. This lack of clarity suggests that it's generally best to develop a few different strategies before committing to one that is measurably better than the others. The final choice must be based on measurements of the performance of multi-threaded and multiprocessing solutions.

Concurrency is a huge topic. As your first exercise, we encourage you to search the web to discover what are considered to be the latest Python concurrency best practices. It can help to investigate material that isn't Python-specific to understand operating system primitives such as semaphores, locks, and queues.

Be sure to search for articles on "free-threading in Python." These will describe Python without the GIL. This is still several years away from being available, but it is a possible future for the language.

If you have used threads in a recent application, take a look at the code and see how you can make it more readable and less bug-prone by using futures. Compare thread and multiprocessing futures to see whether you can gain anything by using multiple CPUs.

Try implementing an AsyncIO service for some basic HTTP requests. If you can get it to the point that a web browser can render a simple GET request, you'll have a good understanding of AsyncIO network transports and protocols.

Make sure you understand the race conditions that happen in threads when you access shared data. Try to come up with a program that uses multiple threads to set shared values in such a way that the data deliberately becomes corrupt or invalid.

In *Chapter 9*, we looked at an example that used `subprocess.run()` to execute a number of `python -m doctest` commands on files within a directory. Review that example and rewrite the code to run each subprocess in parallel using a `futures.ProcessPoolExecutor`.

Looking back at *Chapter 12*, there's an example that runs an external command to create the figures for each chapter. This relies on an external application, `java`, which tends to consume a lot of CPU resources when it runs. Does concurrency help with this example? Running multiple, concurrent Java programs seems to be a terrible burden. Is this a case where the default value for the size of a process pool is too large?

Summary

This chapter ends our exploration of object-oriented programming with a topic that isn't very object-oriented. Concurrency is a difficult problem, and we've only scratched the surface. While the underlying operating system abstractions of processes and threads do not provide an API that is remotely object-oriented, Python offers some really good object-oriented abstractions around them. The threading and multiprocessing packages both provide an object-oriented interface to the underlying mechanics. Futures are able to encapsulate a lot of the messy details into a single object. AsyncIO uses coroutine objects to make our code read as though it runs synchronously, while hiding ugly and complicated implementation details behind a very simple loop abstraction.

Thank you for reading *Python Object-Oriented Programming*, *Fifth Edition*. We hope you've enjoyed the ride and are eager to start implementing object-oriented software in all your future projects!

www.packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

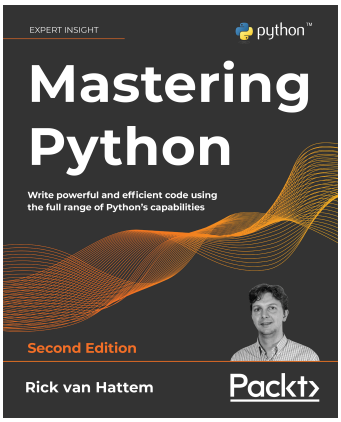
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

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

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

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

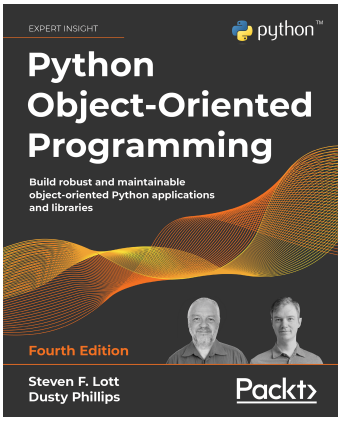


Mastering Python - Second Edition

Rick Van Hattem

ISBN: 978-1-80020-772-1

- Write beautiful Pythonic code and avoid common Python coding mistakes
- Apply the power of decorators, generators, coroutines, and metaclasses
- Use different testing systems like pytest, unittest, and doctest
- Track and optimize application performance for both memory and CPU usage
- Debug your applications with PDB, Werkzeug, and faulthandler
- Improve your performance through asyncio, multiprocessing, and distributed computing
- Explore popular libraries like Dask, NumPy, SciPy, pandas, TensorFlow, and scikit-learn
- Extend Python's capabilities with C/C++ libraries and system calls



Python Object-Oriented Programming - Fourth Edition

Steven F. Lott

Dusty Phillips

ISBN: 978-1-80107-726-2

- Implement objects in Python by creating classes and defining methods
- Extend class functionality using inheritance
- Use exceptions to handle unusual situations cleanly
- Understand when to use object-oriented features, and more importantly, when not to use them
- Discover several widely used design patterns and how they are implemented in Python
- Uncover the simplicity of unit and integration testing and understand why they are so important
- Learn to statically type check your dynamic code
- Understand concurrency with asyncio and how it speeds up programs

Share Your Thoughts

Once you've read *Python Object-Oriented Programming, Fifth Edition*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

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

Subscribe to Deep Engineering

Join thousands of developers and architects who want to understand how software is changing, deepen their expertise, and build systems that last.

Deep Engineering is a weekly expert-led newsletter for experienced practitioners, featuring original analysis, technical interviews, and curated insights on architecture, system design, and modern programming practice.

Scan the QR or visit the link to subscribe for free:
`https://packt.link/deep-engineering-newsletter`

Index

D

K

L

M

N

O

P

U

V