

EXPERT INSIGHT



# Modern Python Cookbook

130+ updated recipes for modern Python 3.12  
with new techniques and tools



**Third Edition**

**Steven F. Lott**

**<packt>**

# Modern Python Cookbook

Third Edition

130+ updated recipes for modern Python 3.12 with  
new techniques and tools

Steven F. Lott



*Packt and this book are not officially connected with Python. This book is an effort from the Python community of experts to help more developers.*

# Modern Python Cookbook

Third Edition

Copyright © 2024 Packt Publishing

*All rights reserved.* No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

**Senior Publishing Product Manager:** Denim Pinto

**Acquisition Editor – Peer Reviews:** Swaroop Singh

**Project Editor:** Parvathy Nair

**Senior Development Editor:** Elliot Dallow

**Copy Editor:** Safis Editing

**Technical Editor:** Karan Sonawane

**Proofreader:** Safis Editing

**Indexer:** Subalakshmi Govindhan

**Presentation Designer:** Ganesh Bhadwalkar

**Developer Relations Marketing Executive:** Vignesh Raju

First published: November 2016

Second edition: July 2020

Third edition: July 2024

Production reference: 1290724

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-83546-638-4

[www.packt.com](http://www.packt.com)

# Contributors

## About the author

**Steven F. Lott** has been turning coffee into software since the days when 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 books, and sometimes manages to attend conferences.

Steven is currently a tech nomad who lives in various places on the East Coast of the US. He tries to live by the words, "Don't come home until you have a story." You can find him online at <https://fosstodon.org/@slott56>.



## About the reviewers

**Abhijeet Jagutai Dada Mote** is a software engineer with expertise in data engineering, data analysis, data science, and automation. He has worked in AdTech, semiconductors, IoT, and media industries and is currently working on a cutting-edge CTV ad platform. He is proficient in Python, development of automation testing frameworks, advanced big data pipelines, and AI-driven anomaly detection systems. He has also contributed to smart city projects, holds various AI related patents, is working on several AI research papers, and delivered a workshop at PyCon Italia. Find out more at: <https://scholar.google.com/citations?user=v4tFOAgAAAAJ>

*I would like to thank my wife, Priyanka, and our one-year-old child, Advait, for their support and patience. I also thank my Aai-Baba, friends, and colleagues who have helped me become who I am today.*

**Francisco Benavides** has over 20 years of experience in programming, with the last 10 focused on Python development for ETL server applications. He has been part of worldwide teams spanning five continents, has been a team leader and specialist in the telecom sector, a Python developer, and has now been a data engineer in the energy sector for over 5 years, where he has developed a multitude of data ingestion platforms using Linux, Azure, and AWS.

*I thank God for the opportunities he has allowed me to take and a special thank you for the blessings I received from my wife and kids, who have endured, at times, long months apart, working and encouraging me to keep going. I also thank the editorial team for trusting me and allowing me to collaborate in the making of this book.*

## 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

<b>Preface</b>	<b>xv</b>
<hr/>	
<b>Chapter 1: Numbers, Strings, and Tuples</b>	<b>1</b>
<hr/>	
Choosing between float, decimal, and fraction .....	2
Choosing between true division and floor division .....	9
String parsing with regular expressions .....	13
Building complicated strings with f-strings .....	18
Building complicated strings from lists of strings .....	23
Using the Unicode characters that aren't on our keyboards .....	26
Encoding strings – creating ASCII and UTF-8 bytes .....	29
Decoding bytes – how to get proper characters from some bytes .....	33
Using tuples of items .....	36
Using NamedTuples to simplify item access in tuples .....	40
<b>Chapter 2: Statements and Syntax</b>	<b>45</b>
<hr/>	
Writing Python script and module files – syntax basics .....	47
Writing long lines of code .....	52
Including descriptions and documentation .....	57
Writing better docstrings with RST markup .....	62
Designing complex if...elif chains .....	67
Saving intermediate results with the := “walrus” operator .....	73

Avoiding a potential problem with break statements .....	77
Leveraging exception matching rules .....	81
Avoiding a potential problem with an except: clause .....	86
Concealing an exception root cause .....	88
Managing a context using the with statement .....	91

**Chapter 3: Function Definitions** **97**

---

Function parameters and type hints .....	98
Designing functions with optional parameters .....	103
Using super flexible keyword parameters .....	109
Forcing keyword-only arguments with the * separator .....	114
Defining position-only parameters with the / separator .....	119
Picking an order for parameters based on partial functions .....	122
Writing clear documentation strings with RST markup .....	129
Designing recursive functions around Python’s stack limits .....	134
Writing testable scripts with the script-library switch .....	139

**Chapter 4: Built-In Data Structures Part 1: Lists and Sets** **145**

---

Choosing a data structure .....	146
Building lists – literals, appending, and comprehensions .....	151
Slicing and dicing a list .....	157
Shrinking lists – deleting, removing, and popping .....	163
Writing list-related type hints .....	169
Reversing a copy of a list .....	173
Building sets – literals, adding, comprehensions, and operators .....	177
Shrinking sets – remove(), pop(), and difference .....	184
Writing set-related type hints .....	188

**Chapter 5: Built-In Data Structures Part 2: Dictionaries** **195**

---

Creating dictionaries – inserting and updating .....	196
--	-----

Shrinking dictionaries – the pop() method and the del statement .....	202
Writing dictionary-related type hints .....	205
Understanding variables, references, and assignment .....	211
Making shallow and deep copies of objects .....	215
Avoiding mutable default values for function parameters .....	221

---

**Chapter 6: User Inputs and Outputs** **229**

Using the features of the print() function .....	230
Using input() and getpass() for user input .....	235
Debugging with f“{value=}” strings .....	241
Using argparse to get command-line input .....	243
Using invoke to get command-line input .....	249
Using cmd to create command-line applications .....	253
Using the OS environment settings .....	257

---

**Chapter 7: Basics of Classes and Objects** **263**

Using a class to encapsulate data and processing .....	265
Essential type hints for class definitions .....	269
Designing classes with lots of processing .....	274
Using typing.NamedTuple for immutable objects .....	280
Using dataclasses for mutable objects .....	283
Using frozen dataclasses for immutable objects .....	289
Optimizing small objects with __slots__ .....	292
Using more sophisticated collections .....	297
Extending a built-in collection – a list that does statistics .....	303
Using properties for lazy attributes .....	307
Creating contexts and context managers .....	313
Managing multiple contexts with multiple resources .....	319

---

**Chapter 8: More Advanced Class Design** **327**

Choosing between inheritance and composition – the “is-a” question .....	328
Separating concerns via multiple inheritance .....	335
Leveraging Python’s duck typing .....	342
Managing global and singleton objects .....	347
Using more complex structures – maps of lists .....	353
Creating a class that has orderable objects .....	358
Deleting from a list of complicated objects .....	364

---

**Chapter 9: Functional Programming Features** **371**

Writing generator functions with the yield statement .....	373
Applying transformations to a collection .....	381
Using stacked generator expressions .....	386
Picking a subset – three ways to filter .....	394
Summarizing a collection – how to reduce .....	399
Combining the map and reduce transformations .....	405
Implementing “there exists” processing .....	412
Creating a partial function .....	417
Writing recursive generator functions with the yield from statement .....	423

---

**Chapter 10: Working with Type Matching and Annotations** **431**

Designing with type hints .....	432
Using the built-in type matching functions .....	440
Using the match statement .....	444
Handling type conversions .....	448
Implementing more strict type checks with Pydantic .....	455
Including run-time valid value checks .....	462

---

**Chapter 11: Input/Output, Physical Format, and Logical Layout** **471**

Using pathlib to work with filenames .....	474
Replacing a file while preserving the previous version .....	482

Reading delimited files with the CSV module .....	488
Using dataclasses to simplify working with CSV files .....	495
Reading complex formats using regular expressions .....	500
Reading JSON and YAML documents .....	508
Reading XML documents .....	516
Reading HTML documents .....	523
<b>Chapter 12: Graphics and Visualization with Jupyter Lab</b>	<b>533</b>
<hr/>	
Starting a Notebook and creating cells with Python code .....	535
Ingesting data into a notebook .....	540
Using pyplot to create a scatter plot .....	546
Using axes directly to create a scatter plot .....	552
Adding details to markdown cells .....	558
Including Unit Test Cases in a Notebook .....	562
<b>Chapter 13: Application Integration: Configuration</b>	<b>567</b>
<hr/>	
Finding configuration files .....	569
Using TOML for configuration files .....	575
Using Python for configuration files .....	580
Using a class as a namespace for configuration .....	585
Designing scripts for composition .....	591
Using logging for control and audit output .....	596
<b>Chapter 14: Application Integration: Combination</b>	<b>605</b>
<hr/>	
Combining two applications into one .....	606
Combining many applications using the <b>Command</b> design pattern .....	614
Managing arguments and configuration in composite applications .....	620
Wrapping and combining CLI applications .....	627
Wrapping a program and checking the output .....	632



---

**Chapter 15: Testing** **641**

Using docstrings for testing .....	643
Testing functions that raise exceptions .....	651
Handling common doctest issues .....	654
Unit testing with the unittest module .....	661
Combining unittest and doctest tests .....	667
Unit testing with the pytest module .....	672
Combining pytest and doctest tests .....	677
Testing things that involve dates or times .....	681
Testing things that involve randomness .....	686
Mocking external resources .....	693

---

**Chapter 16: Dependencies and Virtual Environments** **703**

Creating environments using the built-in venv .....	706
Installing packages with a requirements.txt file .....	711
Creating a pyproject.toml file .....	716
Using pip-tools to manage the requirements.txt file .....	723
Using Anaconda and the conda tool .....	727
Using the poetry tool .....	732
Coping with changes in dependencies .....	736

---

**Chapter 17: Documentation and Style** **743**

The bare minimum: a README.rst file .....	744
Installing Sphinx and creating documentation .....	749
Using Sphinx autodoc to create the API reference .....	754
Identifying other CI/CD tools in pyproject.toml .....	760
Using tox to run comprehensive quality checks .....	764

---

**Other Books You May Enjoy** **773**

---





# Preface

Python is the preferred language choice of developers, engineers, data scientists, and hobbyists everywhere. It is a great scripting language that can power your applications and provide speed, safety, and scalability. By exposing Python as a series of simple recipes, this book can help you gain insights into specific language features in a concrete context. The idea is to avoid abstract discussions of language features and focus on applying the language to concrete data and processing problems.

## What you need for this book

All you need to follow through the examples in this book is a computer running any Python, version 3.12 or newer. Many of the examples can be adapted to work with Python 3 versions prior to 3.12. Material in Chapter 10 describes the `match` statement, introduced with Python 3.10.

We strongly encourage installing a fresh copy of Python, avoiding any pre-installed operating system Python. The language run-time can be downloaded from <https://www.python.org/downloads/>. An alternative is to start with the Miniconda tool (<https://docs.conda.io/en/latest/miniconda.html>) and use **conda** to create a Python 3.12 (or newer) environment.



Python 2 cannot be used anymore. Since 2020, Python 2 is no longer an alternative.

## Who this book is for

The book is for web developers, programmers, enterprise programmers, engineers, and big data scientists. If you are a beginner, this book can get you started. If you are experienced, it will expand your knowledge base. A basic knowledge of programming will help; while some foundational topics are covered, this is not a tutorial on programming or Python.

## What this book covers

There are over 130 recipes in this book. We can decompose them into four general areas:

- **Python Fundamentals**

*Chapter 1, Numbers, Strings, and Tuples*, will look at the different kinds of numbers, how to work with strings, how to use tuples, and how to use the essential built-in types in Python. We will also show ways to exploit the full power of the Unicode character set.

*Chapter 2, Statements and Syntax*, will cover some basics of creating script files. Then we'll move on to looking at some of the complex statements, including `if`, `while`, `for`, `break`, `try`, `raise`, and `with`.

*Chapter 3, Function Definitions*, will look at a number of function definition techniques. We'll devote several recipes to type hints for a variety of types. We'll also address an element of designing a testable script by using functions and a `main-import-switch`.

*Chapter 4, Built-In Data Structures Part 1: Lists and Sets*, begins an overview of the built-in data structures structures that are available and what problems they solve. This includes a number of recipes showing list and set operations, including list and set comprehensions.

*Chapter 5, Built-In Data Structures Part 2: Dictionaries*, continues examining the built-in data structures, looking at dictionaries in detail. This chapter will also look at some more advanced topics related to how Python handles references to objects. It also shows how to handle mutable objects as function parameter default values.

*Chapter 6, User Inputs and Outputs*, explains how to use the different features of the `print()` function. We'll also look at the different functions used to provide user input. The use of f-strings for debugging and the `argparse` module for command-line input are featured.

- **Object-Oriented and Functional Design Approaches**

*Chapter 7, Basics of Classes and Objects*, begins the coverage of object-oriented programming. It shows how to create classes and the type hints related to class definitions. This section has been expanded from previous editions to cover data-classes. It shows how to extend built-in classes, and how to create context managers to manage resources.

*Chapter 8, More Advanced Class Design*, continues the exploration of object-oriented design and programming. This includes an exploration of the composition vs. inheritance question, and shows how to manage the “duck typing” principle of Python.

*Chapter 9, Functional Programming Features*, looks at Python's functional programming features. This style of programming emphasizes function definitions and stateless, immutable objects. The recipes look at generator expressions, using the `map()`, `filter()`, and `reduce()` functions. We also look at ways to create partial functions and some examples of replacing stateful objects with data structures built from collections of immutable objects.

- **More Sophisticated Designs**

*Chapter 10, Working with Type Matching and Annotations*, looks more closely at type hints and the `match` statement. This includes using **Pydantic** to create classes with more strict run-time type-checking. It also looks at introspection of annotated types.

*Chapter 11, Input/Output, Physical Format, and Logical Layout*, will work with paths and files in general. It will look at reading and writing data in a variety of file formats, including CSV, JSON (and YAML), XML, and HTML. The HTML section will emphasize using **Beautiful Soup** for extracting data.

*Chapter 12, Graphics and Visualization with Jupyter Lab*, will use **Jupyter Lab** to create notebooks that use Python for data analysis and visualization. This will show ways to ingest data into a notebook to create plots, and how to use Markdown to create useful documentation and reports from a notebook.

*Chapter 13, Application Integration: Configuration*, will start looking at ways that we can design larger applications. The recipes in this chapter address different ways to handle configuration files and how to manage logging.

*Chapter 14, Application Integration: Combination*, will continue looking at ways to create composite applications from smaller pieces. This will look at object-oriented design patterns and **Command-Line Interface (CLI)** applications. It will also look at using the subprocess module to run existing applications under Python's control.

- **Completing a Project: Fit and Finish**

*Chapter 15, Testing*, provides recipes for using the built-in doctest and unittest testing frameworks used in Python. Additionally, recipes will cover the **pytest** tool.

*Chapter 16, Dependencies and Virtual Environments*, covers tools used to manage virtual environments. The built-in venv, as well as **conda** and **poetry** will be covered. There are a lot of solutions to managing virtual environments, and we can't cover all of them.

*Chapter 17, Documentation and Style*, covers additional tools that can help to create high-quality software. This includes a particular focus on **sphinx** for creating comprehensive, readable documentation. We'll also look at **tox** to automate running tests.

## To get the most out of this book

To get the most out of this book you can download the example code files and the color images as per the instructions below.

## Download the example code files

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Modern-Python-Cookbook-Third-Edition>. This repository is also the best place to start a conversation about specific topics discussed in the book. Feel free to open an issue if you want to engage with the authors or other readers. 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://packt.link/gbp/9781835466384>.

## 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, user input, and Twitter handles are shown as follows: “We can include other contexts through the use of the `include` directive.”

A block of code is set as follows:

```
if distance is None:
    distance = rate * time
elif rate is None:
    rate = distance / time
elif time is None:
    time = distance / rate
```

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

```
>>> import math
>>> math.factorial(52)
80658175170943878571660636856403766975289505440883277824000000000000
```



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](mailto: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](mailto: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://packtpub.com/support/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](mailto: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 *Modern Python Cookbook, Third Edition*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.



<https://packt.link/r/1835466389>

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

## Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781835466384>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

# 1

## Numbers, Strings, and Tuples

This chapter will look at some of the central types of Python objects. We'll look at working with different kinds of numbers, working with strings, and using tuples. These are the simplest kinds of data that Python works with. In later chapters, we'll look at data structures built on these foundations.

While these recipes start with a beginner's level of understanding of Python 3.12, they also provide some deeper background for those familiar with the language. In particular, we'll look at some details of how numbers are represented internally, because this can help when confronted with more advanced numerical programming problems. This will help us distinguish the uses cases for the rich variety of numeric types.

We'll also look at the two different division operators. These have distinct use cases, and we'll look at one kind of algorithm that demands truncated division.

When working with strings, there are several common operations that are important. We'll

explore some of the differences between bytes—as used by our OS files—and strings used to represent Unicode text. We’ll look at how we can exploit the full power of the Unicode character set.

In this chapter, we’ll show the recipes as if we’re working from the `>>>` prompt in interactive Python. This is the prompt that’s provided when running python from the command line or using the Python console in many **Integrated Development Environment (IDE)** tools. This is sometimes called the **read-evaluate-print loop (REPL)**. In later chapters, we’ll change the style so it looks more like a script file. One goal of this chapter is to encourage interactive exploration because it’s a great way to learn the language.

We’ll cover these recipes to introduce basic Python data types:

- *Choosing between float, decimal, and fraction*
- *Choosing between true division and floor division*
- *String parsing with regular expressions*
- *Building complicated strings with f-strings*
- *Building complicated strings from lists of strings*
- *Using the Unicode characters that aren’t on our keyboards*
- *Encoding strings – creating ASCII and UTF-8 bytes*
- *Decoding bytes – how to get proper characters from some bytes*
- *Using tuples of items*
- *Using NamedTuples to simplify item access in tuples*

We’ll start with numbers, work our way through strings, and end up working with simple combinations of objects in the form of tuples and NamedTuple objects.

## Choosing between float, decimal, and fraction

Python offers several ways to work with rational numbers and approximations of irrational numbers. We have three basic choices:

- Float
- Decimal
- Fraction

When we have choices, it helps to have some criteria for making a selection.

## Getting ready

There are three general cases for expressions that involve numbers beyond integers, which are:

1. **Currency:** Dollars, cents, euros, and so on. Currency generally has a fixed number of decimal places and rounding rules to properly quantize results.
2. **Rational Numbers or Fractions:** When we scale a recipe that serves eight, for example, down to five people, we're doing fractional math using a scaling factor of  $\frac{5}{8}$ .
3. **Floating Point:** This includes all other kinds of calculations. This also includes irrational numbers, like  $\pi$ , root extraction, and logarithms.

When we have one of the first two cases, we should avoid floating-point numbers.

## How to do it...

We'll look at each of the three cases separately.

### Doing currency calculations

When working with currency, we should always use the `decimal` module. If we try to use the values of Python's built-in `float` type, we can run into problems with the rounding and truncation of numbers:

1. To work with currency, import the `Decimal` class from the `decimal` module:

```
>>> from decimal import Decimal
```

2. We need to create `Decimal` objects from strings or integers. In this case, we want

7.25%, which is  $\frac{7.25}{100}$ . We can compute the value using Decimal objects:

```
>>> tax_rate = Decimal('7.25')/Decimal(100)
>>> purchase_amount = Decimal('2.95')
>>> tax_rate * purchase_amount
Decimal('0.213875')
```

We could also use `Decimal('0.0725')` instead of doing the division explicitly.

3. To round to the nearest penny, create a penny object:

```
>>> penny = Decimal('0.01')
```

4. Quantize the result using the penny object:

```
>>> total_amount = purchase_amount + tax_rate * purchase_amount
>>> total_amount.quantize(penny)
Decimal('3.16')
```

This uses the default rounding rule of `ROUND_HALF_EVEN`. The `Decimal` module offers other rounding variations. We might, for example, do something like this:

```
>>> import decimal
>>> total_amount.quantize(penny, decimal.ROUND_UP)
Decimal('3.17')
```

This shows the consequences of using a different rounding rule.

## Fraction calculations

When we're doing calculations that have exact fraction values, we can use the `fractions` module to create rational numbers. In this example, we want to scale a recipe for eight down to five people, using  $\frac{5}{8}$  of each ingredient. When the recipe calls for  $2\frac{1}{2}$  cups of rice, what does that scale down to?

To work with fractions, we'll do this:

1. Import the Fraction class from the fractions module:

```
>>> from fractions import Fraction
```

2. Create Fraction objects from strings, integers, or pairs of integers. We created one fraction from a string, '2.5'. We created the second fraction from a floating-point expression, 5 / 8. This **only** works when the denominator is a power of 2:

```
>>> sugar_cups = Fraction('2.5')
>>> scale_factor = Fraction(5/8)
>>> sugar_cups * scale_factor
Fraction(25, 16)
```

We can see that we'll use almost a cup and a half of rice to scale the recipe for five people instead of eight. While float values will often be useful for rational fractions, they may not be exact unless the denominator is a power of two.

## Floating-point approximations

Python's built-in float type can represent a wide variety of values. The trade-off here is that a float value is often an approximation. There may be a small discrepancy that reveals the differences between the implementation of float and the mathematical ideal of an irrational number:

1. To work with float, we often need to round values to make them look sensible. It's important to recognize that all float calculations are an approximation:

```
>>> (19/155)*(155/19)
0.9999999999999999
```

2. Mathematically, the value should be 1. Because of the approximations used, the computed result isn't exactly 1. We can use `round(answer, 3)` to round to three digits, creating a value that's more useful:



```
>>> answer = (19/155)*(155/19)
>>> round(answer, 3)
1.0
```

Approximations have a very important consequence.



Don't compare floating-point values for exact equality.

Code that uses an exact `==` test between floating-point numbers has the potential to cause problems when two approximations differ by a single bit.

The float approximation rules come from the IEEE, and are not a unique feature of Python. Numerous programming languages work with float approximations and have identical behavior.

## How it works...

For these numeric types, Python offers a variety of operators: `+`, `-`, `*`, `/`, `//`, `%`, and `**`. These are for addition, subtraction, multiplication, true division, truncated division, modulo, and raising to a power, respectively. We'll look at the two division operators, `/` and `//`, in the *Choosing between true division and floor division* recipe.

Python will do some conversions between the various numeric types. We can mix `int` and `float` values; the integers will be promoted to floating-point to provide the most accurate answer possible. Similarly, we can mix `int` and `Fraction` as well as mixing `int` and `Decimal`. Note that we cannot casually mix `Decimal` with `float` or `Fraction`; an explicit conversion function will be required.

It's important to note that float values are approximations. The Python syntax allows us to write floating-point values using base 10 digits; however, that's not how values are represented internally.

We can write the value  $8.066 \times 10^{67}$  like this in Python:

```
>>> 8.066e+67
8.066e+67
```

The actual value used internally will involve a binary approximation of the decimal value we wrote. The internal value for this example is this:

```
>>> (6737037547376141 / (2**53)) * (2**226)
8.066e+67
```

The numerator is a big number, 6737037547376141. The denominator is always  $2^{53}$ . This is why values can get truncated.

We can use the `math.frexp()` function to see these internal details of a number:

```
>>> import math
>>> math.frexp(8.066E+67)
(0.7479614202861186, 226)
```

The two parts are called the **mantissa** (or **significand**) and the **exponent**. If we multiply the mantissa by  $2^{53}$ , we always get a whole number, which is the numerator of the binary fraction.

Unlike the built-in float, a `Fraction` is an exact ratio of two integer values. We can create ratios that involve integers with a very large number of digits. We're not limited by a fixed denominator.

A `Decimal` value, similarly, is based on a very large integer value, as well as a scaling factor to determine where the decimal place goes. These numbers can be huge and won't suffer from peculiar representation issues.

## There's more...

The Python `math` module contains several specialized functions for working with floating-point values. This module includes common elementary functions such as square root,

logarithms, and various trigonometry functions. It also has some other functions such as gamma, factorial, and the Gaussian error function.

The `math` module includes several functions that can help us do more accurate floating-point calculations. For example, the `math.fsum()` function will compute a floating-point sum more carefully than the built-in `sum()` function. It's less susceptible to approximation issues.

We can also make use of the `math.isclose()` function to compare two floating-point values, an expression, and a literal `1.0`, to see if they're nearly equal:

```
>>> (19/155)*(155/19) == 1.0
False

>>> math.isclose((19/155)*(155/19), 1.0)
True
```

This function provides us with a way to compare two floating-point numbers meaningfully for near-equality.

Python also offers *complex* numbers. A complex number has a real and an imaginary part. In Python, we write `3.14+2.78j` to represent the complex number  $3.14 + 2.78\sqrt{-1}$ . Python will comfortably convert between float and complex. We have the usual group of operators available for complex numbers.

To support complex numbers, there's the `cmath` package. The `cmath.sqrt()` function, for example, will return a complex value rather than raise an exception when extracting the square root of a negative number. Here's an example:

```
>>> math.sqrt(-2)
Traceback (most recent call last):
...
ValueError: math domain error

>>> import cmath
>>> cmath.sqrt(-2)
```

```
1.4142135623730951j
```

This module is helpful when working with complex numbers.

## See also

- We'll talk more about floating-point numbers and fractions in the *Choosing between true division and floor division* recipe.
- See [https://en.wikipedia.org/wiki/IEEE\\_floating\\_point](https://en.wikipedia.org/wiki/IEEE_floating_point).

## Choosing between true division and floor division

Python offers us two kinds of division operators. What are they, and how do we know which one to use? We'll also look at the Python division rules and how they apply to integer values.

## Getting ready

There are several general cases for division:

- A *div-mod* pair: We want both parts – the quotient and the remainder. The name refers to the division and modulo operations combined together. We can summarize the quotient and remainder as  $q, r = (\lfloor \frac{a}{b} \rfloor, a \bmod b)$ .

We often use this when converting values from one base into another. When we convert seconds into hours, minutes, and seconds, we'll be doing a *div-mod* kind of division. We don't want the exact number of hours; we want a truncated number of hours, and the remainder will be converted into minutes and seconds.

- The *true* value: This is a typical floating-point value; it will be a good approximation to the quotient. For example, if we're computing an average of several measurements, we usually expect the result to be floating-point, even if the input values are all integers.
- A *rational fraction* value: This is often necessary when working in American units of

feet, inches, and cups. For this, we should be using the `Fraction` class. When we divide `Fraction` objects, we always get exact answers.

We need to decide which of these cases apply, so we know which division operator to use.

## How to do it...

We'll look at these three cases separately.

### Doing floor division

When we are doing the *div-mod* kind of calculations, we might use the floor division operator, `//`, and the modulo operator, `%`. The expression `a % b` gives us the remainder from an integer division of `a // b`. Or, we might use the `divmod()` built-in function to compute both at once:

1. We'll divide the number of seconds by 3,600 to get the value of hours. The modulo, or remainder in division, computed with the `%` operator, can be converted separately into minutes and seconds:

```
>>> total_seconds = 7385
>>> hours = total_seconds // 3600
>>> remaining_seconds = total_seconds % 3600
```

2. Next, we'll divide the number of seconds by 60 to get minutes; the remainder is the number of seconds less than 60:

```
>>> minutes = remaining_seconds // 60
>>> seconds = remaining_seconds % 60
>>> hours, minutes, seconds
(2, 3, 5)
```

Here's the alternative, using the `divmod()` function to compute quotient and modulo together:

1. Compute quotient and remainder at the same time:

```
>>> total_seconds = 7385
>>> hours, remaining_seconds = divmod(total_seconds, 3600)
```

2. Compute quotient and remainder again:

```
>>> minutes, seconds = divmod(remaining_seconds, 60)
>>> hours, minutes, seconds
(2, 3, 5)
```

## Doing true division

Performing a true division calculation gives a floating-point approximation as the result. For example, about how many hours is 7,385 seconds? Here's  $\frac{7385}{60}$  using the true division operator:

```
>>> total_seconds = 7385
>>> hours = total_seconds / 3600
>>> round(hours, 4)
2.0514
```

We provided two integer values, but got a floating-point exact result. Consistent with our previous recipe, when using floating-point values, we rounded the result to avoid having to look at tiny error digits.

## Rational fraction calculations

We can do division using Fraction objects and integers. This forces the result to be a mathematically exact rational number:

1. Create at least one Fraction value:

```
>>> from fractions import Fraction
>>> total_seconds = Fraction(7385)
```

2. Use the Fraction value in a calculation. Any integer will be promoted to a Fraction:

```
>>> hours = total_seconds / 3600
>>> hours
Fraction(1477, 720)
```

The denominator of 720 doesn't seem too meaningful. Working with fractions like this requires a bit of finesse to find useful denominators that makes sense to people. Otherwise, converting to a floating-point value can be useful.

3. If necessary, convert the exact `Fraction` into a floating-point approximation:

```
>>> round(float(hours), 4)
2.0514
```

First, we created a `Fraction` object for the total number of seconds. When we do arithmetic on fractions, Python will promote any integers to `Fraction` objects; this promotion means that the math is done as precisely as possible.

## How it works...

Python has two division operators:

- The `/` true division operator produces a true, floating-point result. It does this even when the two operands are integers. This is an unusual operator in this respect. All other operators preserve the type of the data. The true division operation – when applied to integers – produces a float result.
- The `//` truncated division operator always produces a truncated result. For two integer operands, this is the truncated quotient. When floating-point operands are used, this is a truncated floating-point result:

```
>>> 7358.0 // 3600.0
2.0
```

## See also

- For more on the choice between floating-point and fractions, see the *Choosing between float, decimal, and fraction* recipe.
- See PEP-238.

## String parsing with regular expressions

How do we decompose a complex string? What if we have complex, tricky punctuation? Or—worse yet—what if we don't have punctuation, but have to rely on patterns of digits to locate meaningful information?

### Getting ready

The easiest way to decompose a complex string is by generalizing the string into a pattern and then writing a regular expression that describes that pattern.

There are limits to the patterns that regular expressions can describe. When we're confronted with deeply nested documents in a language like HTML, XML, or JSON, we often run into problems and be prohibited from using regular expressions.

The `re` module contains all of the various classes and functions we need to create and use regular expressions.

Let's say that we want to decompose text from a recipe website. Each line looks like this:

```
>>> ingredient = "Kumquat: 2 cups"
```

We want to separate the ingredient from the measurements.

### How to do it...

To write and use regular expressions, we often do this:

1. Generalize the example. In our case, we have something that we can generalize as:

```
(ingredient words): (amount digits) (unit words)
```



2. We've replaced literal text with a two-part summary: what it means and how it's represented. For example, *ingredient* is represented as words, while *amount* is represented as digits. Import the `re` module:

```
>>> import re
```

3. Rewrite the pattern into **regular expression (RE)** notation:

```
>>> pattern_text = r'([\w\s]+):\s+(\d+)\s+(\w+)'
```

We've replaced representation hints such as *ingredient words*, a mixture of letters and spaces, with `[\w\s]+`. We've replaced *amount digits* with `\d+`. And we've replaced *single spaces* with `\s+` to allow one or more spaces to be used as punctuation. We've left the colon in place because, in regular expression notation, a colon matches itself.

For each of the fields of data, we've used `()` to capture the data matching the pattern. We didn't capture the colon or the spaces because we don't need the punctuation characters.

REs typically use a lot of `\` characters. To make this work out nicely in Python, we almost always use *raw strings*. The `r'` tells Python not to look at the `\` characters and not to replace them with special characters that aren't on our keyboards.

4. Compile the pattern:

```
>>> pattern = re.compile(pattern_text)
```

5. Match the pattern against the input text. If the input matches the pattern, we'll get a match object that shows details of the substring that matched:

```
>>> match = pattern.match(ingredient)
>>> match is None
False
>>> match.groups()
('Kumquat', '2', 'cups')
```

6. Extract the named groups of characters from the match object:

```
>>> match.group(1)
'Kumquat'
>>> match.group(2)
'2'
>>> match.group(3)
'cups'
```

Each group is identified by the order of the capture ( ) portions of the regular expression. This gives us a tuple of the different fields captured from the string. We'll return to the use of the tuple data structure in the *Using tuples of items* recipe. This can be confusing in more complex regular expressions; there is a way to provide a name, instead of the numeric position, to identify a capture group.

## How it works...

There are a lot of different kinds of string patterns that we can describe with regular expressions.

We've shown a number of character classes:

- `\w` matches any alphanumeric character (a to z, A to Z, 0 to 9).
- `\d` matches any decimal digit.
- `\s` matches any space or tab character.

These classes also have inverses:

- `\W` matches any character that's not a letter or a digit.
- `\D` matches any character that's not a digit.
- `\S` matches any character that's not some kind of space or tab.

Many characters match themselves. Some characters, however, have a special meaning, and we have to use `\` to escape from that special meaning:

- We saw that `+` as a suffix means to match one or more of the preceding patterns. `\d+`

matches one or more digits. To match an ordinary +, we need to use \+.

- We also have \* as a suffix, which matches zero or more of the preceding patterns. \w\* matches zero or more characters. To match a \*, we need to use \\*.
- We have ? as a suffix, which matches zero or one of the preceding expressions. This character is used in other places, and has a different meaning in the other context. We'll see it used in ?P<name>...)|, where it is inside \verb|)| to define special properties for the grouping.
- The . character matches any single character. To match a . specifically, we need to use \..

We can create our own unique sets of characters using [] to enclose the elements of the set. We might have something like this:

```
(?P<name>\w+)\s* [=:] \s* (?P<value>.*)
```

This has a \w+ to match any number of alphanumeric characters. This will be collected into a group called name. It uses \s\* to match an optional sequence of spaces. It matches any character in the set [=:]. Exactly one of the two characters in this set must be present. It uses \s\* again to match an optional sequence of spaces. Finally, it uses .\* to match everything else in the string. This is collected into a group named value.

We can use this to parse strings, like this:

```
size = 12
weight: 14
```

By being flexible with the punctuation, we can make a program easier to use. We'll tolerate any number of spaces, and either an = or a : as a separator.

## There's more...

A long regular expression can be awkward to read. We have a clever Pythonic trick for presenting an expression in a way that's much easier to read:

```
>>> ingredient_pattern = re.compile(
... r'(?P<ingredient>[\w\s]+):\s+' # name of the ingredient up to the ":"
... r'(?P<amount>\d+)\s+' # amount, all digits up to a space
... r'(?P<unit>\w+)' # units, alphanumeric characters
... )
```

This leverages three syntax rules:

- A statement isn't finished until the ( ) characters match.
- Adjacent string literals are silently concatenated into a single long string.
- Anything between # and the end of the line is a comment, and is ignored.

We've put Python comments after the important clauses in our regular expression. This can help us understand what we did, and perhaps help us diagnose problems later.

We can also use the regular expression's "verbose" mode to add gratuitous whitespace and comments inside a regular expression string. To do this, we must use `re.X` as an option when compiling a regular expression to make whitespace and comments possible. This revised syntax looks like this:

```
>>> ingredient_pattern_x = re.compile(r'''
... (?P<ingredient>[\w\s]+):\s+ # name of the ingredient up to the ":"
... (?P<amount>\d+)\s+ # amount, all digits up to a space
... (?P<unit>\w+) # units, alphanumeric characters
... ''', re.X)
```

We can either break the pattern up into separate string components, or make use of extended syntax to make the regular expression more readable. The benefit of providing names shows up when we use the `groupdict()` method of the match object to extract parsed values by the name associated with the pattern being captured.

## See also

- The *Decoding bytes – how to get proper characters from some bytes* recipe.
- There are many books on regular expressions and Python regular expressions in

particular, like *Mastering Python Regular Expressions* <https://www.packtpub.com/application-development/mastering-python-regular-expressions>.

## Building complicated strings with f-strings

Creating complex strings is, in many ways, the polar opposite of parsing a complex string. We generally use a template with substitution rules to put data into a more complex format.

### Getting ready

Let's say we have pieces of data that we need to turn into a nicely formatted message. We might have data that includes the following:

```
>>> id = "IAD"
>>> location = "Dulles Intl Airport"
>>> max_temp = 32
>>> min_temp = 13
>>> precipitation = 0.4
```

And we'd like a line that looks like this:

```
IAD : Dulles Intl Airport : 32 / 13 / 0.40
```

### How to do it...

1. Create an f-string for the result, replacing all of the data items with placeholders. Inside each placeholder, put a variable name (or an expression.) Note that the string uses the prefix of `f`. This prefix creates a sophisticated string object where values are interpolated into the template when the string is used:

```
f'{id} : {location} : {max_temp} / {min_temp} / {precipitation}'
```

2. For each name or expression, an optional data type can be appended to the names in the template string. The basic data type codes are:
  - `s` for string

- d for decimal number
- f for floating-point number

It would look like this:

```
f'{id:s} : {location:s} : {max_temp:d} / {min_temp:d} /
{precipitation:f}'
```



Because the book's margins are narrow, the string has been broken to fit on the page. It's a single (very wide) line of code.

3. Add length information where required. Length is not always required, and in some cases, it's not even desirable. In this example, though, the length information ensures that each message has a consistent format. For strings and decimal numbers, prefix the format with the length like this: 19s or 3d. For floating-point numbers, use a two-part prefix like 5.2f to specify the total length of five characters, with two to the right of the decimal point. Here's the whole format:

```
>>> f'{id:3s} : {location:19s} : {max_temp:3d} / {min_temp:3d} /
{precipitation:5.2f}'
'IAD : Dulles Intl Airport : 32 / 13 / 0.40'
```

## How it works...

F-strings can do a lot of relatively sophisticated string assembly by interpolating data into a template. There are a number of conversions available.

We've seen three of the formatting conversions—s, d, f—but there are many others. Details can be found in the *Formatted string literals* section of the *Python Standard Library*: [https://docs.python.org/3/reference/lexical\\_analysis.html#formatted-string-literals](https://docs.python.org/3/reference/lexical_analysis.html#formatted-string-literals).

Here are some of the format conversions we might use:

- b is for binary, base 2.

- `c` is for Unicode character. The value must be a number, which is converted into a character. Often, we use hexadecimal numbers for these characters, so you might want to try values such as `0x2661` through `0x2666` to see interesting Unicode glyphs.
- `d` is for decimal numbers.
- `E` and `e` are for scientific notations. `6.626E-34` or `6.626e-34`, depending on which `E` or `e` character is used.
- `F` and `f` are for floating-point. For *not a number*, the `f` format shows lowercase `nan`; the `F` format shows uppercase `NAN`.
- `G` and `g` are for general use. This switches automatically between `E` and `F` (or `e` and `f`) to keep the output in the given sized field. For a format of `20.5G`, up to 20-digit numbers will be displayed using `F` formatting. Larger numbers will use `E` formatting.
- `n` is for locale-specific decimal numbers. This will insert `,` or `.` characters, depending on the current locale settings. The default locale may not have 1,000 separators defined. For more information, see the locale module.
- `o` is for octal, base 8.
- `s` is for string.
- `X` and `x` are for hexadecimal, base 16. The digits include uppercase `A-F` and lowercase `a-f`, depending on which `X` or `x` format character is used.
- `%` is for percentage. The number is multiplied by 100 and the output includes a `%` character.

We have a number of prefixes we can use for these different types. The most common one is the length. We might use `{name:5d}` to put in a 5-digit number. There are several prefixes for the preceding types:

- **Fill and alignment:** We can specify a specific filler character (space is the default) and an alignment. Numbers are generally aligned to the right and strings to the left. We can change that using `<`, `>`, or `^`. This forces left alignment, right alignment, or

centering, respectively. There's a peculiar = alignment that's used to put padding after a leading sign.

- **Sign:** The default rule is a leading negative sign where needed. We can use + to put a sign on all numbers, - to put a sign only on negative numbers, and a space to use a space instead of a plus for positive numbers. In scientific output, we often use `{value:5.3f}`. The space makes sure that room is left for the sign, ensuring that all the decimal points line up nicely.
- **Alternate form:** We can use the # to get an alternate form. We might have something like `{0:#x}`, `{0:#o}`, or `{0:#b}` to get a prefix on hexadecimal, octal, or binary values. With a prefix, the numbers will look like `0xnnn`, `0onnn`, or `0bnnn`. The default is to omit the two-character prefix.
- **Leading zero:** We can include 0 to get leading zeros to fill in the front of a number. Something like `{code:08x}` will produce a hexadecimal value with leading zeroes to pad it out to eight characters.
- **Width and precision:** For integer values and strings, we only provide the width. For floating-point values, we often provide `width.precision`.

There are some times when we won't use a `{name:format}` specification. Sometimes, we'll need to use a `{name!conversion}` specification. There are only three conversions available:

- `{name!r}` shows the representation that would be produced by `repr(name)`.
- `{name!s}` shows the string value that would be produced by `str(name)`; this is the default behavior if you don't specify any conversion. Using `!s` explicitly lets you add string-type format specifiers.
- `{name!a}` shows the ASCII value that would be produced by `ascii(name)`.
- Additionally, there's a handy debugging format specifier available. We can include a trailing equals sign, =, to get a handy dump of a variable or expression. The following example uses both forms:



```
>>> value = 2**12-1
>>> f'{value=} {2**7+1=}'
'value=4095 2**7+1=129'
```

The f-string showed the value of the variable named `value` and the result of an expression, `2**7+1`.

In *Chapter 7*, we'll leverage the idea of the `{name!r}` format specification to simplify displaying information about related objects.

## There's more...

The f-string processing relies on the string `format()` method. We can leverage this method and the related `format_map()` method for cases where we have more complex data structures.

Looking forward to *Chapter 5*, we might have a dictionary where the keys are simple strings that fit with the `format_map()` rules:

```
>>> data = dict(
...     id=id, location=location, max_temp=max_temp,
...     min_temp=min_temp, precipitation=precipitation
... )
>>> '{id:3s} : {location:19s} : {max_temp:3d} / {min_temp:3d} /
{precipitation:5.2f}'.format_map(data)
'IAD : Dulles Intl Airport : 32 / 13 / 0.40'
```

We've created a dictionary object, `data`, that contains a number of values with keys that are valid Python identifiers: `id`, `location`, `max_temp`, `min_temp`, and `precipitation`. We can then use this dictionary with the `format_map()` method to extract values from the dictionary using the keys.

Note that the formatting template here is *not* an f-string. It doesn't have the `f` prefix. Instead of using the automatic formatting features of an f-string, we've done the interpolation "the hard way" using the `format_map()` method of an f-string.

## See also

- More details can be found in the *Formatted string literals* section of the *Python Standard Library*: [https://docs.python.org/3/reference/lexical\\_analysis.html#formatted-string-literals](https://docs.python.org/3/reference/lexical_analysis.html#formatted-string-literals).

## Building complicated strings from lists of strings

How can we make complicated changes to an immutable string? Can we assemble a string from individual characters?

In most cases, the recipes we've already seen give us a number of tools for creating and modifying strings. There are yet more ways in which we can tackle the string manipulation problem. In this recipe, we'll look at using a `list` object as a way to decompose and rebuild a string. This will dovetail with some of the recipes in *Chapter 4*.

## Getting ready

Here's a string that we'd like to rearrange:

```
>>> title = "Recipe 5: Rewriting an Immutable String"
```

We'd like to do two transformations:

- Remove the part before `:`.
- Replace the punctuation with `_` and make all the characters lowercase.

We'll make use of the `string` module:

```
>>> from string import whitespace, punctuation
```

This has two important constants:

- `string.whitespace` lists all of the whitespace characters that are also part of ASCII, including space and tab.

- `string.punctuation` lists punctuation marks that are also part of ASCII. Unicode has a large domain of punctuation marks. This is a widely used subset.

## How to do it...

We can work with a string exploded into a list. We'll look at lists in more depth in *Chapter 4*:

1. Explode the string into a list object:

```
>>> title_list = list(title)
```

2. Find the partition character. The `index()` method for a list has the same semantics as the `index()` method has for a string. It locates the position with the given value:

```
>>> colon_position = title_list.index(':')
```

3. Delete the characters that are no longer needed. The `del` statement can remove items from a list. Unlike strings, lists are mutable data structures:

```
>>> del title_list[:colon_position+1]
```

4. Replace punctuation by stepping through each position. In this case, we'll use a `for` statement to visit every index in the string:

```
>>> for position in range(len(title_list)):
...     if title_list[position] in whitespace+punctuation:
...         title_list[position]= '_'
```

5. The expression `range(len(title_list))` generates all of the values between 0 and `len(title_list)-1`. This assures us that the value of `position` will be each value index in the list. Join the list of characters to create a new string. It seems a little odd to use a zero-length string, `' '`, as a separator when concatenating strings together. However, it works perfectly:

```
>>> title = ''.join(title_list)
>>> title
'_Rewriting_an_Immutable_String'
```

We assigned the resulting string back to the original variable. The original string object, which had been referred to by that variable, is no longer needed: it's automatically removed from memory (this is known as **garbage collection**). The new string object replaces the value of the variable.

## How it works...

This is a change in representation trick. Since a string is immutable, we can't update it. We can, however, convert it into a mutable form; in this case, a list. We can make whatever changes are required to the mutable list object. When we're done, we can change the representation from a list back to a string and replace the original value of the variable.

Lists provide some features that strings don't have. Conversely, strings provide a number of features lists don't have. As an example, we can't convert a list into lowercase the way we can convert a string.

There's an important trade-off here:

- Strings are immutable, which makes them very fast. Strings are focused on Unicode characters. When we look at mappings and sets, we can use strings as keys for mappings and items in sets because the value is immutable.
- Lists are mutable. Operations are slower. Lists can hold any kind of item. We can't use a list as a key for a mapping or an item in a set because the list value could change.

Strings and lists are both specialized kinds of sequences. Consequently, they have a number of common features. The basic item indexing and slicing features are shared. Similarly, a list uses the same kind of negative index values that a string does: the expression `list[-1]` is the last item in a list object.

We'll return to mutable data structures in *Chapter 4*.

## See also

- Sometimes, we need to build a string, and then convert it into bytes. See the *Encoding strings – creating ASCII and UTF-8 bytes* recipe for how we can do this.
- Other times, we'll need to convert bytes into a string. See the *Decoding bytes – how to get proper characters from some bytes* recipe for more information.

## Using the Unicode characters that aren't on our keyboards

A big keyboard might have almost 100 individual keys. Often, fewer than 50 of these keys are letters, numbers, and punctuation. At least a dozen are *function* keys that do things other than simply *insert* letters into a document. Some of the keys are different kinds of *modifiers* that are meant to be used in conjunction with another key—for example, we might have *Shift*, *Ctrl*, *Option*, and *Command*.

Most operating systems will accept simple key combinations that create about 100 or so characters. More elaborate key combinations may create another 100 or so less popular characters. This isn't even close to covering the vast domain of characters from the world's alphabets. And there are icons, emojis, and dingbats galore in our computer fonts. How do we get to all of those glyphs?

## Getting ready

Python works in Unicode. There are thousands of individual Unicode characters available.

We can see all the available characters at [https://en.wikipedia.org/wiki/List\\_of\\_Unicode\\_characters](https://en.wikipedia.org/wiki/List_of_Unicode_characters), as well as at <http://www.unicode.org/charts/>.

We'll need the Unicode character number. We may also want the Unicode character name.

A given font on our computer may not be designed to provide glyphs for all of those characters. In particular, Windows computer fonts may have trouble displaying some of

these characters. Using the following Windows command to change to code page 65001 is sometimes necessary:

```
chcp 65001
```

Linux and macOS rarely have problems with Unicode characters.

## How to do it...

Python uses **escape sequences** to extend the ordinary characters we can type to cover the vast space of Unicode characters. Each escape sequence starts with a `\` character. The next character tells us exactly which of the Unicode characters to create. Locate the character that's needed. Get the name or the number. The numbers are always given as hexadecimal, base 16. Websites describing Unicode often write the character as `U+2680`. The name might be `DIE FACE-1`. Use `\unnnn` with up to a four-digit number, `nnnn`. Or, use `\N{name}` with the spelled-out name. If the number is more than four digits, use `\Unnnnnnnn` with the number padded out to exactly eight digits:

```
>>> 'You Rolled \u2680'
```

```
'You Rolled 🎲'
```

```
>>> 'You drew \U0001F000'
```

```
'You drew 🀄'
```

```
>>> 'Discard \N{MAHJONG TILE RED DRAGON}'
```

```
'Discard 🀅'
```

Yes, we can include a wide variety of characters in Python output. To place a `\` in the string without the following characters being part of an escape sequence, we need to use `\\`. For example, we might need this for Windows file paths.

## How it works...

Python uses Unicode internally. The 128 or so characters we can type directly using the keyboard all have handy internal Unicode numbers.

When we write:

```
'HELLO'
```

Python treats it as shorthand for this:

```
'\u0048\u0045\u004c\u004c\u0046'
```

Once we get beyond the characters on our keyboards, the remaining thousands of characters are identified only by their number.

When the string is being compiled by Python, `\uxxxx`, `\Uxxxxxxxx`, and `\N{name}` are all replaced by the proper Unicode character. If we have something syntactically wrong—for example, `\N{name` with no closing `}`—we’ll get an immediate error from Python’s internal syntax checking.

Regular expressions use a lot of `\` characters and that we specifically do not want Python’s normal compiler to touch them; we used the `r'` prefix on a regular expression string to prevent `\` from being treated as an escape and possibly converted into something else. To use the full domain of Unicode characters, we cannot avoid using `\` as an escape.

What if we need to use Unicode in a regular expression? We’ll need to use `\\` all over the place in the regular expression. We might see something like this:

```
'\\w+[\u2680\u2681\u2682\u2683\u2684\u2685]\\d+'.
```

We couldn’t use the `r'` prefix on the string because we needed to have the Unicode escapes processed. This forced us to use `\\` for elements of the regular expression. We used `\uxxxx` for the Unicode characters that are part of the pattern. Python’s internal compiler will replace `\uxxxx` with Unicode characters and `\\w` will become the required `\w` internally.

When we look at a string at the `>>>` prompt, Python will display the string in its canonical form. Python prefers to display strings with `'` as a delimiter, using `"` when the string contains a `'`. We can use either `'` or `"` for a string delimiter when writing code. Python doesn’t generally display raw strings; instead, it puts all of the necessary escape sequences back into the string:

```
>>> r"\w+"
'\w+'
```

We provided a string in raw form. Python displayed it in canonical form.

## See also

- In the *Encoding strings – creating ASCII and UTF-8 bytes* and the *Decoding bytes – how to get proper characters from some bytes* recipes, we'll look at how Unicode characters are converted into sequences of bytes so we can write them to a file. We'll look at how bytes from a file (or downloaded from a website) are turned into Unicode characters so they can be processed.
- If you're interested in history, you can read up on ASCII and EBCDIC and other old-fashioned character codes here: <http://www.unicode.org/charts/>.

## Encoding strings – creating ASCII and UTF-8 bytes

Our computer files are bytes. When we upload or download from the internet, the communication works in bytes. A byte only has 256 distinct values. Our Python characters are Unicode. There are a lot more than 256 Unicode characters.

How do we map Unicode characters to bytes to write to a file or for transmission?

### Getting ready

Historically, a character occupied 1 byte. Python leverages the old ASCII encoding scheme for bytes; this sometimes leads to confusion between bytes and text strings of Unicode characters.

Unicode characters are encoded into sequences of bytes. There are a number of standardized encodings and a number of non-standard encodings.

Plus, there also are some encodings that only work for a small subset of Unicode characters.



We try to avoid these, but there are some situations where we'll need to use a subset encoding scheme.

Unless we have a really good reason not to, we almost always use UTF-8 encoding for Unicode characters. Its main advantage is that it's a compact representation of the Latin alphabet, which is used for English and a number of European languages.

Sometimes, an internet protocol requires ASCII characters. This is a special case that requires some care because the ASCII encoding can only handle a small subset of Unicode characters.

## How to do it...

Python will generally use our OS's default encoding for files and internet traffic. The details are unique to each OS:

1. We can make a general setting using the `PYTHONIOENCODING` environment variable. We set this outside of Python to ensure that a particular encoding is used everywhere. When using Linux or macOS, use the shell's `export` statement to set the environment variable. For Windows, use the `set` command, or the PowerShell `Set-Item cmdlet`. For Linux, it looks like this:

```
(cookbook3) % export PYTHONIOENCODING=UTF-8
```

2. Run Python:

```
(cookbook3) % python
```

3. We sometimes need to make specific settings when we open a file inside our script. We'll return to this topic in *Chapter 11*. Open the file with a given encoding. Read or write Unicode characters to the file:

```
>>> with open('some_file.txt', 'w', encoding='utf-8') as output:  
...     print('You drew \U0001F000', file=output)  
>>> with open('some_file.txt', 'r', encoding='utf-8') as input:  
...     text = input.read()
```

```
>>> text
'You drew ☺'
```

We can also manually encode characters, in the rare case that we need to open a file in bytes mode; if we use a mode of `wb`, we'll also need to use manual encoding of each string:

```
>>> string_bytes = 'You drew \U0001F000'.encode('utf-8')
>>> string_bytes
b'You drew \xf0\x9f\x80\x80'
```

We can see that a sequence of bytes (`\xf0\x9f\x80\x80`) was used to encode a single Unicode character, `U+1F000`, ☺.

## How it works...

Unicode defines a number of encoding schemes. While UTF-8 is the most popular, there is also UTF-16 and UTF-32. The number is the typical number of bits per character. A file with 1,000 characters encoded in UTF-32 would be 4,000 8-bit bytes. A file with 1,000 characters encoded in UTF-8 could be as few as 1,000 bytes, depending on the exact mix of characters. In UTF-8 encoding, characters with Unicode numbers above `U+007F` require multiple bytes.

Various OSes have their own coding schemes. macOS files can be encoded in Mac Roman or Latin-1. Windows files might use CP1252 encoding.

The point with all of these schemes is to have a sequence of bytes that can be mapped to a Unicode character and—going the other way—a way to map each Unicode character to one or more bytes. Ideally, all of the Unicode characters are accounted for. Pragmatically, some of these coding schemes are incomplete.

The historical form of ASCII encoding can only represent about 100 of the Unicode characters as bytes. It's easy to create a string that cannot be encoded using the ASCII scheme.

Here's what the error looks like:

```
>>> 'You drew \U0001F000'.encode('ascii')
Traceback (most recent call last):
...
UnicodeEncodeError: 'ascii' codec can't encode character '\U0001f000' in
position 9: ordinal not in range(128)
```

We may see this kind of error when we accidentally open a file with an encoding that's not the widely used standard of UTF-8. When we see this kind of error, we'll need to change our processing to select the encoding actually used to create the file. It's almost impossible to guess what encoding was used, so some research may be required to locate metadata about the file that states the encoding.

Bytes are often displayed using printable characters. We'll see `b'hello'` as shorthand for a five-byte value. The letters are chosen using the old ASCII encoding scheme, where byte values from 0x20 to 0x7F will be shown as characters, and outside this range, more complex-looking escapes will be used.

This use of characters to represent byte values can be confusing. The prefix of `b'` is our hint that we're looking at bytes, not proper Unicode characters.

## See also

- There are a number of ways to build strings of data. See the *Building complicated strings with f-strings* and the *Building complicated strings from lists of strings* recipes for examples of creating complex strings. The idea is that we might have an application that builds a complex string, and then we encode it into bytes.
- For more information on UTF-8 encoding, see <https://en.wikipedia.org/wiki/UTF-8>.
- For general information on Unicode encodings, see [http://unicode.org/faq/utf\\_bom.html](http://unicode.org/faq/utf_bom.html).

## Decoding bytes – how to get proper characters from some bytes

How can we work with files that aren't properly encoded? What do we do with files written in ASCII encoding?

A download from the internet is almost always in bytes—not characters. How do we decode the characters from that stream of bytes?

Also, when we use the subprocess module, the results of an OS command are in bytes. How can we recover proper characters?

Much of this is also relevant to the material in *Chapter 11*. We've included this recipe here because it's the inverse of the previous recipe, *Encoding strings – creating ASCII and UTF-8 bytes*.

### Getting ready

Let's say we're interested in offshore marine weather forecasts. Perhaps this is because we are departing the **Chesapeake Bay** for the **Caribbean**.

Are there any special warnings coming from the **National Weather Services** office in Wakefield, Virginia?

Here's the link:

<https://forecast.weather.gov/product.php?site=AKQ&product=SMW&issuedby=AKQ>.

We can download this with Python's `urllib` module:

```
>>> import urllib.request
>>> warnings_uri = (
...     'https://forecast.weather.gov/'
...     'product.php?site=AKQ&product=SMW&issuedby=AKQ'
... )

>>> with urllib.request.urlopen(warnings_uri) as source:
...     forecast_text = source.read()
```

Note that we've enclosed the URI string in `()` and broken it into two separate string literals. Python will concatenate these two adjacent literals into a single string. We'll look at this in some depth in *Chapter 2*.

As an alternative, we can use programs like `curl` or `wget` to get this. At the OS Terminal prompt, we might run the following (long) command:

```
(cookbook3) % curl 'https://forecast.weather.gov/product.php?site=AKQ&product=SMW&issuedby=AKQ' -o AKQ.html
```

Typesetting this book tends to break the command onto many lines. It's really one very long line.

The code repository includes a sample file, `ch01/Text Products for SMW Issued by AKQ.html`.

The `forecast_text` value is a stream of bytes. It's not a proper string. We can tell because it starts like this:

```
>>> forecast_text[:80]
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/x'
```

The data goes on for a while, providing details from the web page. Because the displayed value starts with `b'`, it's bytes, not proper Unicode characters. It was probably encoded with UTF-8, which means some characters could have weird-looking `\xnn` escape sequences instead of proper characters. We want to have the proper characters.

While this data has many easy-to-read characters, the `b'` prefix shows that it's a collection of byte values, not proper text. Generally, a bytes object behaves somewhat like a string object. Sometimes, we can work with bytes directly. Most of the time, we'll want to decode the bytes and create proper Unicode characters from them.

## How to do it...

1. Determine the coding scheme if possible. In order to decode bytes to create proper Unicode characters, we need to know what encoding scheme was used. When we read XML documents, there's a big hint provided within the document:

```
<?xml version="1.0" encoding="UTF-8"?>
```

When browsing web pages, there's often a header containing this information:

```
Content-Type: text/html; charset=ISO-8859-4
```

Sometimes, an HTML page may include this as part of the header:

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
```

In other cases, we're left to guess. In the case of US weather data, a good first guess is UTF-8. Another good guess is ISO-8859-1. In some cases, the guess will depend on the language.

2. The *codecs* — *Codec registry and base classes* section of the *Python Standard Library* lists the standard encodings available. Decode the data:

```
>>> document = forecast_text.decode("UTF-8")
>>> document[:80]
'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/X'
```

The `b'` prefix is no longer used to show that these are bytes. We've created a proper string of Unicode characters from the stream of bytes.

3. If this step fails with an exception, we guessed wrong about the encoding. We need to try another encoding in order to parse the resulting document.

Since this is an HTML document, we should use **Beautiful Soup** to extract the data. See <http://www.crummy.com/software/BeautifulSoup/>.

We can, however, extract one nugget of information from this document without completely parsing the HTML:

```
>>> import re
>>> content_pattern = re.compile(r"// CONTENT STARTS(.*)// CONTENT ENDS",
re.MULTILINE | re.DOTALL)
>>> content_pattern.search(document)
<re.Match object; span=(8530, 9113), match='// CONTENT STARTS HERE
-->\n\n<span style="font-s>
```

This tells us what we need to know: there are no warnings at this time. This doesn't mean smooth sailing, but it does mean that there aren't any major weather systems that could cause catastrophes.

## How it works...

See the *Encoding strings – creating ASCII and UTF-8 bytes* recipe for more information on Unicode and the different ways that Unicode characters can be encoded into streams of bytes.

At the foundation of the OS, files and network connections are built up from bytes. It's our software that decodes the bytes to discover the content. It might be characters, images, or sounds. In some cases, the default assumptions are wrong and we need to do our own decoding.

## See also

- Once we've recovered the string data, we have a number of ways of parsing or rewriting it. See the *String parsing with regular expressions* recipe for examples of parsing a complex string.
- For more information on encodings, see <https://en.wikipedia.org/wiki/UTF-8> and [http://unicode.org/faq/utf\\_bom.html](http://unicode.org/faq/utf_bom.html).

## Using tuples of items

What's the best way to represent simple  $(x, y)$  and  $(r, g, b)$  groups of values? How can we keep things that are pairs, such as latitude and longitude, together?

## Getting ready

In the *String parsing with regular expressions* recipe, we skipped over an interesting data structure.

We had data that looked like this:

```
>>> ingredient = "Kumquat: 2 cups"
```

We parsed this into meaningful data using a regular expression, like this:

```
>>> import re
>>> ingredient_pattern =
re.compile(r'(?P<ingredient>\w+):\s+(?P<amount>\d+)\s+(?P<unit>\w+)')
>>> match = ingredient_pattern.match(ingredient)
>>> match.groups()
('Kumquat', '2', 'cups')
```

The result is a tuple object with three pieces of data. There are lots of places where this kind of grouped data can come in handy.

## How to do it...

We'll look at two aspects to this: putting things into tuples and getting things out of tuples.

### Creating tuples

There are lots of places where Python creates tuples of data for us. In the *Getting ready* section of the *String parsing with regular expressions* recipe, we showed you how a regular expression match object will create a tuple of text that was parsed from a string.

We can create our own tuples, too. Here are the steps:

1. Enclose the data in `()`.
2. Separate the items with `,`.

```
>>> from fractions import Fraction
>>> my_data = ('Rice', Fraction(1/4), 'cups')
```



There's an important special case for the one-tuple, or singleton. We have to include the `,` even when there's only one item in the tuple:

```
>>> one_tuple = ('item', )
>>> len(one_tuple)
1
```

The `()` characters aren't always required. There are a few times where we can omit them. It's not a good idea to omit them.

It's the comma that creates a tuple of values. This means we can see funny things when we have an extra comma:

```
>>> 355,
(355, )
```

The comma after 355 turns the value into a singleton tuple.

We can also create a tuple by conversion from another sequence. For example, `tuple([355])` creates a singleton tuple from a singleton list.

## Extracting items from a tuple

The idea of a tuple is to be a container with a number of items that's fixed by the problem domain: for example, for (red, green, blue) color numbers, the number of items is always three.

In our example, we've got an ingredient, and amount, and units. This must be a three-item collection. We can look at the individual items in two ways:

- By index position; that is, positions are numbered starting with zero from the left:

```
>>> my_data[1]
Fraction(1, 4)
```

- Using multiple assignment:

```
>>> ingredient, amount, unit = my_data
>>> ingredient
'Rice'
>>> unit
'cups'
```

Tuples—like strings—are immutable. We can't change the individual items inside a tuple. We use tuples when we want to keep the data together.

## How it works...

Tuples are one example of the more general Sequence class. We can do a few things with sequences.

Here's an example tuple that we can work with:

```
>>> t = ('Kumquat', '2', 'cups')
```

Here are some operations we can perform on this tuple:

- How many items in t?

```
>>> len(t)
3
```

- How many times does a particular value appear in t?

```
>>> t.count('2')
1
```

- Which position has a particular value?

```
>>> t.index('cups')
2
>>> t[2]
'cups'
```

- When an item doesn't exist, we'll get an exception:

```
>>> t.index('Rice')
Traceback (most recent call last):
...
ValueError: tuple.index(x): x not in tuple
```

- Does a particular value exist?

```
>>> 'Rice' in t
False
```

## There's more...

A tuple, like a string, is a sequence of items. In the case of a string, it's a sequence of characters. In the case of a tuple, it's a sequence of many things. Because they're both sequences, they have some common features. We've noted that we can pluck out individual items by their index position. We can use the `index()` method to locate the position of an item.

The similarities end there. A string has many methods it can use to create a new string that's a transformation of a string, plus methods to parse strings, plus methods to determine the content of the strings. A tuple doesn't have any of these bonus features. It's—perhaps—the simplest possible data structure.

## See also

- We looked at one other sequence, the list, in the *Building complicated strings from lists of strings* recipe.
- We'll also look at sequences in *Chapter 4*.

## Using NamedTuples to simplify item access in tuples

When we worked with tuples, we had to remember the positions as numbers. When we use a  $(r, g, b)$  tuple to represent a color, can we use “red” instead of zero, “green” instead of

1, and “blue” instead of 2?

## Getting ready

Let’s continue looking at items in recipes. The regular expression for parsing the string had three attributes: `ingredient`, `amount`, and `unit`. We used the following pattern with names for the various substrings:

```
r'(?P<ingredient>\w+):\s+(?P<amount>\d+)\s+(?P<unit>\w+)'
```

The resulting data tuple looked like this:

```
>>> item = match.groups()
>>> item
('Kumquat', '2', 'cups')
```

While the matching between `ingredient`, `amount`, and `unit` is pretty clear, using something like the following isn’t ideal. What does 1 mean? Is it really the quantity?

```
>>> from fractions import Fraction
>>> Fraction(item[1])
Fraction(2, 1)
```

We want to define tuples with names, as well as positions.

## How to do it...

1. We’ll use the `NamedTuple` class definition from the `typing` package:

```
>>> from typing import NamedTuple
```

2. With this base class definition, we can define our own unique tuples, with names for the items:

```
>>> class Ingredient(NamedTuple):
...     ingredient: str
...     amount: str
```

```
...     unit: str
```

3. Now, we can create an instance of this unique kind of tuple by using the classname:

```
>>> item_2 = Ingredient('Kumquat', '2', 'cups')
```

4. When we want a value from the tuple, we can use a name instead of the position:

```
>>> Fraction(item_2.amount)
Fraction(2, 1)
>>> f"Use {item_2.amount} {item_2.unit} fresh {item_2.ingredient}"
'Use 2 cups fresh Kumquat'
```

## How it works...

The `NamedTuple` class definition introduces a core concept from *Chapter 7*. We've extended the base class definition to add unique features for our application. In this case, we've named the three attributes each `Ingredient` tuple must contain.

Because a subclass of `NamedTuple` class is a tuple, the order of the attribute names is fixed. We can use a reference like the expression `item_2[0]` as well as the expression `item_2.ingredient`. Both names refer to the item in index 0 of the tuple, `item_2`.

The core tuple types can be called “anonymous tuples” or maybe “index-only tuples.” This can help to distinguish them from the more sophisticated “named tuples” introduced through the `typing` module.

Tuples are very useful as tiny containers of closely related data. Using the `NamedTuple` class definition makes them even easier to work with.

## There's more...

We can have a mixed collection of values in a tuple or a named tuple. We need to perform conversion before we can build the tuple. It's important to remember that a tuple cannot ever be changed. It's an immutable object, similar in many ways to the way strings and

numbers are immutable.

For example, we might want to work with amounts that are exact fractions. Here's a more sophisticated definition:

```
>>> from typing import NamedTuple
>>> from fractions import Fraction
>>> class IngredientF(NamedTuple):
...     ingredient: str
...     amount: Fraction
...     unit: str
```

These objects require some care to create. If we're using a bunch of strings, we can't simply build this object from three string values; we need to convert the amount into a `Fraction` instance. Here's an example of creating an item using a `Fraction` conversion:

```
>>> item_3 = IngredientF('Kumquat', Fraction('2'), 'cups')
```

This tuple has a more useful value for the amount of each ingredient. We can now do mathematical operations on the amounts:

```
>>> f'{item_3.ingredient} doubled: {item_3.amount * 2}'
'Kumquat doubled: 4'
```

It's very handy to explicitly state the data type within the `NamedTuple` class definition. It turns out Python doesn't use the type information directly. Other tools, for example, **mypy**, can check the type hints in a `NamedTuple` against the operations in the rest of the code to be sure they agree.

## See also

- We'll look at class definitions in *Chapter 7*.

## Join our community Discord space

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

<https://packt.link/dHrHU>



# 2

## Statements and Syntax

Python syntax is designed to be simple. In this chapter, we'll look at some of the most commonly used statements in the language as a way to understand the rules. Concrete examples can help clarify the language's syntax.

We'll cover some of the basics of creating script files first. Then we'll move on to looking at some of the more commonly used statements. Python only has about 20 or so different kinds of imperative statements in the language. We've already looked at two kinds of statements in *Chapter 1*, the assignment statement and the expression statement.

When we write something like this:

```
>>> print("hello world")
hello world
```

We're actually executing a statement that contains only the evaluation of a function, `print()`. This kind of statement—where we evaluate a function or a method of an object—is common.



The other kind of statement we've already seen is the assignment statement. Python has many variations on this theme. Most of the time, we're assigning a single value to a single variable. When a function returns a tuple as a result, we can unpack that collection and assign more than one variable at the same time in a single assignment statement. It is done like this:

```
>>> quotient, remainder = divmod(355, 113)
```

The recipes in this chapter will look at the `if`, `while`, `for`, `with`, and `try` statements. We'll also touch on a few of the simpler statements as we go, like `pass`, `break`, and `raise`.

In later chapters, we'll look at other statements. Here's a summary:

Statement	Chapter
<code>def</code>	<i>Chapter 3</i>
<code>return</code>	<i>Chapter 3</i>
<code>import</code>	<i>Chapter 3</i>
<code>del</code>	<i>Chapter 4</i>
<code>class</code>	<i>Chapter 7</i>
<code>match</code>	<i>Chapter 8</i>
<code>type</code>	<i>Chapter 10</i>
<code>assert</code>	<i>Chapter 10</i>

*Table 2.1: Python Statements and Chapters*

In this chapter, we'll look at the following recipes:

- *Writing Python script and module files – syntax basics*
- *Writing long lines of code*
- *Including descriptions and documentation*
- *Writing better docstrings with RST markup*
- *Designing complex if...elif chains*
- *Saving intermediate results with the `:=` "walrus" operator*

- *Avoiding a potential problem with break statements*
- *Leveraging exception matching rules*
- *Avoiding a potential problem with an except: clause*
- *Concealing an exception root cause*
- *Managing a context using the with statement*

We'll start by looking at the big picture – scripts and modules – and then we'll move down into details of individual statements.

## Writing Python script and module files – syntax basics

The point of Python (and programming in general) is to create automated solutions to problems that involve data and processing. Further, the software we write is a kind of knowledge representation; this means clarity is perhaps the most important quality aspect of software.

In Python, we implement automated solutions by creating script files. These are the top-level, main programs of Python programming. In addition to main scripts, we may also create modules (and packages of modules) to help organize the software into intellectually manageable chunks. A script is a module; however, it has a distinct intent to do useful processing when started by the OS.

A key part of creating clear, readable Python files is making sure our code follows the widely adopted conventions.

For example, we need to be sure to save our files in UTF-8 encoding. While ASCII encoding is still supported by Python, it's a poor choice for modern programming. We'll also need to be sure our editor uses spaces instead of the tab character. This is often a configuration setting in programming editors. Using Unix newlines is also helpful for portability.

## Getting ready

To edit Python scripts, we'll need a good programming editor. It's nearly impossible to suggest just one. So we'll suggest a few.

The JetBrains PyCharm editor has numerous features. The community edition version is free. See <https://www.jetbrains.com/pycharm/download/>.

ActiveState has Komodo IDE, which is also very sophisticated. The Komodo Edit version is free and does some of the same things as the full Komodo IDE. See <http://komodoide.com/komodo-edit/>.

Notepad++ is good for Windows developers. See <https://notepad-plus-plus.org>.

BBEdit is very nice for macOS X developers. See <http://www.barebones.com/products/bbedit/>. Sublime is also popular on macOS X. See <https://www.sublimetext.com>.

For Linux developers, there are several built-in editors, including **Vim** and **gedit**. Since Linux tends to be biased toward developers, the editors available are all suitable for writing Python.

It is helpful is to have two windows open while working:

- An editor to create the final script or module file.
- A terminal session with Python's >>> prompt, where we can try things out to see what works and what doesn't.

Most editors recognize the .py extension and provide appropriate formatting based on PEP-8. This generally includes the following:

- The file encoding should be UTF-8.
- Indentation should be four spaces.
- We want the *Tab* key on the keyboard to insert spaces instead of the tab character, `\t`.

Once the editor is configured, we can write a script file that other people can easily use or

extend.

## How to do it...

Here's how we create a script file:

1. The first line of most Python script files looks like this:

```
#!/usr/bin/env python3
```

This sets an association between the file you're writing and Python. If the file's mode is set to executable with the bash `chmod` command, and the directory is on the OS `PATH` list, the script will be a first-class application, as usable as any of the built-in commands.

For Windows, the filename-to-program association is done through a setting in the **Default Programs** control panel. Find the panel for **Set Associations**, and make sure `.py` files are bound to the Python program. This is often set by the installer, and we rarely need to change it or set it manually.

2. After the preamble, convention suggests we include a triple-quoted block of text. This is the documentation string (called a **docstring**) for the file we're going to create:

```
"""  
A summary of this script.  
"""
```

Because Python triple-quoted strings can be indefinitely long, feel free to write as much as necessary. This should be the primary vehicle for describing the script or library module. This can even include examples of how it works.

3. Now comes the interesting part of the script: the part that really does something. We can write all the statements we need to get the job done. For now, we'll use this as a placeholder:

```
print('hello world')
```

This isn't much, but at least the script does something. It's common to create function and class definitions, as well as to write statements to use the functions and classes to do things.

For our first, simple script, all of the statements must begin at the left margin and must be complete on a single line. There are many Python statements that have blocks of statements nested inside them. These internal blocks of statements will be indented to clarify their scope. Generally—because we set indentation to four spaces—we can hit the *Tab* key to properly indent the code.

Our file should look like this:

```
#!/usr/bin/env python3
"""
My First Script: Calculate an important value.
"""

print(355 / 113)
```

## How it works...

Unlike other languages, there's very little *boilerplate* in Python. There's only one line of *overhead* and even the `#!/usr/bin/env python3` line is generally optional.

Why do we set the encoding to UTF-8? While the language was originally designed to work using just the original 128 ASCII characters, we often find that ASCII is limiting. This is legal Python if we save our file in UTF-8:

```
 $\mu = 355/113$ 
print( $\mu$ )
```

It's important to be consistent when choosing between spaces and tabs in Python. They are both more or less invisible, and mixing them up can easily lead to errors when trying

to run the script. Spaces are suggested.

The initial `#!` line is a comment. Because the two characters are sometimes called sharp and bang, the combination is called “shebang.” Everything between a `#` and the end of the line is ignored. The Linux loader (a program named **execve**) looks at the first few bytes of a file to see what the file contains. These first few bytes are sometimes called *magic bytes* because the loader’s behavior seems magical. When present, this two-character sequence of `#!` is followed by the path to the program responsible for processing the rest of the data in the file. We prefer to use `/usr/bin/env` to start the Python program for us. We can leverage the `env` program to make Python-specific environment settings.

## There’s more...

The *Python Standard Library* documents are derived, in part, from the documentation strings present in the module files. It’s common practice to write sophisticated docstrings in modules, packages, and scripts. There are tools like **pydoc** and **Sphinx** that can reformat the module docstrings into elegant documentation. We’ll look at this in the *Writing better docstrings with RST markup* recipe, as well as the *Using Sphinx autodoc to create the API reference* recipe in *Chapter 17*.

Additionally, unit test cases can be included in the docstrings. Tools like **doctest** can extract examples from the document strings and execute the code to see if the answers in the documentation match the answers found by running the code. This is the subject of many recipes in *Chapter 15*. Many examples in this book are validated by **doctest**.

Triple-quoted documentation strings are preferred over `#` comments. While all text between `#` and the end of the line is ignored, this is limited to a single line; the conventional approach is to use it sparingly. A docstring can be of indefinite size; they are used widely.

There’s another bit of overhead that’s sometimes included. The **Vim** and **gedit** editors let us keep edit preferences in the file. This is called a **modeline**. Here’s a typical modeline that’s useful for Python:

```
# vim: tabstop=8 expandtab shiftwidth=4 softtabstop=4
```

This makes sure any tab characters will be transformed into eight spaces; when we hit the *Tab* key, we'll shift four spaces. This is widely used because tab characters are traditionally indented eight spaces, and this replacement is likely to create proper indentation. This setting is embedded in the code; we don't have to do any **Vim** setup to apply these settings to our Python script files.

## See also

- We'll look at how to write useful document strings in the *Including descriptions and documentation* and *Writing better docstrings with RST markup* recipes.
- For more information on suggested style, see PEP-8.

## Writing long lines of code

There are many times when we need to write lines of code that are so long that they're very hard to read. Many people like to limit the length of a line of code to 80 characters or fewer. It's a well-known principle of graphic design that a narrower area of text is easier to read. See <http://webtypography.net/2.1.2> for a deeper discussion of line width and readability.

While fewer characters per line is easier on the eyes, our code can refuse to cooperate with this principle. How can we break long Python statements into more manageable pieces?

## Getting ready

Let's say we've got something like this:

```
>>> import math

>>> example_value = (63/25) * (17+15*math.sqrt(5)) / (7+15*math.sqrt(5))

>>> mantissa_fraction, exponent = math.frexp(example_value)
>>> mantissa_whole = int(mantissa_fraction*2**53)
>>> message_text = f'the internal representation is
{mantissa_whole:d}/2**53*2**{exponent:d}'
>>> print(message_text)
```

```
the internal representation is 7074237752514592/2**53*2**2
```

This code includes a long formula, and a long format string into which we're injecting values. This looks bad when typeset in a book; the f-string line may be broken incorrectly. It may look bad on our screen when trying to edit this script. (For more on f-strings, see *Building complicated strings with f-strings* in *Chapter 1*.)

We can't haphazardly break Python statements into chunks. The syntax rules are clear that a statement must be complete on a single *logical* line.

The term "logical line" provides a hint as to how we can proceed. Python makes a distinction between logical lines and physical lines; we'll leverage these syntax rules to break up long statements.

## How to do it...

Python gives us several ways to wrap long statements so they're more readable:

- We can use `\` at the end of a line to continue the logical line onto the next physical line. While this always works, it can be hard to spot the `\`.
- Python has a rule that a statement can span multiple *logical* lines because the `()`, `[]`, and `{}` characters must balance. Further, we can also exploit the way Python automatically concatenates adjacent string literals to make a single, longer string literal: `"a" "b"` is the same as `"ab"`.
- In some cases, we can decompose a statement into multiple statements by assigning intermediate results to separate variables.

We'll look at each one of these in separate parts of this recipe.

### Using a backslash to break a long statement into logical lines

1. If there's a *meaningful* break, insert the `\` to separate the statement:



```
>>> message_text = f'the internal representation is \
... {mantissa_whole:d}/2**53*2**{exponent:d}'
```

For this to work, the `\` must be the *last* character on the line. An extra space after the `\` is fairly hard to see; some care is required. The PEP-8 proposal provides guidelines on formatting and tends to discourage this technique.

In spite of this being a little hard to see, the `\` can always be used. Think of it as the last resort in making a line of code more readable.

## Using the () characters to break a long statement into sensible pieces

1. Write the whole statement on one line, even if it's confusing:

```
>>> import math

>>> example_value1 = (63/25) * (17+15*math.sqrt(5)) /
(7+15*math.sqrt(5))
```

Add the extra `()` characters, which don't change the value but allow breaking the expression into multiple lines:

```
>>> example_value2 = (63/25) * ( (17+15*math.sqrt(5)) /
(7+15*math.sqrt(5)) )

>>> example_value2 == example_value1
True
```

2. Break the line inside the `()` characters:

```
>>> example_value3 = (63/25) * (
...     (17+15*math.sqrt(5))
...     / (7+15*math.sqrt(5))
... )

>>> example_value3 == example_value1
True
```

The matching `()` characters technique is quite powerful and will work in a wide variety of

cases. This is widely used and highly recommended.

We can almost always find a way to add extra `()` characters to a statement. In rare cases when we can't add `()` characters, we can fall back on using `\` to break the statement into sections.

## Using string literal concatenation

We can combine the `()` characters with another rule that joins adjacent string literals. This is particularly effective for long, complex format strings:

1. Wrap the long string value in the `()` characters.
2. Break the string into meaningful substrings:

```
>>> message_text = (  
... f'the internal representation '  
... f'is {mantissa_whole:d}/2**53*2**{exponent:d}'  
... )  
  
>>> message_text  
'the internal representation is 7074237752514592/2**53*2**2'
```

We can always break a long string literal into adjacent pieces. We can then use as many physical line breaks as we need. With string literal values, no explicit operator is needed.

## Assigning intermediate results to separate variables

Here's the context for this technique:

```
>>> import math  
  
>>> example_value = (63/25) * (17+15*math.sqrt(5)) / (7+15*math.sqrt(5))
```

We can break this into three intermediate values:

1. Identify sub-expressions in the overall expression. Assign these to variables:

```
>>> a = (63/25)
>>> b = (17+15*math.sqrt(5))
>>> c = (7+15*math.sqrt(5))
```

2. Replace the sub-expressions with the variables that were created:

```
>>> example_value = a * b / c
```

We can always take a sub-expression and assign it to a variable, and use the variable everywhere the sub-expression was used. The `15*sqrt(5)` product is repeated; this, too, is a good candidate for refactoring the expression.

We didn't give these variables descriptive names. In some cases, the sub-expressions have some semantics that we can capture with meaningful names.

## How it works...

The Python language manual makes a distinction between logical lines and physical lines. A logical line contains a complete statement. It can span multiple physical lines through a technique called **line joining**. The manual identifies two techniques: **explicit line joining** and **implicit line joining**.

The use of `\` for explicit line joining is sometimes helpful. Because it's easy to overlook, it's not generally encouraged. PEP-8 suggests this should be the method of last resort.

The use of `()` for implicit line joining can be used in many cases. It often fits semantically with the structure of the expressions, so it is encouraged.

## There's more...

Expressions are used widely in a number of Python statements. Any expression can have `()` characters added. This gives us a lot of flexibility.

There are, however, a few places where we may have a long statement that does not specifically involve a long expression. The most notable example of this is the `import`

statement—it can become long, but doesn't use any expressions. In spite of not having a proper expression, it does, however, still permit the use of `()`. The following example shows we can surround a very long list of imported names:

```
>>> from math import (  
...     sin, cos, tan,  
...     sqrt, log, frexp)
```

While the `()` characters are emphatically *not* part of an expression, they are part of the syntax available to help make the statement more readable.

## See also

- Implicit line joining also applies to the matching `[]` and `{}` characters. These apply to collection data structures that we'll look at in *Chapter 4*.

## Including descriptions and documentation

When we have a useful script, we often need to leave notes for ourselves—and others—on what it does, how it solves some particular problem, and when it should be used. This recipe contains a suggested outline to help make the documentation reasonably complete.

## Getting ready

If we've used the *Writing Python script and module files – syntax basics* recipe to start a script file, we'll have a small documentation string in place. We'll expand on this documentation string in this recipe.

There are other places where documentation strings should be used. We'll look at these additional locations in *Chapter 3* and *Chapter 7*.

We have two general kinds of modules for which we'll be writing summary docstrings:

- **Library modules:** These files will contain mostly function definitions as well as class definitions. The docstring summary should focus on the definitions in the module, describing what the module is. The docstring can provide examples of using

the functions and classes that are defined in the module. In *Chapter 3*, and *Chapter 7*, we'll look more closely at these modules.

- **Scripts:** These are files that we generally expect will do some real work. The docstring should describe what the module does and how to use it. The options, environment variables, and configuration files are important parts of this docstring.

We will sometimes create files that contain a little of both. This requires a proper balance between doing and being.

## How to do it...

The first step in writing documentation is the same for both library modules and scripts:

1. Write a brief summary of what the script or module is or does. The summary doesn't need to dig too deeply into how it works. Like a lede in a newspaper article, it introduces the who, what, when, where, how, and why of the module. Details will follow in the body of the docstring.

It can help to avoid needless phrases like *This script*. We might start our module docstring like this:

```
"""
Downloads and decodes the current Special Marine Warning (SMW)
for the area 'AKQ'.
"""
```

We'll separate the other steps based on the general focus of the module.

## Writing docstrings for scripts

When we document a script, we need to focus on the needs of a person who will use the script.

1. Start as shown earlier, creating a summary sentence.
2. Sketch an outline for the rest of the docstring. We'll be using **ReStructuredText** (RST) markup. Write the topic on one line, then put a line of = under the topic to

make it a proper section title. Remember to leave a blank line between each topic.

Topics may include:

- **SYNOPSIS:** A summary of how to run this script. If the script uses the `argparse` module to process command-line arguments, the help text produced by `argparse` is the ideal synopsis text. Other installable tools like **click** or **invoke** can also produce elegant synopsis text. (See *Using argparse to get command-line input* in Chapter 6.)
- **DESCRIPTION:** An explanation of what this script does.
- **OPTIONS:** This provides the details of all parameters and options. (See *Using argparse to get command-line input* in Chapter 6.)
- **ENVIRONMENT:** This provides the place to describe the environment variables and what they mean. (See *Using the OS environment settings* in Chapter 6.)
- **FILES:** The names of files that are created or read by a script are very important pieces of information.
- **EXAMPLES:** Some examples of using the script are always helpful. In some cases, this is the only part a user will read.
- **SEE ALSO:** Any related scripts or background information.

Other topics that might be interesting include **EXIT STATUS**, **AUTHOR**, **BUGS**, **REPORTING BUGS**, **HISTORY**, or **COPYRIGHT**. In some cases, advice on reporting bugs, for instance, doesn't really belong in a module's docstring, but rather elsewhere in the project's GitHub or SourceForge pages.

3. Fill in the details under each topic. It's important to be accurate. Since the documentation is in the same file as the code, it's easier to be correct, complete, and consistent.

Here's an example of a docstring for a script:

```

"""
Downloads and decodes the current Special Marine Warning (SMW)
for the area \textquotesingle AKQ\textquotesingle{}

SYNOPSIS
=====

::

    python3 akq_weather.py

DESCRIPTION
=====

Downloads the Special Marine Warnings

Files
=====

Writes a file, 'AKW.html'.

EXAMPLES
=====

Here's an example::

    slott\> python3 akq_weather.py
    None issued by this office recently.

```

In the SYNOPSIS section, we used `::` as a separate paragraph. In the EXAMPLES section, we used `::` at the end of a paragraph. Both versions are hints to the RST processing tools that the indented section that follows should be typeset as code. See *Chapter 17, Documentation and Style*.

## Writing docstrings for library modules

When we document a library module, we need to focus on the needs of a programmer who will import the module to use it in their code:

1. Sketch an outline for the rest of the docstring. We'll be using RST markup. Write the topic on one line. Include a line of `=` characters under each topic to make the topic

into a proper heading. Remember to leave a blank line between each paragraph.

2. Start as shown previously, creating a summary sentence:

- **DESCRIPTION:** A summary of what the module contains and why the module is useful
- **MODULE CONTENTS:** The classes and functions defined in this module
- **EXAMPLES:** Examples of using the module

3. Fill in the details for each topic. The module contents may be a long list of class or function definitions. The docstring should be a summary. Within each class or function, we'll have a separate docstring with the details for that item.

## How it works...

Over the decades, the *man page* outline has evolved to contain a complete description of Linux commands. This general approach to writing documentation has proven useful and resilient. We can capitalize on a large body of experience, and structure our documentation to follow the man page model.

We want to prepare module docstrings that can be used by the **Sphinx** Python documentation generator (see <http://www.sphinx-doc.org/en/stable/>). This is the tool used to produce Python's documentation files. The **autodoc** extension in **Sphinx** will read the docstring headers on our modules, classes, and functions to produce the final documentation that looks like other modules in the Python ecosystem.

## There's more...

RST markup has a simple, central syntax rule: paragraphs are separated by blank lines.

This rule makes it easy to write documents that can be examined by the various RST processing tools and reformatted to look nice.

It can be challenging to write good software documentation. There's a broad chasm between too little information and documentation that recapitulates details apparent from looking



at the code.

What's important is to focus on the needs of a person who doesn't know too much about the software or how it works, but can read the Python code. Provide this *semi-knowledgeable* user with the information they need to understand what the software does and how to use it.

## See also

- We look at additional techniques in *Writing better docstrings with RST markup*.
- If we've used the *Writing Python script and module files – syntax basics* recipe, we'll have put a documentation string in our script file. When we build functions in *Chapter 3*, and classes in *Chapter 7*, we'll look at other places where documentation strings can be placed.
- See <http://www.sphinx-doc.org/en/stable/> for more information on **Sphinx**.
- For more background on the man page outline, see [https://en.wikipedia.org/wiki/Man\\_page](https://en.wikipedia.org/wiki/Man_page).

## Writing better docstrings with RST markup

When we have a useful script, we often need to leave notes on what it does, how it works, and when it should be used. Many tools for producing documentation, including **Docutils**, work with RST markup. This allows us to write plain text documentation. It can include some special punctuation to pick a **bold** or *italic* font variant to call attention to details. In addition, RST permits organizing content via lists and section headings.

## Getting ready

In the *Including descriptions and documentation* recipe, we looked at putting some basic documentation into a module. We'll look at a few of the RST formatting rules for creating readable documentation.

## How to do it...

1. Start with an outline of the key point, creating RST section titles to organize the material. A section title has a one-line title followed by a line of underline characters using =, -, ^, ~ as long as the title.

A heading will look like this:

```
Topic
=====
```

The heading text is on one line and the underlining characters are on the next line. This must be surrounded by blank lines. There can be more underline characters than title characters, but never fewer.

The RST tools will infer our chosen pattern of using underlining characters. As long as the underline characters are used consistently, the docutil tools will detect the document's structure.

When starting out, it can help to have an explicit standard for heading underlines:

Character	Level
=	1
-	2
^	3
~	4

2. Fill in the various paragraphs. Paragraphs (including the section titles) are separated by at least one empty line.
3. If the programming editor has a spell checker, use it. Doing this can be frustrating because the code samples often have abbreviations that fail spell checking.

## How it works...

The **Docutils** conversion programs will examine the document, looking for sections and body elements. A section is identified by a title. The underlines are used to organize the

sections into a properly nested hierarchy.

A properly nested document might have the following sequence of underline characters:

TITLE

=====

SOMETHING

-----

MORE

^^^^

EXTRA

^^^^

LEVEL 2

-----

LEVEL 3

^^^^^^

When an HTML file is created from the documentation it will have `<h1>`, `<h2>`, and `<h3>` tags for the various levels. Creating a `LATEX` file requires some additional configuration choices, but the common `Article` template means the resulting document will use `\section`, `\subsection`, and `\subsubsection` headings. These final presentation choices aren't our primary concern when writing; the most important point is to use proper underlines to reflect the desired organization.

There are several different kinds of body elements the RST parser can recognize. We've shown a few. A more complete list includes:

- **Paragraphs of text:** These might use inline markup for different kinds of emphasis or highlighting.

- **Literal blocks:** These are introduced with `::` and indented with four spaces. They may also be introduced with the `.. parsed-literal::` directive. A **doctest** block is indented with four spaces and includes the Python `>>>` prompt.
- **Lists, tables, and block quotes:** We'll look at these later. These can contain other body elements.
- **Footnotes:** These are special paragraphs. When rendered, they may be displayed at the bottom of a page or at the end of a section. These can also contain other body elements.
- **Hyperlink targets, substitution definitions, and RST comments:** These are more specialized text items that we won't look at closely here.

## There's more...

In the *Including descriptions and documentation* recipe, we looked at several different kinds of body elements we might use:

- **Paragraphs of text:** This is a block of text surrounded by blank lines. Within these, we can make use of *inline* markup to emphasize words or phrases. We'll look at inline markup in the *Writing better docstrings with RST markup* recipe.
- **Lists:** These are paragraphs that begin with something that looks like a number or a bullet. We might have paragraphs like this.

It helps to have bullets because:

- They can help clarify
- They can help organize

Other characters can be used at the start of the line, but `-` and `*` seem to be the most common choices.

- **Numbered lists:** There are a variety of patterns that are recognized. This includes leading digits or letters followed by `.` or `)`. Using `#` instead of a digit or letter will continue from the previous paragraph value.

- **Literal blocks:** A code sample is presented literally, without looking for RST elements. The text for this must be indented. A handy prefix is `::`. A `.. code-block::` directive is also possible.
- **Directives:** A directive is a paragraph that generally looks like `.. directive::`. It may have some content that's indented to be contained within the directive. It might look like this:

```
.. important::
```

```
    Do not flip the bozo bit.
```

The `.. important::` text is the directive. This is followed by text indented within the directive.

## Using directives

Docutils has several built-in directives. The **Sphinx** tool adds a large number of additional directives with a variety of features.

Some of the most commonly used directives are the admonitions: `attention`, `caution`, `danger`, `error`, `hint`, `important`, `note`, `tip`, `warning`, and a generic admonition. These are compound body elements because they have nested text within them. Above, we provided an example of the `important` admonition.

## Using inline markup

Within a paragraph, we have several forms of inline markup we can use:

- We can surround a word or phrase with `*` for `*emphasis*`. This is commonly typeset as *italic*.
- We can surround a word or phrase with `**` for `**strong**`. This is commonly typeset as **bold**.
- We surround references with single back-ticks, ```. Links are followed by an underscore, `_`. We might use ``section title`_` to refer to a specific section within a document.

We don't generally need to put any markup around URLs. The **Docutils** tools recognize these. Sometimes we want a word or phrase to be shown and the URL concealed. We can use this: ``the  $\textbf{Sphinx}$  documentation <http://www.sphinx-doc.org/en/stable/>`_.`

- We can surround code-related words with a double back-tick, ````, to make them look like ```code```. This will be typeset as code.

There's also a more general technique called a *role*. A role starts with `:word:` as the role name, followed by the applicable word or phrase in single ``` back-ticks. A text role looks like this: `:strong:`this``.

There are a number of standard role names, including `:emphasis:`, `:literal:`, `:code:`, `:math:`, `:pep-reference:`, `:rfc-reference:`, `:strong:`, `:subscript:`, `:superscript:`, and `:title-reference:`. Some of these are also available with simpler markup like `*emphasis*` or `**strong**`.

Also, we can define new roles with a directive. If we want to do very sophisticated processing, we can provide the **Docutils** tool with class definitions for handling new roles. This allows us to tweak the way our document is processed.

## See also

- For more information on RST syntax, see <http://docutils.sourceforge.net>. This includes a description of the **Docutils** tool.
- For information on **Sphinx Python Documentation Generator**, see <http://www.sphinx-doc.org/en/stable/>.

## Designing complex if...elif chains

In most cases, our scripts will involve a number of choices. Sometimes the choices are simple, and we can judge the quality of the design with a glance at the code. In other cases, the choices are more complicated, and it's not easy to determine whether or not our `if` statements are designed properly to handle all of the conditions.

In the simplest case, we have one condition,  $C$ , and its inverse,  $\neg C$ . These are the two conditions for an `if...else` statement. One condition,  $C$ , is stated in the `if` clause; the inversion condition,  $\neg C$ , is implied in the `else` clause.

This follows the **Law of the Excluded Middle**: we're claiming there's no missing alternative between the two conditions,  $C$  and  $\neg C$ . For a complex condition, though, this can be difficult to visualize.

If we have something like:

```
if weather == Weather.RAIN and plan == Plan.GO_OUT:
    bring("umbrella")
else:
    bring("sunglasses")
```

It may not be immediately obvious, but we've omitted a number of possible alternatives. The `weather` and `plan` variables have four different combinations of values. One of the conditions is stated explicitly, the other three are assumed:

- `weather == RAIN` and `plan == GO_OUT`. Bringing an umbrella seems right.
- `weather != RAIN` and `plan == GO_OUT`. Bringing sunglasses seems appropriate.
- `weather == RAIN` and `plan != GO_OUT`. If we're staying in, then neither accessory seems right.
- `weather != RAIN` and `plan != GO_OUT`. Again, the accessory question seems moot if we're not going out.

How can we be sure we haven't missed anything? How can we be sure we have not conflated too many things into a condition that's assumed instead of being stated?

## Getting ready

Let's look at a concrete example of an `if...elif` chain. In the casino game of **Craps**, there are a number of rules that apply to a roll of two dice. These rules apply on the first roll of the game, called the *come-out* roll:

- 2, 3, or 12 is *craps*, which is a loss for most bets.
- 7 or 11 is a winner for most bets.
- The remaining numbers establish a **point**. The dice-rolling continues based on another set of rules.

We'll use this set of three conditions as an example for looking at this recipe because it has a potentially vague clause in it.

## How to do it...

When we write an `if` statement, even when it appears trivial, we need to be sure that all conditions are covered.

1. Enumerate the conditions we know. In our example, we have three rules: the (2, 3, 12) rule, the (7, 11) rule, and a vague statement of “the remaining numbers.” This can form a first draft of an `if` statement.
2. Determine the universe of all possible alternatives. For this example, there are 11 alternative outcomes: the numbers from 2 to 12, inclusive.
3. Compare the various `if` and `elif` conditions,  $C$ , with the universe of alternatives,  $U$ . There are three possible design patterns:
  - We have more `if` conditions in the code than are possible in the universe of alternatives,  $C \subset U$ . The most common cause is failing to completely enumerate all possible alternatives in the universe. We might, for example, have modeled dice using 0 to 5 instead of 1 to 6. The universe of alternatives appears to be the values from 0 to 10, yet there are conditions for 11 and 12.
  - We have gaps in the conditions in our code,  $U \setminus C \neq \emptyset$ . The most common cause of alternatives in the universe without a clearly-stated `if` condition is failing to fully understand the conditions in the code. We might, for example, have enumerated the values as two tuples instead of sums. The numbers 2, 3 and 12 are defined by a number of pairs, including (1, 1), (1, 2), and (6, 6). It's



possible to overlook the condition (2, 1), leaving this untested by any clause of the `if` statement.

- We can prove there's a match between conditions expressed in the code and the universe of alternatives,  $U \equiv C$ . This is ideal. The universe of all possible alternatives matches all the conditions in the `if` and `elif` clauses of the statement.

In this example, it's easy to enumerate all of the possible alternatives. In other cases, it can take some careful reasoning to understand any gaps or omissions.

In this example, we have a vague term, **remaining numbers**, which we can replace with the list of values (4, 5, 6, 8, 9, 10). Supplying a list removes any possible gaps and doubts.

When there are exactly two alternatives, we can write a condition expression for one of the alternatives. The other condition can be implied; an `if` and `else` will work.

When we have more than two alternatives, we can use this recipe to write a chain of `if` and `elif` statements, one statement per alternative:

1. Write an `if ... elif ... elif` chain that covers all of the known alternatives. For our example, it might start like this:

```
dice = die_1 + die_2
if dice in (2, 3, 12):
    game.craps()
elif dice in (7, 11):
    game.winner()
elif dice in (4, 5, 6, 8, 9, 10):
    game.point(dice)
```

2. Add an `else` clause that raises an exception, like this:

```
else:
    raise Exception('Design Problem')
```

This extra `else` gives us a way to positively identify when a logic problem is found. We

can be sure that any design error we made will lead to a conspicuous problem when the program runs. Ideally, we'll find any problems while we're unit testing.

In this case, it is clear that all 11 alternatives are covered by the `if` statement conditions. The extra `else` can't ever be used. Not all real-world problems have this kind of easy proof that all the alternatives are covered by conditions. It can help to provide a noisy failure mode.

## How it works...

Our goal is to be sure that our program works reliably. While testing helps, we can still have the same wrong assumptions when doing design and creating test cases.

While rigorous logic is essential, we can still make mistakes. Further, someone doing ordinary software maintenance might introduce an error. Adding a new feature to a complex `if` statement is a potential source of problems.

This **Else-Raise** design pattern forces us to be explicit for each and every condition. Nothing is assumed. As we noted previously, any error in our logic will be uncovered if the exception gets raised.

Crashing with an exception is sensible behavior in the presence of a design problem. While an alternative is to write a message to an error log, a program with this kind of profound design flaw should be viewed as fatally broken.

## There's more...

In many cases, we can derive an `if...elif...elif` chain from an examination of the desired post condition at some point in the program's processing. For example, we may need a statement that establishes something like  $m$  is equal to the larger of  $a$  or  $b$ .

(For the sake of working through the logic, we'll avoid Python's handy `m = max(a, b)`, and focus on the way we can compute a result from exclusive choices.)

We can formalize the final condition like this:

$$(m = a \vee m = b) \wedge m \geq a \wedge m \geq b$$

We can work backward from this final condition, by writing the goal as an assert statement:

```
# do something
assert (m == a or m == b) and m >= a and m >= b
```

Once we have the goal stated, we can identify statements that lead to that goal. Clearly, assignment statements like  $m = a$  or  $m = b$  would be appropriate, but each of these works only under limited conditions.

We can derive the precondition that shows when these statements should be used. The preconditions for an assignment statement will be written in `if` and `elif` expressions.

We need to use the statement  $m = a$  when  $a \geq b$ . Similarly, we need to use the statement  $m = b$  when  $b \geq a$ . Rearranging logic into code gives us this:

```
if a >= b:
    m = a
elif b >= a:
    m = b
else:
    raise Exception('Design Problem')

assert (m == a or m == b) and m >= a and m >= b
```

Note that our universe of conditions,  $U = \{a \geq b, b \geq a\}$ , is complete; there's no other possible relationship. Also notice that in the edge case of  $a = b$ , we don't actually care which assignment statement is used. Python will process the decisions in order and will execute  $m = a$ . The fact that this choice is consistent shouldn't have any impact on our design of the `if...elif...elif` chain. We can design the conditions without regard to the order of evaluation of the clauses.

## See also

- This is somewhat similar to the syntactic problem of the “dangling else.” See <https://docs.oracle.com/javase/specs/jls/se9/html/jls-14.html>.

This isn’t the same problem; Python’s indentation removes the “dangling else” syntax problem. This is an adjacent semantic problem of trying to be sure that all conditions are properly accounted for in a complex `if...elif...elif` chain.

## Saving intermediate results with the := “walrus” operator

Sometimes we’ll have a complex condition where we want to preserve an expensive intermediate result for later use. Imagine a condition that involves a complex calculation; the cost of computing is high measured in time, input-output operations, memory resources, or all three.

An example includes doing repetitive searches using the Regular Expression (`re`) package. A `match()` method can do quite a bit of computation before returning either a `Match` object or a `None` object to show the pattern wasn’t found. Once this computation is completed, we may have several uses for the result, and we emphatically do *not* want to perform the computation again. Often, the initial use is the simple check to see if the result is a `Match` object or `None`.

This is an example where it can be helpful to assign a name to the value of an expression and also use the expression in an `if` statement. We’ll look at how to use the “assignment expression” or “walrus” operator. It’s called the walrus because the assignment expression operator, `:=`, looks like the face of a walrus to some people.

## Getting ready

Here’s a summation where—eventually—each term becomes so small that there’s no point in continuing to add it to the overall total:

$$\sum_{0 \leq n < \infty} \left( \frac{1}{2n+1} \right)^2$$

In effect, this is something like the following summation function:

```
>>> s = sum((1 / (2 * n + 1)) ** 2 for n in range(0, 20_000))
```

What's not clear is the question of how many terms are required. In the example, we've summed 20,000 values. But what if 16,000 are enough to provide an accurate answer?

We don't want to write a summation like this:

```
>>> b = 0
>>> for n in range(0, 20_000):
...     if (1 / (2 * n + 1)) ** 2 >= 0.000_000_001:
...         b = b + (1 / (2 * n + 1)) ** 2
```

This example repeats an expensive computation,  $(1/(2*n+1))**2$ . We can avoid processing that includes this kind of time-wasting overhead by using the walrus operator.

## How to do it...

1. First we isolate an expensive operation that's part of a conditional test. In this example, the variable `term` is used to hold the expensive result:

```
>>> p = 0
>>> for n in range(0, 20_000):
...     term = (1 / (2 * n + 1)) ** 2
...     if term >= 0.000_000_001:
...         p = p + term
```

2. Rewrite the assignment statement to use the `:=` assignment operator. This replaces the simple condition of the `if` statement.
3. Add an `else` condition to break out of the `for` statement if no more terms are needed.

Here's the results of these two steps:

```
>>> q = 0
>>> for n in range(0, 20_000):
...     if (term := (1 / (2 * n + 1)) ** 2) >= 0.000_000_001:
...         q = q + term
...     else:
...         break
```

Note that we changed the summation variable. In the previous step of the recipe, it was `p`. In this step, it's `q`. This permits easy side-by-side comparisons to be sure the results are still correct.

The assignment expression `:=` lets us do two things in the `if` statement.

## How it works...

The assignment expression operator `:=` saves an intermediate result. The operator's result value is the same as the right-hand side operand. This means that the expression `a + (b := c+d)` is the same as the expression `a+(c+d)`. The difference between the expression `a + (b := c+d)` and the expression `a+(c+d)` is the side-effect of setting the value of the `b` variable partway through the evaluation.

An assignment expression can be used in almost any kind of context where expressions are permitted in Python. The most common cases are `if` statements. Another good idea is inside a `while` condition.

They're also forbidden in a few places. They cannot be used as the operator in an expression statement. We're specifically prohibited from writing `a := 2` as a statement: there's a perfectly good assignment statement for this purpose and an assignment expression, while similar in intent, is potentially confusing.

## There's more...

We can do some more optimization of our infinite summation example, shown earlier in this recipe. The use of a `for` statement and a `range()` object seems simple. The problem is that we want to end the `for` statement early—when the terms being added are so small that they have no significant change in the final sum.

We can combine the early exit with the term computation:

```
>>> r = 0
>>> n = 0
>>> while (term := (1 / (2 * n + 1)) ** 2) >= 0.000_000_001:
...     r += term
...     n += 1
```

We've used a `while` statement with the assignment expression operator. This will compute a value using  $(1/(2*n+1))^{**2}$ , and assign this to the `term` variable. If the value is significant, we'll add it to the sum, `r`, and increment the value for the `n` variable. If the value assigned to `term` is too small to be significant, the `while` statement will end.

Here's another example, showing how to compute running sums of a collection of values. This looks forward to concepts in *Chapter 4*. Specifically, this shows a list comprehension built using the assignment expression operator:

```
>>> data = [11, 13, 17, 19, 23, 29]
>>> total = 0
>>> running_sum = [(total := total + d) for d in data]
>>> total
112
>>> running_sum
[11, 24, 41, 60, 83, 112]
```

We've started with some data, in the `data` variable. This might be minutes of exercise each day for most of a week. The value of the final `running_sum` variable is a list object, built by evaluating the expression `(total := total + d)` for each value, `d`, in the `data` variable. Because the assignment expression changes the value of the `total` variable, the resulting list is the result of each new value being accumulated.

## See also

- For details on assignment expression, see PEP-572, where the feature was first described.

## Avoiding a potential problem with break statements

The common way to understand a for statement is that it creates a *for all* condition. At the end of the statement, we can assert that, for all items in a collection, the processing in the body of the statement has been done.

This isn't the only meaning a for statement can have. When the break statement is used inside the body of a for statement, it changes the semantics to *there exists*. When the break statement leaves the for (or while) statement, we can assert there exists at least one item that caused the enclosing statement to end.

There's a side issue here. What if the for statement ends without executing the break statement? Either way, we're at the statement after the for statement. The condition that's true upon leaving a for or while statement with a break statement can be ambiguous. We can't *easily* tell; this recipe gives some design guidance.

The problem is magnified when we have multiple break statements, each with its own condition. How can we minimize the problems created by having these complicated conditions for leaving a for or while statement?

### Getting ready

When parsing configuration files, we often need to find the first occurrence of a : or = character in a string. The **property file** format uses a property name and : or = followed by a value.

Finding the punctuation mark is an example of a *there exists* modification to a for statement. We don't want to process *all* characters; we want to know where the leftmost : or = character is found.

Here's the sample data we're going use as an example:

```
>>> sample_1 = "some_name = the_value"
```



Here's a small for statement to locate the leftmost `:` or `=` character in the sample string value:

```
>>> for position in range(len(sample_1)):
...     if sample_1[position] in '=:':
...         break
>>> print(f"name={sample_1[:position]!r}",
...       f"value={sample_1[position+1:]!r}")
name='some_name ' value=' the_value'
```

When the `=` character is found, the `break` statement ends the `for` statement. The value of the `position` variable shows where the desired character was found.

What about the following edge case?

```
>>> sample_2 = "name_only"
>>> for position in range(len(sample_2)):
...     if sample_2[position] in '=:':
...         break
>>> print(f"name={sample_2[:position]!r}",
...       f"value={sample_2[position+1:]!r}")
name='name_onl' value=''
```

The result is awkwardly wrong: the `y` character got dropped from the value of `name`. Why did this happen? And, more importantly, how can we make the condition at the end of the `for` statement clearer?

## How to do it...

Every statement establishes a post-condition. When designing a `for` or `while` statement, we need to articulate the condition that should be true at the end of the statement. Ideally, the post-condition is something simple like `text[position] in '=:'`. However, in the case where there's no `=` or `:` in the given text, the overly simple post-condition can't be true.

At the end of the `for` statement, one of these two things are true:

- Either the character with the index of `position` is `:` or `=`

- Or all characters have been examined and no character is `:` or `=`

Our application code needs to handle both cases.

1. Write the obvious post-condition. We sometimes call this the *happy-path* condition because it's the one that's true when nothing unusual has happened:

```
assert text[position] in '=: ' # We found a = or :
```

2. Create the overall post-condition by adding the conditions for the edge cases. In this example, we have two additional conditions:

- There's no `=` or `:`.
- There are no characters at all. This means the `len()` is zero, and the `for` statement never actually did anything. This also means the `position` variable will never be created.

In this example, then, we have discovered a total of three conditions:

- `len(text) == 0`
  - `not('=' in text or ':' in text)`, which can be stated in a number of ways. `not(text[position] == ':' or text[position] == '=')` might be most clear.
  - `text[position] in '=: '`
3. A `while` statement can be redesigned to have the complete set of post conditions in the `while` clause. This can eliminate the need for a `break` statement. Proper initialization of variables is still required.
  4. When a `for` statement is being used, proper initialization of variables is required. Add `if` statements for the various terminating conditions after the body of the `for` statement. Here's the resulting `for` statement and a complicated `if` statement to examine all of the possible post conditions:

```

>>> position = -1
>>> for position in range(len(sample_2)):
...     if sample_2[position] in '=:':
...         break
...
>>> if position == -1:
...     print(f"name=None value=None")
... elif not(sample_2[position] == ':' or sample_2[position] == '='):
...     print(f"name={sample_2!r} value=None")
... else:
...     print(f"name={sample_2[:position]!r}",
...           f"value={sample_2[position+1:]!r}")
name='name_only' value=None

```

In the statements after the for statement, we've enumerated all of the terminating conditions explicitly.

## How it works...

This approach forces us to work out the post-condition carefully so that we can be absolutely sure that we know all the reasons the for or while statement ended.

The idea here is to forego any assumptions or intuition. With a little bit of discipline, we can be sure of the post-conditions. It's imperative to be explicit about the condition that's true when a statement works. This is the goal of our software, and we can work backward from the goal by choosing the simplest statements that will make the goal conditions true.

## There's more...

We can also use an else clause on a for statement to determine if the statement finished normally or a break statement was executed. We can use something like this:

```

>>> for position in range(len(sample_2)):
...     if sample_2[position] in '=:':
...         name, value = sample_2[:position], sample_2[position+1:]
...         break
...     else:
...         if len(sample_2) > 0:

```

```
...     name, value = sample_2, None
...     else:
...         name, value = None, None
>>> print(f"{name=!r} {value=!r}")
name='name_only' value=None
```

Using an else clause in a for statement is sometimes confusing, and we don't recommend it. It's not clear if this version is substantially better than any of the alternatives. It's too easy to forget the reason why the else is executed because it's used so rarely.

## See also

- A classic article on this topic is by David Gries, *A note on a standard strategy for developing loop invariants and loops*. See <http://www.sciencedirect.com/science/article/pii/0167642383900151>

## Leveraging exception matching rules

The try statement lets us capture an exception. When an exception is raised, we have a number of choices for handling it:

- **Ignore it:** If we do nothing, the program stops. We can do this in two ways—don't use a try statement in the first place, or don't have a matching except clause in the try statement.
- **Log it:** We can write a message and use a raise statement to let the exception propagate after writing to a log. The expectation is that this will stop the program.
- **Recover from it:** We can write an except clause to do some recovery action to undo any effects of the partially completed try clause.
- **Silence it:** If we do nothing (that is, use the pass statement), then processing is resumed after the try statement. This silences the exception, but does not correct the underlying problem, or supply alternative results as a recovery attempt.
- **Rewrite it:** We can raise a different exception. The original exception becomes a

context for the newly raised exception.

What about nested contexts? In this case, an exception could be ignored by an inner try but handled by an outer context. The basic set of options for each try context is the same. The overall behavior of the software depends on the nested definitions.

The design of a try statement depends on the way that Python exceptions form a class hierarchy. For details, see the *Exception hierarchy* section of *Python Standard Library*. For example, the `ZeroDivisionError` exception is also an `ArithmeticError` and an `Exception`. For another example, the `FileNotFoundError` exception is also an `OSError` as well as an `Exception`.

This hierarchy can lead to confusion if we're trying to handle detailed exceptions as well as generic exceptions.

## Getting ready

Let's say we're going to make use of the `shutil` module to copy a file from one place to another. Most of the exceptions that might be raised indicate a problem too serious to work around. However, in the specific event of a `FileNotFoundError` exception, we'd like to attempt a recovery action.

Here's a rough outline of what we'd like to do:

```
>>> from pathlib import Path
>>> import shutil

>>> source_dir = Path.cwd()/"data"
>>> target_dir = Path.cwd()/"backup"
>>> for source_path in source_dir.glob('**/*.csv'):
...     source_name = source_path.relative_to(source_dir)
...     target_path = target_dir / source_name
...     shutil.copy(source_path, target_path)
```

We have two directory paths, `source_dir` and `target_dir`. We've used the `glob()` method to locate all of the files under `source_dir` that have `*.csv` files.

The expression `source_path.relative_to(source_dir)` gives us the tail end of the filename, the portion after the directory. We use this to build a new, similar path under the `target_dir` directory. This assures that a file named `wc1.csv` in the `source_dir` directory will have a similar name in the `target_dir` directory.

The problems arise with handling exceptions raised by the `shutil.copy()` function. We need a `try` statement so that we can recover from certain kinds of errors. We'll see this kind of error if we try to run this:

```
Traceback (most recent call last):
...
FileNotFoundError: [Errno 2] No such file or directory: ...
```

(We've replaced some details with `...` because they'll be different on your computer.)

This exception is raised when the backup directory hasn't been created. It will also happen when there are subdirectories inside the `source_dir` directory tree that don't also exist in the `target_dir` tree. How do we create a `try` statement that handles these exceptions and creates the missing directories?

## How to do it...

1. Write the code we want to use indented in the `try` block:

```
>>> try:
...     shutil.copy(source_path, target_path)
```

2. Include the most specific exception classes first in an `except` clause. In this case, we have a meaningful response to the specific `FileNotFoundError` exception.
3. Include any more general exceptions later. In this case, we'll report any generic `OSError` exception that's encountered. This leads to the following:

```
>>> try:
...     target = shutil.copy(source_path, target_path)
... except FileNotFoundError:
```

```
...     target_path.mkdir(exist_ok=True, parents=True)
...     target = shutil.copy(source_path, target_path)
...     except OSError as ex:
...         print(f"Copy {source_path} to {target_path} error {ex}")
```

We've matched exceptions with the most specific first and the more generic after that.

We handled the `FileNotFoundError` exception by creating the missing directories. Then we tried the `copy()` again, knowing it would now work properly.

We logged any other exceptions of the class `OSError`. For example, if there's a permission problem, that error will be written to a log and the next file will be tried. Our objective is to try and copy all of the files. Any files that cause problems will be logged, but the overall copying process will continue.

And, yes, the line of code to copy the files is repeated in two distinct contexts. The first repetition is when there has been no error. The second is after attempted recovery from the initial error. To an extent, this feels like breaking the **Don't Repeat Yourself** principle. Let's look at the alternative, which doesn't seem as good.

To meet the **DRY** standard, we could try to nest this operation in a `for` statement. The `break` statement is used if things work, otherwise, multiple attempts can be made. The extra complication of the `for` statement seems to be worse than the repetition.

A common compromise is to write a one-line function that reduces the repetition to the name of the function. This has the advantage of making it possible to change to another of the `shutil` copy functions in one place.

## How it works...

Python's matching rules for exceptions are intended to be simple:

- Process `except` clauses in order.
- Match the actual exception against the exception class (or tuple of exception classes). A match means that the actual exception object (or any of the base classes of the

exception object) is of the given class in the except clause.

These rules show why we put the most specific exception classes first and the more general exception classes last. A generic exception class like `Exception` will match almost every kind of exception. We don't want this first, because no other clauses will be checked.

There's an even more generic class, the `BaseException` class. There's no good reason to ever handle exceptions of this class. If we do, we will be catching `SystemExit` and `KeyboardInterrupt` exceptions; this interferes with the ability to kill a misbehaving application. We only use the `BaseException` class as a superclass when defining new exception classes that exist outside the normal exception hierarchy.

## There's more...

Our example includes a nested context in which a second exception can be raised. Consider this except clause snippet (taken out of context):

```
... except FileNotFoundError:
...     target_path.parent.mkdir(exist_ok=True, parents=True)
...     target = shutil.copy(source_path, target_path)
```

If the `mkdir()` method or `shutil.copy()` functions actually raise exceptions while handling the original `FileNotFoundError` exception, it won't be handled. Any exceptions raised within an except clause can crash the program as a whole. Handling these nested exceptions can involve nested try statements.

We can rewrite the exception clause to include a nested try during recovery:

```
>>> try:
...     target = shutil.copy(source_path, target_path )
... except FileNotFoundError:
...     try:
...         target_path.parent.mkdir(exist_ok=True, parents=True)
...         target = shutil.copy(source_path, target_path)
...     except OSError as ex2:
...         print(f"{target_path.parent} problem: {ex2}")
```



```
... except OSError as ex:  
...     print(f"Copy {source_path} to {target_path} error {ex}")
```

In this example, a nested context writes one message for an `OSError` exception. In the outer context, a slightly different error message is used to log a similar error. In both cases, processing can continue. The distinct error messages can make it slightly easier to debug the problems.

## See also

- In the *Avoiding a potential problem with an `except:` clause* recipe, we look at some additional considerations when designing exception handling statements.

## Avoiding a potential problem with an `except:` clause

There are some common mistakes in exception handling. These can cause programs to become unresponsive.

One of the mistakes we can make is to use the `except:` clause with no named exception class to match. There are a few other mistakes that we can make if we're not cautious about the exceptions we try to handle.

This recipe will show some common exception handling errors that we can avoid.

## Getting ready

When code can raise a variety of exceptions, it's sometimes tempting to try and match as many as possible. Matching too many exception classes can interfere with stopping a misbehaving Python program. We'll extend the idea of *what not to do* in this recipe.

## How to do it...

We need to avoid using the bare `except:` clause. Instead, use `except Exception:` to match the most general kind of exception that an application can reasonably handle.

Handling too many exception classes can interfere with our ability to stop a misbehaving Python program. When we hit *Ctrl + C*, or send a `SIGINT` signal via the OS's `kill -2` command, we generally want the program to stop. We rarely want the program to write a message and keep running. If we use a bare `except:` clause, we can accidentally silence important exceptions.

There are a few other classes of exceptions that we should be wary of attempting to handle:

- `SystemError`
- `RuntimeError`
- `MemoryError`

Generally, these exceptions mean things are going badly somewhere in Python's internals. Rather than silence these exceptions, or attempt some recovery, we should allow the program to fail, find the root cause, and fix it.

Further, if we capture any of these exceptions, we can interfere with the way these internal exceptions are handled:

- `SystemExit`
- `KeyboardInterrupt`
- `GeneratorExit`

Trying to handle these exceptions can cause a program to become unresponsive at exactly the time we need to stop it.

## How it works...

There are three techniques we should avoid:

- Don't match the `BaseException` class in an `except BaseException:` clause.
- Don't use `except:` with no exception class. This matches all exceptions, including exceptions we should avoid trying to handle.

- Don't match exceptions from which there's no sensible recovery.

If we handle too many kinds of exceptions, we may exacerbate a problem, transforming it into a larger and more mysterious problem by way of flawed exception handling.

It's a noble aspiration to write a program that never crashes. Interfering with some of Python's internal exceptions, however, doesn't create a more reliable program. Instead, it creates a program where a clear failure is masked and made into an obscure problem.

## See also

- In the *Leveraging exception matching rules* recipe, we look at some considerations when designing exception-handling statements.

## Concealing an exception root cause

Exceptions contain a root cause. The default behavior of internally raised exceptions is to use an implicit `__context__` attribute to include the root cause of an exception. In some cases, we may want to deemphasize the root cause because it's misleading or unhelpful for debugging.

This technique is almost always paired with an application or library that defines a unique exception. The idea is to show the unique exception without the clutter of an irrelevant exception from outside the application or library.

## Getting ready

Assume we're writing some complex string processing. We'd like to treat a number of different kinds of detailed exceptions as a single generic error so that users of our software are insulated from the implementation details. We can attach details to the generic error.

## How to do it...

1. To create a new exception, we can do this:

```
>>> class MyAppError(Exception):  
...     pass
```

This creates a new, unique class of exception that our library or application can use.

2. When handling exceptions, we can conceal the root cause exception like this:

```
>>> try:  
...     None.some_method(42) # Raises an exception  
... except AttributeError as exception:  
...     raise MyAppError("Some Known Problem") from None
```

In this example, we raise a new instance of the module's unique `MyAppError` exception class. The new exception will not have any connection with the root cause `AttributeError` exception.

## How it works...

The Python exception classes all have a place to record the cause of the exception. We can set this `__cause__` attribute using the `raise Visible from RootCause` statement. This is done implicitly using the exception context.

Here's how it looks when this exception is raised:

```
>>> try:  
...     None.some_method(42)  
... except AttributeError as exception:  
...     raise MyAppError("Some Known Problem") from None  
Traceback (most recent call last):  
...  
MyAppError: Some Known Problem
```

The underlying cause has been concealed. If we omit the `from None` in the `raise` statement, then the exception will include two parts and will be quite a bit more complex. When the root cause is shown, the output looks more like this:

```

Traceback (most recent call last):
  File "<doctest recipe_09.txt[3]>", line 2, in <module>
    None.some_method(42)
    ^^^^^^^^^^^^^^^^^^^
AttributeError: 'NoneType' object has no attribute 'some_method'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File ...
    exec...
  File ...
    raise...
MyAppError: Some Known Problem

```

This shows the underlying `AttributeError` exception. This may be an implementation detail that's unhelpful and better left off the printed display of the exception.

The more useful part of the exception (with some details replaced by `...`) follows the initial (and possibly irrelevant) root cause information.

## There's more...

There are a number of internal attributes of an exception. These include `__cause__`, `__context__`, `__traceback__`, and `__suppress_context__`. The overall exception context is in the `__context__` attribute. The cause, if provided via a `raise` from statement, is in `__cause__`. The context for the exception is available but can be suppressed from being printed.

## See also

- In the *Leveraging exception matching rules* recipe, we look at some considerations when designing exception-handling statements.
- In the *Avoiding a potential problem with an `except: clause`* recipe, we look at some additional considerations when designing exception-handling statements.

## Managing a context using the with statement

There are many instances where our scripts will be entangled with external resources. The most common examples are disk files and network connections to external hosts. A common bug is retaining these entanglements forever, tying up these resources uselessly. These are sometimes called a memory **leak** because the available memory is reduced each time a new file is opened without closing a previously used file.

We'd like to isolate each entanglement so that we can be sure that the resource is acquired and released properly. The idea is to create a context in which our script uses an external resource. At the end of the context, our program is no longer bound to the resource and we want to be guaranteed that the resource is released.

### Getting ready

Let's say we want to write lines of data to a file in CSV format. When we're done, we want to be sure that the file is closed and the various OS resources—including buffers and file handles—are released. We can do this in a context manager, which guarantees that the file will be properly closed.

Since we'll be working with CSV files, we can use the `csv` module to handle the details of the formatting:

```
>>> import csv
```

We'll also use the `pathlib` module to locate the files we'll be working with:

```
>>> from pathlib import Path
```

For the purposes of having something to write, we'll use this silly data source:

```
>>> some_source = [  
...     [2,3,5],  
...     [7,11,13],
```

```
... [17,19,23]]
```

We'll also need a working directory. In the examples, we're using data under the current working directory. We can create this directory using a terminal window command, or we can create it from within Python:

```
>>> Path.cwd().mkdir("data", exists_ok=True)
```

This will give us a context in which to learn about the with statement.

## How to do it...

1. Create the context by opening the Path, or creating the network connection with `urllib.request.urlopen()`. Other common contexts include creating archives like zip files and tar files. Here's the essential context creation for an open file:

```
>>> target_path = Path.cwd() / "data" / "test.csv"
>>> with target_path.open('w', newline='') as target_file:
```

2. Include all the processing, indented within the with statement:

```
>>> target_path = Path.cwd() / "data" / "test.csv"
>>> with target_path.open('w', newline='') as target_file:
...     writer = csv.writer(target_file)
...     writer.writerow(['column', 'data', 'heading'])
...     writer.writerows(some_source)
```

3. When we use a file as a context manager, the file is automatically closed at the end of the indented context block. Even if an exception is raised, the file is still closed properly. Outdent the processing that is done after the context is finished and the resources are released:

```
>>> target_path = Path.cwd() / "data" / "test.csv"
>>> with target_path.open('w', newline='') as target_file:
...     writer = csv.writer(target_file)
```

```
...     _ = writer.writerow(['column', 'data', 'heading'])
...     writer.writerows(some_source)
>>> print(f'finished writing {target_path.name}')
finished writing test.csv
```

The statements outside the `with` context will be executed after the context is closed. The named resource — the file opened by `target_path.open()` — will be properly closed.

(We assign the result of the `writerow()` method of a writer to the `_` variable. This is a trick required to avoid showing this result. It's the number 21, telling us how many characters were written.)

Even if an exception is raised inside the `with` context, the file is still properly closed. The context manager is notified of the exception. It can close the file and allow the exception to propagate.

## How it works...

A context manager is notified of three significant events surrounding the indented block of code:

- Entry to the context
- Normal exit from the context with no exception
- Exit from the context because of an exception

The context manager will—under all conditions—disentangle our program from external resources. Files can be closed. Network connections can be dropped. Database transactions can be committed or rolled back. Locks can be released.

We can experiment with this by including a manual exception inside the `with` statement. This can show that the file was properly closed:



```
>>> try:
...     with target_path.open('w', newline='') as target_file:
...         writer = csv.writer(target_file)
...         _ = writer.writerow(['column', 'data', 'heading'])
...         _ = writer.writerow(some_source[0])
...         raise Exception("Testing")
... except Exception as exc:
...     print(f"{target_file.closed}")
...     print(f"{exc}")
target_file.closed=True
exc=Exception('Testing')
>>> print(f"finished writing {target_path.name}")
finished writing test.csv
```

In this example, we've wrapped the real work in a `try` statement. This allows us to raise an exception after writing the first line of data to the CSV file. Because the exception handling is outside the `with` context, the file is closed properly. All resources are released and the part that was written is properly accessible and usable by other programs.

The output confirms the expected file state:

```
target_file.closed=True
exc=Exception('Testing')
```

This shows us that the file was properly closed. It also shows us the message associated with the exception to confirm that it was the exception we raised manually. This kind of technique allows us to work with expensive resources like database connections and network connections and be sure these don't "leak."

Resource leak is a common description used when resources are not released properly back to the OS. It's as if a pool is slowly drained away, and the application stops working because there are no more available OS network sockets or file handles. The `with` statement can be used to properly disentangle our Python application from OS resources.

## There's more...

Python offers us a number of context managers. We noted that an open file is a context, as is an open network connection created by `urllib.request.urlopen()`.

For all file operations, and all network connections, we should always use a `with` statement as a context manager. It's very difficult to find an exception to this rule.

It turns out that the `decimal` module makes use of a context manager to allow localized changes to the way decimal arithmetic is performed. We can use the `decimal.localcontext()` function as a context manager to change rounding rules or precision for calculations isolated by a `with` statement.

We can define our own context managers, also. The `contextlib` module contains functions and decorators that can help us create context managers around resources that don't explicitly offer them.

When working with locks, the `with` statement context manager is the ideal way to acquire and release a lock. See <https://docs.python.org/3/library/threading.html#with-locks> for the relationship between a lock object created by the `threading` module and a context manager.

## See also

- See PEP-343 for the origins of the `with` statement.
- Numerous recipes in *Chapter 9*, will make use of this technique. The recipes *Reading delimited files with the CSV module*, *Reading complex formats using regular expressions*, and *Reading HTML documents*, among others, will make use of the `with` statement.

## Join our community **Discord** space

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

<https://packt.link/dHrHU>



# 3

## Function Definitions

Function definitions are a way to decompose a large problem into smaller problems. Mathematicians have been doing this for centuries. It's a way to package our Python programming into intellectually manageable chunks.

We'll look at a number of function definition techniques in these recipes. This will include ways to handle flexible parameters and ways to organize the parameters based on some higher-level design principles.

We'll also look at the `typing` module and how we can create more formal type hints for our functions. Using type hints will prepare our code so we can use tools like **mypy** to confirm the data types are used properly throughout a program. Type hints aren't required, but they often identify potential inconsistencies, allowing us to write code that prevents problems.

In this chapter, we'll look at the following recipes:

- *Function parameters and type hints*
- *Designing functions with optional parameters*

- *Using super flexible keyword parameters*
- *Forcing keyword-only arguments with the \* separator*
- *Defining position-only parameters with the / separator*
- *Picking an order for parameters based on partial functions*
- *Writing clear documentation strings with RST markup*
- *Designing recursive functions around Python's stack limits*
- *Writing testable scripts with the script-library switch*

## Function parameters and type hints

Through a number of Python Enhancement Proposals, type hints have grown in sophistication. The **mypy** tool is one way to validate these type hints to be sure the hints and the code agree. All the examples shown in this book have been checked with the **mypy** tool.

This extra syntax for the hints is optional. It has limited use at runtime and has no performance costs.

### Getting ready

We'll need to download and install the **mypy** tool. Generally, this is done with the following terminal command:

```
(cookbook3) % python -m pip install mypy
```

Using the `python -m pip` command ensures the `pip` command will be associated with the currently active virtual environment. In this example, the prompt shows a virtual environment named `cookbook3`.

We can also use the **pyright** tool to examine type hints.

For an example of type hints, we'll look at some color computations. The first of these is extracting the Red, Green, and Blue values from the color codes commonly used in the

style sheets for HTML pages. There are a variety of ways of encoding the values, including strings, integers, and tuples. Here are some of the varieties of data types:

- A string of six hexadecimal characters with a leading # , for example, "#C62D42"
- A string of six hexadecimal characters, for example, "C62D42"
- A Python numeric value, for example, 0xC62D42
- A three-tuple of R, G, and B integers, for example, (198, 45, 66)

For strings and numbers, we use the type name directly, `str` or `int`. For tuples, we use a more complicated-looking `tuple[int, int, int]`.

The target is three integer values. A conversion from string or integer to three values involves two separate steps:

1. If the value is a string, convert to a single integer using the `int()` function.
2. For single integer values, split the integer into three separate values using the `>>` and `&` operators. This is the core computation for converting a single integer value, `hx_int`, into three separate `r`, `g`, `b` values:

```
r, g, b = (hx_int >> 16) & 0xFF, (hx_int >> 8) & 0xFF, hx_int & 0xFF
```

A single RGB integer has three separate values that are combined via bit shifting. The red value was shifted left 16 bits. To extract this component, the value is shifted right 16 bits using the `>>` operator. The `&` operator applies `0xff` as a “mask” to save only 8 bits of a potentially larger number. To extract the green component, shift right 8 bits. The blue value occupies the least-significant 8 bits.

## How to do it...

For some functions, it can be easiest to start with a working implementation and add hints. Here’s how it works:

1. Write the function without any hints:

```

def hex2rgb_1(hx_int):
    if isinstance(hx_int, str):
        if hx_int[0] == "#":
            hx_int = int(hx_int[1:], 16)
        else:
            hx_int = int(hx_int, 16)
    r, g, b = (hx_int >> 16) & 0xff, (hx_int >> 8) & 0xff, hx_int &
    0xff
    return r, g, b

```

2. Add the result hint. It's based on the return statement. In this example, the return is a tuple of three integers, `tuple[int, int, int]`.
3. Add the parameter hints. In this case, we've got two alternative types for the parameter: it can be a string or an integer. In the formal language of the type hints, this is a union of two types. The parameter can be described as `Union[str, int]` or `str | int`. If `Union` is used, the definition must be imported from the typing module.

Combining the hints into a function leads to the following definition:

```

def hex2rgb(hx_int: int | str) -> tuple[int, int, int]:
    if isinstance(hx_int, str):
        if hx_int[0] == "#":
            hx_int = int(hx_int[1:], 16)
        else:
            hx_int = int(hx_int, 16)
    r, g, b = (hx_int >> 16) & 0xff, (hx_int >> 8) & 0xff, hx_int & 0xff
    return r, g, b

```

## How it works...

These type hints have no impact when the Python code is executed. The hints are designed for people to read and for external tools, like **mypy**, to verify. A tool can confirm that the `hx_int` variable is always used as either an integer or a string.

In the `r, g, b =` assignment statement, the value for `hx_int` is expected to be an integer.

The **mypy** tool can confirm the operators are appropriate for integer values, and the return type matches the computed types.

We can observe the **mypy** tool's analysis of a type by inserting the `reveal_type(hx_int)` function into our code. This statement has function-like syntax; it's only used when running the **mypy** tool. We will only see output from this when we run **mypy**, and we have to remove this extra line of code before we try to do anything else with the module.

The output looks like this when we run **mypy** at the shell prompt on the `recipe_01_reveal.py` file:

```
(cookbook3) % mypy src/ch03/recipe_01_reveal.py
src/ch03/recipe_01_reveal.py:15: note: Revealed type is "builtins.int"
Success: no issues found in 1 source file
```

The output from the `reveal_type(hx_int)` line tells us **mypy** is certain the variable will have an integer value after the first `if` statement is complete. Once we've seen the revealed type information, we need to delete the `reveal_type(hx_int)` line from the file.

## There's more...

Let's look at a related computation. This converts RGB numbers into **Hue-Saturation-Lightness (HSL)** values. These HSL values can be used to compute complementary colors. An additional algorithm required to convert from HSL back into RGB values can help encode colors for a web page:

- RGB to HSL: We'll look at this closely because it has complex type hints.
- HSL to complement: There are a number of theories on what the "best" complement might be. We'll gloss over the details.
- HSL to RGB: This will be the final step, but we'll ignore the details of this computation.

We won't look closely at two of the implementations. They are not horribly complicated, but these computation details can be a distraction from understanding the types and type hints. See <https://www.easyrgb.com/en/math.php>.



We start by roughing out a definition of the function with a stub definition, like this:

```
def rgb_to_hsl_t(rgb: tuple[int, int, int]) -> tuple[float, float, float]:  
    ...
```

This can help us visualize a number of related functions to be sure they all have consistent types. The other two functions have stubs like these:

```
def hsl_comp_t(hsl: tuple[float, float, float]) -> tuple[float, float,  
float]:  
    ...
```

```
def hsl_to_rgb_t(hsl: tuple[float, float, float]) -> tuple[int, int, int]:  
    ...
```

After writing down this initial list of stub definitions, we can see some type hints are repeated in slightly different contexts. This suggests we need to create a separate named type to avoid repetition of the details. We'll provide a name for the repeated type detail:

```
from typing import TypeAlias  
  
RGB_a: TypeAlias = tuple[int, int, int]  
  
HSL_a: TypeAlias = tuple[float, float, float]  
  
def rgb_to_hsl(color: RGB_a) -> HSL_a:  
    ...  
  
def hsl_complement(color: HSL_a) -> HSL_a:  
    ...  
  
def hsl_to_rgb(color: HSL_a) -> RGB_a:  
    ...
```

This overview of the various functions can be helpful for assuring that each function uses

data in a way that's consistent with other functions.

The names `RGB_a` and `HSL_a` include a suffix of `_a` to help distinguish these type aliases from other examples in this recipe. In a practical application, the suffix strings like `_a` to show the name is an alias are going to become visual clutter and should be avoided.

As noted in the *Using NamedTuples to simplify item access in tuples* in Chapter 1, we can provide a more descriptive set of names for these tuple types:

```
from typing import NamedTuple

class RGB(NamedTuple):
    red: int
    green: int
    blue: int
```

We've defined a unique, new `NamedTuple` subclass, called `RGB`. Using names can help clarify the intent behind the code.

## See also

- The **mypy** project contains a wealth of information. See <https://mypy.readthedocs.io> for more information on the way type hints work.
- The **pyright** project is another helpful type hint tool. See <https://microsoft.github.io/pyright> for more information.

## Designing functions with optional parameters

When we define a function, we often have a need for optional parameters. This allows us to write functions that are more flexible and easier to read.

We can also think of this as a way to create a family of closely related functions. Each function has a slightly different collection of parameters – called the **signature** – but all sharing the same simple name. This is sometimes called an “overloaded” function. Within the `typing` module, an `@overload` decorator can help create type hints in the more complicated cases.

An example of an optional parameter is the built-in `int()` function. This function has two signatures:

- `int(str) -> int`. For example, the value of `int('355')` has a value of 355. An optional base parameter defaults to a value of 10.
- `int(str, base) -> int`. For example, the value of `int('163', 16)` is 355. In this case, the base parameter value is 16.

## Getting ready

A great many games rely on collections of dice. The casino game of *Craps* uses two dice. A game like *Zonk* (or *Greed* or *Ten Thousand*) uses six dice. It's handy to have a dice-rolling function that can handle all of these variations.

## How to do it...

We have two approaches to designing a function with optional parameters:

- **General to particular:** Start by designing the most general solution and provide defaults for the most common case.
- **Particular to general:** Start by designing several related functions. We then merge them into one general function that covers all of the cases, singling out one of the original functions to be the default behavior.

We'll look at the **particular to general** approach first, because it's often easier to start with a number of concrete examples.

### Particular to general design

Throughout this example, we'll use slightly different names as the function evolves. This simplifies unit testing the different versions and comparing them. Here's how we'll proceed:

1. Write one game function. We'll start with the *Craps* game because it seems to be the simplest:

```
import random

def die() -> int:
    return random.randint(1, 6)

def craps() -> tuple[int, int]:
    return (die(), die())
```

We defined a function, `die()`, to encapsulate a basic fact about standard dice. Five platonic solids are often used, yielding four-sided, six-sided, eight-sided, twelve-sided, and twenty-sided dice. The `randint()` expression assumes a six-sided cube.

2. Write the next game function. We'll move on to the *Zonk* game:

```
def zonk() -> tuple[int, ...]:
    return tuple(die() for x in range(6))
```

We've used a generator expression to create a tuple object with a collection of six dice. We'll look at generator expressions in depth in *Chapter 9*.

The generator expression in the body of the `zonk()` function has a variable, `x`, which is required syntax, but the value is ignored. It's also common to see this written as `tuple(die() for _ in range(6))`. The variable `_` is a valid Python variable name, often used when a variable name is required, but is never used.

3. Locate the common features in the `craps()` and `zonk()` functions. In this case, we can refactor the design of the `craps()` function to follow the pattern set by the `zonk()` function. Rather than building exactly two evaluations of the `die()` function, we can introduce a generator expression based on `range(2)` that will evaluate the `die()` function twice:

```
def craps_v2() -> tuple[int, ...]:
    return tuple(die() for x in range(2))
```

Merge the two functions. This will often involve exposing a variable that had

previously been a literal value:

```
def dice_v2(n: int) -> tuple[int, ...]:  
    return tuple(die() for x in range(n))
```

This provides a general function that covers the needs of both the *Craps* and *Zonk* games.

4. Identify the most common use case and make this the default value for any parameters that were introduced. If our most common simulation was *Craps*, we might do this:

```
def dice_v3(n: int = 2) -> tuple[int, ...]:  
    return tuple(die() for x in range(n))
```

Now, we can use `dice_v3()` for the *Craps* game. We'll need to use the expression `dice_v3(6)` for the first roll of a *Zonk* game.

5. Check the type hints to be sure they describe the parameters and the return values. In this case, we have one parameter with an integer value, and the return is a tuple of integers, described by `tuple[int, ...]`.

Throughout this example, the name evolved from `dice()` to `dice_v2()` and then to `dice_v3()`. This can make it easier to see the differences here in the recipe. Once a final version is written, it makes sense to delete the others and rename the final versions of these functions to `dice()`, `craps()`, and `zonk()`. The story of their evolution may become a blog post, but it doesn't need to be preserved in the code.

## General to particular design

When following the **general to particular** strategy, we'll identify all of the needs first. It can be difficult to foresee all the alternatives, making this more challenging. We'll often do this by introducing variables to the requirements:

1. Summarize the requirements for dice-rolling. We might start with a list like this:
  - *Craps*: Two dice

- First roll in *Zonk*: Six dice
  - Subsequent rolls in *Zonk*: One to six dice
2. Rewrite the requirements with an explicit parameter in place of any literal value. We'll replace all of our numbers with a parameter,  $n$ . This will take on values of 2, 6, or a value in the range  $1 \leq n \leq 6$ . We want to be sure we've properly parameterized each of the various functions.
  3. Write the function that fits the general pattern:

```
def dice_d1(n):  
    return tuple(die() for x in range(n))
```

In the third case – subsequent rolls in *Zonk* – we identified a constraint of  $1 \leq n \leq 6$ , imposed by the application program to play *Zonk*.

4. Provide a default value for the most common use case. If our most common simulation was *Craps*, we might do this:

```
def dice_d2(n=2):  
    return tuple(die() for x in range(n))
```

5. Add type hints. These will describe the parameters and the return values. In this case, we have one parameter with an integer value, and the return is a tuple of integers, described by `tuple[int, ...]`:

```
def dice(n: int=2) -> tuple[int, ...]:  
    return tuple(die() for x in range(n))
```

Now, we can use this `dice()` function for *Craps*. We'll need to use `dice(6)` for the first roll in *Zonk*.

In this recipe, the name didn't need to evolve through multiple versions. The name evolution is only useful in a book for unit testing each example.

This version looks precisely like `dice_v2()` from the previous recipe. This isn't an accident

– the two design strategies often converge on a common solution.

## How it works...

Python's rules for providing parameter values enable several ways to ensure that each parameter is given an argument value. We can think of the process like this:

1. Where there are default values, set those parameters. Default values make these optional.
2. For arguments without names – for example, `dice(2)` – the argument values are assigned to the parameters by position.
3. For arguments with names – for example, `dice(n=2)` – the argument values are assigned to parameters by name.
4. If any parameter still lacks a value, raise a `TypeError` exception.

The rules also allow us to mix positional values with named values. This makes some parameters optional by providing a default value.

## There's more...

It helps to write functions that are specialized versions of our more generalized function. These functions can simplify an application:

```
def craps_v3():  
    return dice(2)  
  
def zork_v3():  
    return dice(6)
```

Our application features – `craps_v3()` and `zork_v3()` – depend on a general function, `dice()`.

These form layers of dependencies, saving us from having to understand too many details. This idea of layered abstractions is sometimes called **chunking**, a way of managing complexity by isolating the details.

## See also

- We'll extend on some of these ideas in the *Picking an order for parameters based on partial functions* recipe, later in this chapter.
- We've made use of optional parameters that involve immutable objects. In this recipe, we focused on numbers. In *Chapter 4*, we'll look at mutable objects, which have an internal state that can be changed. In the *Avoiding mutable default values for function parameters* recipe, we'll look at some additional considerations for optional values.

## Using super flexible keyword parameters

Some design problems involve solving a simple equation for one unknown when given enough known values. For example, rate, time, and distance have a simple linear relationship. We can solve for any one when given the other two.

There are three related solutions to  $r \times t = d$ :

- $d = r \times t$
- $r = \frac{d}{t}$
- $t = \frac{d}{r}$

When designing electrical circuits, for example, a similar set of equations is used based on Ohm's law. In that case, the equations tie together resistance, current, and voltage.

In some cases, we want an implementation that can perform any of the three different calculations based on what's known and what's unknown.

## Getting ready

We'll build a single function that can solve a **Rate-Time-Distance (RTD)** calculation by embodying all three solutions, given any two known values. With minor variable name changes, this applies to a surprising number of real-world problems.

We don't necessarily want a single value as an answer. We can slightly generalize this



by creating a small Python dictionary with the three values in it; two are given, one is computed. We'll look at dictionaries in more detail in *Chapter 5*.

We'll use the warnings module instead of raising an exception when there's a problem:

```
import warnings
```

Sometimes, it is more helpful to produce a result that is doubtful than to stop processing.

## How to do it...

1. Solve the equation for each of the unknowns. There are three separate expressions:

- `distance = rate * time`
- `rate = distance / time`
- `time = distance / rate`

2. Wrap each expression in an if statement based on one of the values being None when it's unknown:

```
if distance is None:
    distance = rate * time

elif rate is None:
    rate = distance / time

elif time is None:
    time = distance / rate
```

3. Refer to the *Designing complex if...elif chains* recipe from *Chapter 2*, for guidance on designing these complex if...elif chains. Include a variation of the **Else-Raise** option:

```
else:
    warnings.warning("Nothing to solve for")
```

4. Build the resulting dictionary object:

```
return dict(distance=distance, rate=rate, time=time)
```

5. Wrap all of this as a function using keyword parameters with default values of `None`. This leads to parameter types of `Optional[float]`, often stated as `float | None`. The return type is a dictionary with string keys, summarized as `dict[str, float | None]`. It looks like this:

```
def rtd(
    distance: float | None = None,
    rate: float | None = None,
    time: float | None = None,
) -> dict[str, float | None]:

    if distance is None and rate is not None and time is not None:
        distance = rate * time
    elif rate is None and distance is not None and time is not None:
        rate = distance / time
    elif time is None and distance is not None and rate is not None:
        time = distance / rate
    else:
        warnings.warn("Nothing to solve for")

    return dict(distance=distance, rate=rate, time=time)
```

The type hints tend to make the function definition so long it has to be spread across five physical lines of code. The presence of so many optional values is difficult to summarize!

We can use the resulting function like this:

```
>>> rtd(distance=31.2, rate=6)
{'distance': 31.2, 'rate': 6, 'time': 5.2}
```

This shows us that going 31.2 nautical miles at a rate of 6 knots will take 5.2 hours.

For a nicely formatted output, we might do this:

```
>>> result = rtd(distance=31.2, rate=6)

>>> ('At {rate}kt, it takes '
... '{time}hrs to cover {distance}nm').format_map(result)
'At 6kt, it takes 5.2hrs to cover 31.2nm'
```

To break up the long string, we used our knowledge from the *Designing complex if...elif chains* recipe in *Chapter 2*.

To make the warning more visible, the `warnings` module can be used to set a filter that elevates the warning to an error. Use the expression `warnings.simplefilter('error')` to transform warnings into visible exceptions.

## How it works...

Because we've provided default values for all of the parameters, we can provide argument values for any two of the three parameters, and the function can then solve for the third parameter. This saves us from having to write three separate functions.

Returning a dictionary as the final result isn't essential to this. It's a handy way to show inputs and outputs. It allows the function to return a uniform result, no matter which parameter values were provided.

## There's more...

We have an alternative formulation for this, one that involves more flexibility. Python functions have an *all other keywords* parameter, prefixed with `**`.

We can leverage the flexible keywords parameter and insist that all arguments be provided as keywords:

```
def rtd2(**keywords: float) -> dict[str, float | None]:

    rate = keywords.get('rate')
    time = keywords.get('time')
    distance = keywords.get('distance')
```

```
# etc.
```

The keywords type hint states that all of the values for these parameters will be float objects. In some rare case, not all of the keyword parameters are the same type; in this case, some redesign may be helpful to make the types clearer.

This version uses the dictionary `get()` method to find a given key in the dictionary. If the key is not present, a default value of `None` is provided.

The dictionary's `get()` method permits a second parameter, the default, which can be provided instead of `None` if the key is not present.

This kind of open-ended design has the potential advantage of being much more flexible. One potential disadvantage is that the actual parameter names are hard to discern, since they're not part of the function definition, but instead part of the function's body. We can follow the *Writing better docstrings with RST markup* recipe and provide a good docstring. It seems much better, though, to provide the parameter names explicitly as part of the Python code rather than implicitly through documentation.

This has another, and more profound, disadvantage. The problem is revealed in the following bad example:

```
>>> rtd2(distnace=31.2, rate=6)
{'distance': None, 'rate': 6, 'time': None}
```

This isn't the behavior we want. The misspelling of "distance" is not reported as a `TypeError` exception. The misspelled parameter name is not reported anywhere. To uncover these errors, we'd need to add some programming to pop items from the keywords dictionary and report errors on names that remain after the expected names were removed:

```
def rtd3(**keywords: float) -> dict[str, float | None]:  
  
    rate = keywords.pop("rate", None)  
    time = keywords.pop("time", None)  
    distance = keywords.pop("distance", None)  
    if keywords:  
        raise TypeError(  
            f"Invalid keyword parameter: {''.join(keywords.keys())}")
```

This design will spot spelling errors. The extra processing suggests explicit parameter names might be better than the flexibility of an unbounded collection of names.

## See also

- We look at the documentation of functions in the *Writing better docstrings with RST markup* recipe in *Chapter 2*.

## Forcing keyword-only arguments with the \* separator

There are some situations where we have a large number of positional parameters for a function. Pragmatically, a function with more than about three parameters can be confusing. A great deal of conventional mathematics seems to focus on one- and two-parameter functions. There don't seem to be too many common mathematical operators that involve three or more operands.

When it gets difficult to remember the required order for the parameters, this suggests there are too many parameters.

## Getting ready

We'll look at a function to prepare a wind-chill table and write the data to a CSV format output file. We need to provide a range of temperatures, a range of wind speeds, and information on the file we'd like to create. This is a lot of parameters.

One formula for the apparent temperature, the wind chill,  $T_{wc}$ , is this:

$$T_{wc}(T_a, V) = 13.2 + 0.6215T_a - 11.37V^{0.16} + 0.3965T_aV^{0.16}$$

The wind chill temperature,  $T_{wc}$ , is based on the air temperature,  $T_a$ , in degrees C, and the wind speed,  $V$ , in KPH.

For Americans, this requires some conversions:

- Convert the temperature,  $T_a$ , from Farenheit,  $^{\circ}F$ , into Celsius,  $^{\circ}C$ :  $T_a = \frac{5(F-32)}{9}$ .
- Convert windspeed,  $V$ , from MPH,  $V_{mph}$ , into KPH:  $V = 1.609344V_{mph}$ .
- The result,  $T_{wc}$ , needs to be converted from  $^{\circ}C$  back to  $^{\circ}F$ :  $F = 32 + \frac{9T_{wc}}{5}$ .

We won't fold these American conversions into the solution. We'll leave this as an exercise for you.

The function to compute the wind-chill temperature,  $T_{wc}()$  looks like this:

```
def T_wc(T: float, V: float) -> float:
    return 13.12 + 0.6215*T - 11.37*V**0.16 + 0.3965*T*V**0.16
```

This function has an unusual name,  $T_{wc}()$ . We've matched the formal definition of  $T_{wc}$ , rather than enforcing the PEP-8 rule of beginning function names with a lowercase letter. In this case, it seems better to stick with names used in the literature, rather than imposing a name based on language conventions.

One approach to creating a wind-chill table is to create something like this:

```
import csv
from typing import TextIO

def wind_chill(
    start_T: int, stop_T: int, step_T: int,
    start_V: int, stop_V: int, step_V: int,
    target: TextIO
) -> None:
```

```

"""Wind Chill Table."""
writer= csv.writer(target)
heading = ['']+[str(t) for t in range(start_T, stop_T, step_T)]
writer.writerow(heading)
for V in range(start_V, stop_V, step_V):
    row = [float(V)] + [
        T_wc(T, V)
        for T in range(start_T, stop_T, step_T)
    ]
    writer.writerow(row)

```

Before we get to the design problem, let's look at the essential processing. We expect the function using this will have opened an output file using the with context. This follows the *Managing a context using the with statement* recipe in *Chapter 2*. Within this context, we've created a write for the CSV output file. We look at this in more depth in *Chapter 11*.

The value for the heading variable includes a list literal and a comprehension that builds a list. We look at lists in *Chapter 4*. We look at comprehensions and generator expressions in *Chapter 9*.

Similarly, each row of the table is built by an expression that combines a single float value with a list comprehension. The list consists of values computed by the wind-chill function, `T_wc()`. We provide the wind velocity, `V`, based on the row in the table. We also provide a temperature, `T`, based on the column in the table.

The `wind_chill()` function's overall definition presents a problem: the `wind_chill()` function has seven distinct positional parameters. When we try to use this function, we wind up with code like the following:

```

>>> from pathlib import Path
>>> p = Path('data/wc1.csv')
>>> with p.open('w', newline='') as target:
...     wind_chill(0, -45, -5, 0, 20, 2, target)

```

What are all those numbers? Is there something we can do to help explain the purposes behind all those numbers?

## How to do it...

When we have a large number of parameters, it helps to require the use of keyword arguments instead of positional arguments. We can use the `*` as a separator between two groups of parameters.

For our example, the resulting function definition has the following stub definition:

```
def wind_chill_k(
    *,
    start_T: int, stop_T: int, step_T: int,
    start_V: int, stop_V: int, step_V: int,
    target: Path
) -> None:
```

Let's see how it works in practice with different kinds of parameters:

1. When we try to use the confusing positional parameters, we'll see this:

```
>>> wind_chill_k(0, -45, -5, 0, 20, 2, target)
Traceback (most recent call last):
...
TypeError: wind_chill_k() takes 0 positional arguments but 7 were
given
```

2. We must use the function with explicit parameter names, as follows:

```
>>> p = Path('data/wc2.csv')
>>> with p.open('w', newline='') as output_file:
...     wind_chill_k(start_T=0, stop_T=-45, step_T=-5,
...     start_V=0, stop_V=20, step_V=2,
...     target=output_file)
```

This use of mandatory keyword parameters forces us to write a longer, but clearer, statement each time we use this complicated-seeming function.



## How it works...

The `*` character, when used as a parameter definition, separates two collections of parameters:

- Before `*`, we list the argument values that can be *either* positional or named by keyword. In this example, we don't have any of these parameters.
- After `*`, we list the argument values that must be given with a keyword. For our example, this is all of the parameters.

The `print()` function exemplifies this. It has three keyword-only parameters for the output file, the field separator string, and the line end string.

## There's more...

We can, of course, combine this technique with default values for the various parameters. We might, for example, make a change to this, thus introducing a single default value:

```
import sys
from typing import TextIO

def wind_chill_k2(
    *,
    start_T: int, stop_T: int, step_T: int,
    start_V: int, stop_V: int, step_V: int,
    target: TextIO = sys.stdout
) -> None:
    ...
```

We can now use this function in two ways:

- Here's a way to print the table on the console, using the default target:

```
>>> wind_chill_k2(
...     start_T=0, stop_T=-45, step_T=-5,
...     start_V=0, stop_V=20, step_V=2)
```

- Here's a way to write to a file using an explicit target:

```
>>> import pathlib
>>> path = pathlib.Path("data/wc3.csv")
>>> with path.open('w', newline='') as output_file:
...     wind_chill_k2(target=output_file,
...                   start_T=0, stop_T=-45, step_T=-5,
...                   start_V=0, stop_V=20, step_V=2)
```

We can be more confident in these changes because the parameters must be provided by name. We don't have to check carefully to be sure about the order of the parameters.

As a general pattern, we suggest doing this when there are more than three parameters for a function. It's easy to remember one or two. Most mathematical operators are unary or binary. While a third parameter may still be easy to remember, the fourth (and subsequent) parameter will become very difficult to recall.

## See also

- See the *Picking an order for parameters based on partial functions* recipe for another application of this technique.

## Defining position-only parameters with the / separator

We can use the / character in the parameter list to separate the parameters into two groups. Before /, all argument values work positionally. After the / parameter, argument values may be given positionally, or names may be used.

This should be used for functions where the following conditions are all true:

- A few positional parameters are used (no more than three).
- And they are all required.
- And the order is so obvious that any change might be confusing.

This has always been a feature of the standard library. As an example, the `math.sin()` function can only use positional parameters. The formal definition is as follows:

```
>>> help(math.sin)
Help on built-in function sin in module math:

sin(x, /)
    Return the sine of x (measured in radians).
```

Even though there's an `x` parameter name, we can't use this name. If we try to, we'll see the following exception:

```
>>> import math
>>> math.sin(x=0.5)
Traceback (most recent call last):
...
TypeError: math.sin() takes no keyword arguments
```

The `x` parameter can only be provided positionally. The output from the `help()` function provides a suggestion of how the `/` separator can be used to make this happen.

## Getting ready

Position-only parameters are used by some of the internal built-ins; the design pattern can also be helpful, though, in our functions. To be useful, there must be very few position-only parameters. Since most mathematical operators have one or two operands, this suggests one or two position-only parameters can be useful.

We'll consider two functions for conversion of units from the Fahrenheit system used in the US and the Centigrade system used almost everywhere else in the world:

- Convert from  $^{\circ}F$  into  $^{\circ}C$ :  $C = \frac{5(F-32)}{9}$
- Convert from  $^{\circ}C$  into  $^{\circ}F$ :  $F = 32 + \frac{9C}{5}$

Each of these functions has a single argument, making it a reasonable example for a position-only parameter.

## How to do it...

1. Define the function:

```
def F_1(c: float) -> float:  
    return 32 + 9 * c / 5
```

2. Add the / parameter separator after the position-only parameters:

```
def F_2(c: float, /) -> float:  
    return 32 + 9 * c / 5
```

For these examples, we put a `_1` and `_2` suffixes on the function names to make it clear which definition goes with each step of the recipe. These are two versions of the same function, and they should have the same name. They're separated to show the history of writing the functions; this is not a practical naming convention except when writing a book where some partially complete functions have their own unit tests.

## How it works...

The `/` separator divides the parameter names into two groups. In front of `/` are parameters where the argument values must be provided positionally: named argument values cannot be used. After the `/` are parameters where names are permitted.

Let's look at a slightly more complex version of our temperature conversions:

```
def C(f: float, /, truncate: bool=False) -> float:  
    c = 5 * (f - 32) / 9  
    if truncate:  
        return round(c, 0)  
    return c
```

This function has a position-only parameter named `f`. It also has the `truncate` parameter, which can be provided by name. This leads to three separate ways to use this function, as shown in the following examples:

```
>>> C(72)
22.2222222222222222

>>> C(72, truncate=True)
22.0

>>> C(72, True)
22.0
```

The first example shows the position-only parameter and the output without any rounding. This is an awkwardly complex-looking value.

The second example uses the named parameter style to set the non-positional parameter, `truncate`, to `True`. The third example provides both argument values positionally.

## There's more...

This can be combined with the `*` separator to create very sophisticated function signatures. The parameters can be decomposed into three groups:

- Parameters before the `/` separator must be given by position. These must be first.
- Parameters after the `/` separator can be given by position or name.
- Parameters after the `*` separator must be given by name only. These names are provided last, since they can never be matched by position.

## See also

- See the *Forcing keyword-only arguments with the `*` separator* recipe for details on the `*` separator.

## Picking an order for parameters based on partial functions

The term *partial function* is widely used to describe the *partial application* of a function. Some of the argument values are fixed, while others vary. We might have a function,

$f(a, b, c)$ , where there are fixed values for  $a$  and  $b$ . With fixed values, we have a new version of the function,  $f_p(c)$ .

When we look at complex functions, we'll sometimes see a pattern in the ways we use the function. We might, for example, evaluate a function many times with some argument values that are fixed by context, and other argument values that are changing with the details of the processing. Having some fixed argument values suggests a partial function.

Creating a partial function can simplify our programming by avoiding code to repeat the argument values that are fixed by a specific context.

## Getting ready

We'll look at a version of the haversine formula. This computes distances between two points,  $p_1 = (lon_1, lat_1)$  and  $p_2 = (lon_2, lat_2)$ , on the surface of the Earth:

$$a = \sqrt{\sin^2\left(\frac{lat_2 - lat_1}{2}\right) + \cos(lat_1) \cos(lat_2) \sin^2\left(\frac{lon_2 - lon_1}{2}\right)}$$

$$c = 2 \arcsin a$$

The essential calculation yields the central angle,  $c$ , between two points. The angle is measured in radians. We must convert this angle into distance by multiplying by the Earth's mean radius in some given units. If we multiply the angle  $c$  by a radius of 3,959 miles, we'll convert the angle into miles.

Here's an implementation of this function:

```
from math import radians, sin, cos, sqrt, asin

MI = 3959
NM = 3440
KM = 6372
```

```

def haversine(
    lat_1: float, lon_1: float,
    lat_2: float, lon_2: float, R: float
) -> float:
    """Distance between points.
    R is Earth's radius.
    R=MI computes in miles. Default is nautical miles.

    >>> round(haversine(36.12, -86.67, 33.94, -118.40, R=6372.8), 5)
    2887.25995
    """

    Δ_lat = radians(lat_2) - radians(lat_1)
    Δ_lon = radians(lon_2) - radians(lon_1)
    lat_1 = radians(lat_1)
    lat_2 = radians(lat_2)
    a = sqrt(
        sin(Δ_lat / 2) ** 2 +
        cos(lat_1) * cos(lat_2) * sin(Δ_lon / 2) ** 2
    )
    return R * 2 * asin(a)

```

The doctest example uses an Earth radius with an extra decimal point that's not used elsewhere. This example's output will match other examples found online.

The problem we often have is the value for R rarely changes for a specific context. One context may use kilometers throughout the application, while another uses nautical miles. We'd like to impose a context-specific default value like `R = NM` to get nautical miles in a given context without having to edit the module.

We'll look at several common approaches to providing a consistent value for an argument.

## How to do it...

In some cases, an overall context will establish a single value for a parameter. The value will rarely change. The following are three common approaches to providing a consistent value for an argument:

- Wrap the function in a new function that provides the default value.

- Create a partial function with the default value. This has two further refinements:
  - We can provide defaults as a keyword parameters.
  - We can provide defaults as positional parameters.

We'll look at each of these in separate variations in this recipe.

## Wrapping a function

Here's how we can revise the function, slightly, and create a wrapper:

1. Make some parameters positional and some parameters keywords. We want the contextual features – the ones that rarely change – to be keywords. The parameters that change more frequently should be left as positional:

```
def haversine_k(  
    lat_1: float, lon_1: float,  
    lat_2: float, lon_2: float, *, R: float  
) -> float:  
    ... # etc.
```

We can follow the *Forcing keyword-only arguments with the \* separator* recipe.

2. We can then write a wrapper function that will apply all of the positional arguments, unmodified. It will supply the additional keyword argument as part of the long-running context:

```
def nm_haversine_1(*args):  
    return haversine_k(*args, R=NM)
```

We have the `*args` construct in the function declaration to accept all positional argument values in a single tuple, `args`. We use a similar-looking `*args` when evaluating the `haversine()` function to expand the tuple into all of the positional argument values to this function.

In this case, all the types are `float`. We can use `*args: float` to provide a suitable hint. This doesn't always work out, and this style of handling arguments – while simple-looking



– can hide problems.

## Creating a partial function with keyword parameters

One approach to defining functions that work well as partial function is to use keyword parameters:

1. We can follow the *Forcing keyword-only arguments with the \* separator* recipe to do this. We might change the basic haversine function so that it looks like this:

```
def haversine_k(
    lat_1: float, lon_1: float,
    lat_2: float, lon_2: float, *, R: float
) -> float:
    ... # etc.
```

2. Create a partial function using the keyword parameter:

```
from functools import partial
nm_haversine_3 = partial(haversine, R=NM)
```

The `partial()` function builds a new function from an existing function and a concrete set of argument values. The `nm_haversine_3()` function has a specific value for `R` provided when the `partial` was built.

We can use this like we'd use any other function:

```
>>> round(nm_haversine_3(36.12, -86.67, 33.94, -118.40), 2)
1558.53
```

We get an answer in nautical miles, allowing us to do boating-related calculations. Having a fixed value for `R=NM` leaves the code slightly simpler-looking, and much more trust-worthy. The possibility of one computation having an incorrect value for `R` is eliminated.

## Creating a partial function with positional parameters

If we try to use `partial()` with positional arguments, we're constrained to providing the leftmost parameter values in the partial definition. This leads us to think of the first few

arguments to a function as candidates for being hidden by a partial function or a wrapper:

1. We need to change the basic haversine function to put the R parameter first. This makes it slightly easier to define a partial function. Here's the changed definition:

```
def p_haversine(
    R: float,
    lat_1: float, lon_1: float, lat_2: float, lon_2: float
) -> float:
    # etc.
```

2. Create a partial function using the positional parameter:

```
from functools import partial
nm_haversine_4 = partial(p_haversine, NM)
```

The `partial()` function builds a new function from an existing function and a concrete set of argument values. The `nm_haversine_4()` function has a specific value for the first parameter, R, that's provided when the partial was built.

We can use this like we'd use any other function:

```
>>> round(nm_haversine_4(36.12, -86.67, 33.94, -118.40), 2)
1558.53
```

We get an answer in nautical miles, allowing us to do boating-related calculations easily. The code can use a version of the `haversine()` function without the annoying detail of repeating the `R=NM` argument value.

## How it works...

A partial function is, essentially, identical to a wrapper function. We can build partials freely in the middle of other, more complex, pieces of a program. Note that creating partial functions leads to a few additional considerations when looking at the order for positional parameters:

- If we try to use `*args` in the wrapper, these must be defined last. All of these

parameters become anonymous. This anonymity means tools like **mypy** may have problems confirming the parameters are being used correctly. The documentation will not show the necessary details, either.

- The leftmost positional parameters are easiest to provide a value for when creating a partial function.
- Any keyword-only parameters, defined after the `*` separator, are also a good choice to provide as part of a partial definition.

These considerations can lead us to look at the leftmost argument as being a kind of context: these parameters are expected to change rarely and can be provided more easily by partial function definitions.

## There's more...

There's yet another way to wrap a function – we can also build a lambda object. The following example will also work:

```
nm_haversine_L = lambda *args: haversine_k(*args, R=NM)
```

This relies on the `haversine_k()` function definition, where the `R` parameter is marked as keyword-only. Without this clear separation between positional and keyword argument values, this lambda definition will result in a warning from **mypy**. If we use the original `haversine()` function, the warning tells us that it's possible for `R` to get multiple values.

A lambda object is a function that's been stripped of its name and body. The function definition is reduced to just two essentials:

- The parameter list, `*args`, in this example.
- A single expression, which is the result, `haversine_k(*args, R=NM)`. A lambda cannot have any statements.

The lambda approach makes it difficult to create type hints. This limits its utility. Further, the PEP-8 recommendations suggest assigning a lambda to a variable should never be done.

## See also

- We'll also look at extending this design further in the *Writing testable scripts with the script-library switch* recipe.
- For more functional programming techniques, see *Functional Python Programming*: <https://www.packtpub.com/product/functional-python-programming-3rd-edition-third-edition/9781803232577>. This has numerous examples of using lambdas and partial functions.

## Writing clear documentation strings with RST markup

How can we clearly document what a function does? Can we provide examples? Of course we can, and we really should. In the *Including descriptions and documentation* recipe in *Chapter 2*, and in the *Writing better docstrings with RST markup* recipe, we looked at some essential documentation techniques. Those recipes introduced **ReStructuredText (RST)** for module docstrings.

We'll extend those techniques to write RST for function docstrings. When we use a tool such as Sphinx, the docstrings from our function will become elegant-looking documentation that describes what our function does.

### Getting ready

In the *Forcing keyword-only arguments with the \* separator* recipe, we looked at a function to compute wind-chill, given the temperature and wind-speed.

In the recipe, we'll show several versions of the function with trailing `_0` in the name. Pragmatically, this name change is not a good idea. For the purposes of making the evolution of this function clear in this book, however, it seems helpful to give each new variant a distinct name.

We need to annotate this function with some more complete documentation.

## How to do it...

We'll generally write the following things for a function description:

- Synopsis
- Description
- Parameters
- Returns
- Exceptions
- Test cases
- Anything else that seems meaningful

Here's how we'll create documentation for a function. We can apply a similar method to a method of a class, or even a module.

1. Write the synopsis. A proper subject isn't required. Don't write *This function computes...*; we can start with *Computes...*. There's no reason to overstate the context:

```
def T_wc_1(T, V):  
    """Computes the wind chill temperature."""
```

To help clarify the evolution of this function's docstring in this book, we've appended a suffix of `_1` to the name.

2. Write the description and provide details:

```
def T_wc_2(T, V):  
    """Computes the wind chill temperature.  
    The wind-chill,  $T_{wc}$ ,  
    is based on air temperature, T, and wind speed, V.  
    """
```

In this case, we used a little block of typeset math in our description. The `:math:` interpreted text role uses  $\LaTeX$ math typesetting. Tools like Sphinx can use MathJax or jsMath to do handle math typesetting.

3. Describe the parameters. For positional parameters, it's common to use `:param name:` description. Sphinx will tolerate a number of variations, but this is common. For parameters that must be keywords, it's common to use `:key name:` as the prefix to the description.

```
def T_wc_3(T: float, V: float):
    """Computes the wind chill temperature
    The wind-chill,  $T_{wc}$ ,
    is based on air temperature, T, and wind speed, V.

    :param T: Temperature in °C
    :param V: Wind Speed in kph
    """
```

4. Describe the return value using `:returns:`:

```
def T_wc_4(T: float, V: float) -> float:
    """Computes the wind chill temperature
    The wind-chill,  $T_{wc}$ ,
    is based on air temperature, T, and wind speed, V.

    :param T: Temperature in °C
    :param V: Wind Speed in kph

    :returns: Wind-Chill temperature in °C
    """
```

5. Identify the important exceptions that might be raised. Use the `:raises exception:` markup to define the reasons for the exception. There are several possible variations, but `:raises exception:` seems to be popular:

```
def T_wc_5(T: float, V: float) -> float:
    """Computes the wind chill temperature
    The wind-chill,  $T_{wc}$ ,
    is based on air temperature, T, and wind speed, V.

    :param T: Temperature in °C
    :param V: Wind Speed in kph
```

```

:returns: Wind-Chill temperature in °C

:raises ValueError: for wind speeds under 4.8 kph or T above 10°C
"""

```

6. Include a doctest test case, if possible:

```

def T_wc(T: float, V: float) -> float:
    """Computes the wind chill temperature
    The wind-chill, :math:`T_{wc}`,
    is based on air temperature, T, and wind speed, V.

    :param T: Temperature in °C
    :param V: Wind Speed in kph

    :returns: Wind-Chill temperature in °C

    :raises ValueError: for wind speeds under 4.8 kph or T above 10°C

    >>> round(T_wc(-10, 25), 1)
    -18.8

```

7. Write any additional notes and helpful information. We could add the following to the docstring:

```

See https://en.wikipedia.org/wiki/Wind\_chill
.. math::

    T_{wc}(T_a, V) = 13.2 + 0.6215 T_a - 11.37 V ^ {0.16} + 0.3965
    T_a V ^ {0.16}

```

We've included a reference to a Wikipedia page that summarizes wind-chill calculations and has links to more detailed information.

We've also included a `.. math::` directive with the LaTeX formula that's used in the function. This will often typeset nicely, providing a very readable version of the code.

## How it works...

For more information on docstrings, see the *Including descriptions and documentation* recipe in *Chapter 2*. While **Sphinx** is popular, it isn't the only tool that can create documentation from the docstring comments. The **pydoc** utility that's part of the Python Standard Library can also produce good-looking documentation from the docstring comments.

The **Sphinx** tool relies on the core features of RST processing in the **Docutils** package. See <https://pypi.python.org/pypi/docutils> for more information.

The RST rules are relatively simple. Most of the additional features in this recipe leverage the *interpreted text roles* of RST. Each of our `:param T:`, `:returns:`, and `:raises ValueError:` constructs is a text role. The RST processor can use this information to decide on a style and structure for the content. The style usually includes a distinctive font. The context might be an HTML **definition list** format.

## There's more...

In many cases, we'll also need to include cross-references among functions and classes. For example, we might have a function that prepares a wind-chill table. This function might have documentation that includes a reference to the `T_wc()` function.

Sphinx will generate these cross-references using a special `:func:` text role:

```
def wind_chill_table() -> None:
    """Uses :func:`T_wc` to produce a wind-chill
    table for temperatures from -30°C to 10°C and
    wind speeds from 5kph to 50kph.
    """
    ... # etc.
```

We've used `:func:`T_wc`` to create a reference from one function in the RST documentation to another function. Sphinx will turn these into proper hyperlinks.



## See also

- See the *Including descriptions and documentation* and *Writing better docstrings with RST markup* recipes in *Chapter 2*, for other recipes that show how RST works.

## Designing recursive functions around Python's stack limits

Some functions can be defined clearly and succinctly using a recursive formula. There are two common examples of this.

The factorial function has the following recursive definition:

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ n \times (n - 1)! & \text{if } n > 0. \end{cases}$$

The recursive rule for computing a Fibonacci number,  $F_n$ , has the following definition:

$$F_n = \begin{cases} 1 & \text{if } n = 0 \vee n = 1, \\ F_{n-1} + F_{n-2} & \text{if } n > 1. \end{cases}$$

Each of these involves a case that has a simple defined value and a case that involves computing the function's value, based on other values of the same function.

The problem we have is that Python imposes an upper limit for these kinds of recursive function evaluations. While Python's integers can easily compute the value of 1000!, the stack limit prevents us from computing this casually.

Pragmatically, the filesystem is an example of a recursive data structure. Each directory contains subdirectories. Recursive function definitions can be used on directory trees. The cases with defined values come from processing the non-directory files.

We can often refactor a recursive design to eliminate the recursion and replace it with

iteration. While doing recursion elimination, we'd like to preserve as much of the original mathematical clarity as possible.

## Getting ready

Many recursive function definitions follow the pattern set by the factorial function. This is sometimes called **tail recursion** because the recursive case can be written at the tail of the function body:

```
def fact_r(n: int) -> int:
    if n == 0:
        return 1
    return n * fact_r(n - 1)
```

The last expression in the function refers to the same function, but uses a different argument value.

We can restate this, avoiding the recursion limits in Python.

## How to do it...

A tail recursion can also be described as a **reduction**. We're going to start with a collection of values, and then reduce them to a single value:

1. Expand the rule to show all of the details:  $n! = n \times (n - 1) \times (n - 2) \times (n - 3) \times \dots \times 1$ . This helps ensure we understand the recursive rule.
2. Write a loop or generator to create all the values:  $N = \{n, n - 1, n - 2, n - 3, \dots, 1\}$ . In Python, this can be as simple as `range(1, n+1)`. In some cases, though, we might have to apply some transformation function to the base values:  $N = \{f(i) \mid 1 \leq i < n + 1\}$ . This is a list comprehension; see *Building lists – literals, appending, and comprehensions* in Chapter 4.
3. Incorporate the reduction function. In this case, we're computing a large product, using multiplication. We can summarize this as  $\prod_{1 \leq x < n+1} x$ .

Here's an implementation in Python:

```
def prod_i(int_iter: Iterable[int]) -> int:
    p = 1
    for x in int_iter:
        p *= x
    return p
```

An equivalent function is available in the `math` module. Rather than write it out as shown above, we can use `from math import prod`.

The `prod_i()` function can be used as follows to compute a factorial value:

```
>>> prod_i(range(1, 6))
120
>>> fact(5)
120
```

This works nicely. We've optimized the `prod_i()` function into an iterative function. This revision avoids the potential stack overflow problems the recursive version suffers from.

Note that the `range` object is lazy; it doesn't create a big list object, avoiding the allocation of a great deal of memory. A `range` object returns individual values as they are consumed by the `prod_i()` function.

## How it works...

A tail recursion definition is handy because it's short and easy to remember. Mathematicians like this because it can help clarify what a function means.

Many static, compiled languages create optimized code in a manner similar to the technique we've shown here. This works by injecting a special instruction into the virtual machine's byte code – or the actual machine code – to re-evaluate the function without creating a new stack frame. Python doesn't have this feature. In effect, this optimization transforms a recursion into a kind of `while` statement:

```
def loop_fact(n: int) -> int:
    p = n
    while n != 1:
        n = n-1
        p *= n
    return p
```

The injection of the special byte code instruction will lead to code that runs quickly, without revealing the intermediate revisions. The resulting instructions will not be a perfect match for the source text, however, leading to potential difficulties in locating bugs.

## There's more...

Computing the  $F_n$  Fibonacci number involves an additional problem. If we're not careful, we'll compute a lot of values more than once:

To compute  $F_5 = F_4 + F_3$ , for example, we'll evaluate this:

$$F_5 = (F_3 + F_2) + (F_2 + F_1)$$

Expanding the definition of  $F_3$  and  $F_2$  shows a number of redundant computations.

The Fibonacci problem involves two recursions. If we write it naively, it might look like this:

```
def fibo(n: int) -> int:
    if n <= 1:
        return 1
    else:
        return fibo(n-1) + fibo(n-2)
```

It's difficult to do a simple mechanical transformation to turn something like this example into a tail recursion. We have two ways to reduce the computation complexity of this:

- Use memoization
- Restate the problem

The **memoization** technique is easy to apply in Python. We can use the `@functools.cache` as a decorator. It looks like this:

```
from functools import cache

@cache
def fibo_r(n: int) -> int:
    if n < 2:
        return 1
    else:
        return fibo_r(n - 1) + fibo_r(n - 2)
```

Adding this decorator will optimize a more complex recursion.

Restating the problem means looking at it from a new perspective. In this case, we can think of computing all Fibonacci numbers up to and including the desired  $F_n$ . We only want the last value in this sequence. Computing a number of intermediate values can be reasonably efficient.

Here's a generator function that does this:

```
from collections.abc import Iterator

def fibo_iter() -> Iterator[int]:
    a = 1
    b = 1
    yield a
    while True:
        yield b
        a, b = b, a + b
```

This function is an infinite iteration of Fibonacci numbers. It uses Python's `yield` so that it emits values in a lazy fashion. When a client function uses this iterator, the next number in the sequence is computed as each number is consumed.

Here's a function that consumes the values and also imposes an upper limit on the otherwise infinite iterator:

```
def fibo_i(n: int) -> int:
    for i, f_i in enumerate(fibo_iter()):
        if i == n:
            break
    return f_i
```

This function consumes a sequence of values from the `fibo_iter()` iterator. When the desired number has been reached, the `break` statement ends the `for` statement.

We've optimized the recursive solution and turned it into an iteration that avoids the potential for stack overflow.

## See also

- See the *Avoiding a potential problem with break statements* recipe in *Chapter 2*.

## Writing testable scripts with the script-library switch

It's often very easy to create a Python script file. When we provide a script file to Python, it runs immediately. In some cases, there are no function or class definitions; the script file is the sequence of Python statements.

These script files are very difficult to test. Additionally, they're also difficult to reuse. When we want to build larger and more sophisticated applications from a collection of script files, we're often forced to re-engineer a script into one or more functions.

## Getting ready

Let's say that we have a handy implementation of the haversine distance function called `haversine()`, and it's in a file named `recipe_11.py`.

The file contains the functions and definitions shown in the *Picking an order for parameters based on partial functions* in this chapter. This includes a partial function, `nm_haversine()`, to compute distances in nautical miles. The script also contains the following top-level code:

```
source_path = Path("data/waypoints.csv")
with source_path.open() as source_file:
    reader = csv.DictReader(source_file)
    start = next(reader)
    for point in reader:
        d = nm_haversine(
            float(start['lat']),
            float(start['lon']),
            float(point['lat']),
            float(point['lon'])
        )
        print(start, point, d)
        start = point
```

This Python script opens a file, `data/waypoints.csv`, and does some processing on that file. While this is handy to use, we can't easily test it.

If we try to import the `haversine()` function for a unit test, we'll execute the other parts of the script. How can we refactor this module so we can import the useful functions without it printing a display of distances between waypoints in the `waypoints.csv` file?

## How to do it...

Writing a Python script can be called an *attractive nuisance*; it's attractively simple, but it's difficult to test effectively. Here's how we can transform a script into a testable and reusable library:

1. Identify the statements that do the *work* of the script. This means distinguishing between *definitions* and *actions*. Statements such as `import`, `def`, and `class` are definitional – they create objects but don't take a direct action to compute or produce the output. Almost all other statements take some action. Because some assignment statements might be part of type hint definition, or might create useful constants, the distinction is entirely one of intent.
2. In our example, we have some assignment statements that are more definition than action. These assignments are analogous to `def` statements; they only set variables

that are used later. Here are the generally definitional statements:

```
from math import radians, sin, cos, sqrt, asin
from functools import partial

MI = 3959
NM = 3440
KM = 6373

def haversine(
    lat_1: float, lon_1: float,
    lat_2: float, lon_2: float, *, R: float) -> float:
    ... # etc.

nm_haversine = partial(haversine, R=NM)
```

The rest of the statements in the module are designed to take an action toward producing the printed results.

3. Wrap the actions into a function. Try to pick a descriptive name. If there's no better name, use `main()`. In this example the action computes distances, so we'll call the function `distances()`.

```
def distances_draft():
    source_path = Path("data/waypoints.csv")
    with source_path.open() as source_file:
        reader = csv.DictReader(source_file)
        start = next(reader)
        for point in reader:
            d = nm_haversine(
                float(start['lat']),
                float(start['lon']),
                float(point['lat']),
                float(point['lon'])
            )
            print(start, point, d)
            start = point
```

In the above example, we named the function `distances_draft()` to assure that it's



clearly distinct from a more final version. Practically, using distinct names like this as code evolves toward completion isn't necessary, unless writing a book where it's essential to unit test intermediate steps.

4. Where possible, extract literals and turn them into parameters. This is often a simple movement of the literal to a parameter with a default value.

```
def distances(  
    source_path: Path = Path("data/waypoints.csv")  
) -> None:  
    ... # etc.
```

This makes the script reusable because the path is now a parameter instead of an assumption.

5. Include the following `if` statement as the only high-level action statements in the script file:

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

We've packaged the action of the script as a function. The top-level action script is now wrapped in an `if` statement so that it isn't executed during `import` but is executed when the script is run directly.

## How it works...

An important rule for Python is that an `import` of a module is essentially the same as running the module as a script. The statements in the file are executed, in order, from top to bottom.

When we `import` a file, we're generally interested in executing the `def` and `class` statements. We might be interested in some assignment statements that define useful globals. Sometimes, we're not interested in executing the main program.

When Python runs a script, it sets a number of built-in special variables. One of these is

`__name__`. This variable has two different values, depending on the context in which the file is being executed:

- The top-level script, executed from the command line: In this case, the value of the built-in special name of `__name__` is set to `"__main__"`.
- A file being executed because of an `import` statement: In this case, the value of `__name__` is the name of the module being created from reading the file and executing the Python statements.

The standard name of `"__main__"` may seem a little odd at first. Why not use the filename in all cases? This special name is assigned because a Python script can be read from one of many sources. It can be a file. Python can also be read from the stdin pipeline, or it can be provided on the Python command line using the `-c` option.

## There's more...

We can now build useful work around a reusable library. We might make an application script file that look like this:

```
from pathlib import Path
from ch03.recipe_11 import distances

if __name__ == "__main__":
    for trip in 'trip_1.csv', 'trip_2.csv':
        distances(Path('data') / trip)
```

The goal is to decompose a practical solution into two collections of features:

- The definition of classes and functions
- A very small action-oriented script that uses the definitions to do useful work

We often start with a script that conflates both sets of features. This kind of script can be viewed as a **spike solution**. Our spike solution can evolve toward a more refined solution as soon as we're sure that it works. A *spike* or *piton* is a piece of removable mountain-climbing gear that enables us to climb safely.

After starting with a spike, we can elevate our design and refactor the code into definitions and actions. Tests can then import the module to test the various definitions without taking actions that might overwrite important files.

## See also

- In *Chapter 7*, we look at class definitions. These are another kind of widely used definitional statement, in addition to function definitions.
- The *Reading delimited files with the CSV module* recipe that we look at in *Chapter 11* also addresses CSV file reading.

## Join our community Discord space

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

<https://packt.link/dHrHU>



# 4

## Built-In Data Structures Part 1: Lists and Sets

Python has a rich collection of built-in data structures. These data structures are sometimes called “containers” or “collections” because they contain a collection of individual items. These structures cover a wide variety of common programming situations.

We’ll look at an overview of the various collections that are built in and what problems they solve. After the overview, we will look at the list and set collections in detail.

The built-in tuple and string types were part of *Chapter 1, Numbers, Strings, and Tuples*. These structures are sequences, making them similar in many ways to the `list` collection. However, strings and tuples seem to have more in common with immutable numbers.

The next chapter, *Chapter 5*, will look at dictionaries, as well as some more advanced topics also related to lists and sets. In particular, it will look at how Python handles references to mutable collection objects. This has consequences in the way functions need to be defined that accept lists or sets as parameters.

In this chapter, we'll look at the following recipes, all related to Python's built-in data structures:

- *Choosing a data structure*
- *Building lists – literals, appending, and comprehensions*
- *Slicing and dicing a list*
- *Shrinking lists – deleting, removing, and popping*
- *Writing list-related type hints*
- *Reversing a copy of a list*
- *Building sets – literals, adding, comprehensions, and operators*
- *Shrinking sets – remove(), pop(), and difference*
- *Writing set-related type hints*

## Choosing a data structure

Python offers a number of built-in data structures to help us work with collections of data. It can be confusing to match the data structure features with the problem we're trying to solve.

How do we choose which structure to use?

### Getting ready

Before we put data into a collection, we'll need to consider how we'll gather the data, and what we'll do with the collection once we have it. One big question is how to identify a particular item within the collection. Python offers a variety of choices.

### How to do it...

1. Is the programming focused on the existence of a value? An example of this is validating an input value. When the user enters something that's in a collection,

their input is valid; otherwise, the entry is invalid. Simple membership tests suggest using a set:

```
def confirm() -> bool:
    yes = {"yes", "y"}
    no = {"no", "n"}
    while (answer := input("Confirm: ")).lower() not in (yes | no):
        print("Please respond with yes or no")
    return answer in yes
```

A set holds items in no particular order. If order matters, then a list is more appropriate.

2. Are we going to identify items by their position in the collection? An example includes the lines in an input file—the line number is its position in the collection.

When we identify an item using an index or position, we must use a list:

```
>>> month_name_list = ["Jan", "Feb", "Mar", "Apr",
... "May", "Jun", "Jul", "Aug",
... "Sep", "Oct", "Nov", "Dec"]
>>> month_name_list[8]
'Sep'
>>> month_name_list.index("Feb")
1
```

We have created a list, `month_name_list`, with 12 string items in a specific order. We can pick an item by providing the index position. We can also use the `index()` method to return the index position of an item in the list. List index values in Python always start with zero. While a list has a simple membership test, the test can be slow for a very large list, and a set might be a better idea if many such tests will be needed.

If the number of items in the collection is fixed—for example, RGB colors have three values—this suggests a tuple instead of a list. If the number of items will grow and change, then the list collection is a better choice than the tuple collection.

3. Are we going to identify the items in a collection by a key value that's distinct

from the item's index? An example might include a mapping between strings of characters—words, for example—and integers that represent the frequencies of those words. Another example might be a mapping between a color name and the RGB tuple for that color. We'll look at mappings and dictionaries in *Chapter 5, Built-In Data Structures Part 2: Dictionaries*. The important distinction is mappings do not locate items by a numerical index position the way lists do.

4. Consider the mutability of items in a set collection (and the keys in a dictionary). Each item in a set must be an immutable object. Numbers, strings, and tuples are all immutable and can be collected into sets. Since list, dictionary, and set objects are mutable, they can't be used as items in a set. It's impossible to build a set of list objects, for example.

Rather than create a set of list items, we can transform each item into an immutable tuple object. Similarly, dictionary keys must be immutable. We can use a number, a string, or a tuple as a dictionary key. We can't use a list, or a set, or any other mutable object as a dictionary key.

## How it works...

Each of Python's built-in collections offers a specific set of unique features. The collections also offer a large number of overlapping features. The challenge for programmers new to Python is to map the unique features of each collection to the problem they are trying to solve.

The `collections.abc` module provides a kind of road map through the built-in container classes. This module defines the **Abstract Base Classes (ABCs)** underlying the concrete classes we use. We'll use the names from this set of definitions to guide us through the features.

From the ABCs, we can see that there are places for a total of three general kinds of collections with six implementation choices:

- **Set:** Its unique feature is that items are either members or not. This means duplicates

are ignored:

- **Mutable set:** The built-in set collection
- **Immutable set:** The built-in frozenset collection
- **Sequence:** Its unique feature is that items are provided with an index position:
  - **Mutable sequence:** The built-in list collection
  - **Immutable sequence:** The built-in tuple collection. This is the subject of some recipes in *Chapter 1*.
- **Mapping:** Its unique feature is that each item has a key that refers to a value:
  - **Mutable mapping:** The built-in dict collection. This is the subject of *Chapter 5*.
  - **Immutable mapping:** Interestingly, there's no built-in frozen mapping.

Python's libraries offer additional implementations of these core collection types. The collections module include:

- **namedtuple:** A tuple that offers names for each item in a tuple. It's slightly clearer to use `rgb_color.red` than `rgb_color[0]`.
- **deque:** A double-ended queue. It's a mutable sequence with optimizations for pushing and popping from each end. We can do similar things with a list, but deque is more efficient when changes at both ends are needed.
- **defaultdict:** A dict that can provide a default value for a missing key.
- **Counter:** A dict that is designed to count occurrences of a key. This is sometimes called a multiset or a bag.
- **ChainMap:** A dict that combines several dictionaries into a single mapping.

Additionally, there's an older `OrderedDict` class. This class retains the keys in the order in which they were created. Starting with Python 3.7, the dictionary keys for an ordinary



dictionary are retained in the order they were created, making the `OrderedDict` class redundant.

## There's more...

There's still more in the *Python Standard Library*. We can also use the `heapq` module, which defines a kind of list that acts as a high-performance priority queue. The `bisect` module includes methods for searching a sorted list very quickly. This lets us create a `list` object, which can have performance that is a little closer to the very fast lookups of a dictionary.

We can find descriptions of data structures on summary web pages, like this one: <https://thealgorist.com>. We'll take a quick look at four additional families of data structures:

- **Arrays:** The Python `array` module supports densely packed arrays of values. The `numpy` module also offers very sophisticated array processing.
- **Trees:** Generally, tree structures can be used to create sets, sequential lists, or key-value mappings. We can look at a tree as an implementation technique for building sets or dictionaries. We often build tree structures using objects and class definitions.
- **Hashes:** Python uses hashes to implement dictionaries and sets. This leads to good speed but potentially large memory consumption.
- **Graphs:** Python doesn't have a built-in graph data structure. However, we can easily represent a graph structure with a dictionary where each node has a list of adjacent nodes. External libraries like **NetworkX**, **Pyoxigraph**, and **RDFLib** support sophisticated graph databases.

We can—with a little cleverness—implement almost any kind of data structure in Python. While it's often the case that the built-in structures have the essential we may be able to locate a built-in structure that can be pressed into service. We'll look at mappings and dictionaries in *Chapter 5, Built-In Data Structures Part 2: Dictionaries*.

## See also

- For high-performance array processing, see <https://numpy.org>.

- For advanced graph analysis, see <https://networkx.github.io>.
- For graph manipulation and storage, see <https://pyoxigraph.readthedocs.io/en/stable/>.
- For graph manipulation, see <https://rdflib.readthedocs.io/en/stable/>.

## Building lists – literals, appending, and comprehensions

If we've decided to create a collection based on each item's position in the container—a list—we have several ways of building this structure. We'll look at a number of ways we can assemble a list object from the individual items.

In some cases, we'll need a list because it allows duplicate values, unlike a set. This is common in statistical work. A different structure, called a *multiset*, can also be useful for a statistically oriented collection that permits duplicates. This collection is available in the standard library as `collections.Counter`.

### Getting ready

Let's say we need to do some statistical analyses of some file sizes. Here's a short script that will provide us with the sizes of some files:

```
>>> from pathlib import Path
>>> home = Path.cwd() / "data"
>>> for path in sorted(home.glob('*.csv')):
...     print(path.stat().st_size, path.name)
260 binned.csv
250 ch14_r03.csv
2060 ch14_r04.csv
45 craps.csv
225 fuel.csv
156 fuel2.csv
28 output.csv
19760 output_0.csv
19860 output_1.csv
19645 output_2.csv
```

```
19971 output_3.csv
19588 output_4.csv
...
```

We've used a `pathlib.Path` object to represent a directory in our filesystem. The `glob()` method expands all names that match a given pattern.

We'd like to accumulate a list object that has the various file sizes. From that list, we can compute the total size and average size.

## How to do it...

We have many ways to create list objects:

- **Literal:** We can create a literal display of a list using a sequence of values surrounded by `[]` characters. For example, `[1, 2, 3]`. Python needs to match an opening `[` and a closing `]` to see a complete logical line, so the literal can span physical lines. For more information, refer to the *Writing long lines of code* recipe in *Chapter 2*.
- **Conversion function:** We can convert some other data collection into a list using the `list()` function.
- **Append method:** We have list methods that allow us to build a list one item at a time. These methods include `append()`, `extend()`, and `insert()`. We'll look at the `append()` method in the *Building a list with the `append()` method* section of this recipe.
- **Comprehension:** A comprehension is a specialized generator expression that computes a list from a source object. We'll look at this in detail in the *Writing a list comprehension* section of this recipe.

The first two ways to create a list are single Python expressions. The last two are more complex, and we'll show recipes for each of them.

### Building a list with the `append()` method

1. Create an empty list using literal syntax, `[]`, or the `list()` function:

```
>>> file_sizes = []
```

2. Iterate through some source of data. Append the items to the list using the `append()` method:

```
>>> home = Path.cwd() / "data"
>>> for path in sorted(home.glob('*.*csv')):
...     file_sizes.append(path.stat().st_size)

>>> print(file_sizes)
[260, 250, 2060, 45, 225, 156, 28, 19760, 19860, 19645, 19971, 19588,
19999, 20000, 20035, 19739, 19941, 215, 412, 28, 166, 0, 1810, 0, 0,
16437, 20295]
>>> print(sum(file_sizes))
240925
```

When we print the list, Python displays it in literal notation. This is handy if we ever need to copy and paste the list into another script.

It's very important to note that the `append()` method does not return a value. The `append()` method mutates the list object, and does not return anything.

## Writing a list comprehension

The goal of a list comprehension is to create an object that occupies the syntax role of a literal:

1. Write the wrapping `[]` brackets that surround the list object to be built.
2. Write the source of the data. This will include the target variable. Note that there's no `:` at the end of the `for` clause because we're not writing a complete statement:

```
[... for path in home.glob('*.*csv')]
```

3. Prefix the `for` clause with an expression to evaluate to create each value that goes into the sequence from the value of target variable. Again, since this is only a single expression, we cannot use complex statements here:

```
[path.stat().st_size
 for path in home.glob('*.csv')]
```

Here's an example of list object construction:

```
>>> [path.stat().st_size
...   for path in sorted(home.glob('*.csv'))]
[260, 250, 2060, 45, 225, 156, 28, 19760, 19860, 19645, 19971, 19588, 19999,
20000, 20035, 19739, 19941, 215, 412, 28, 166, 0, 1810, 0, 0, 16437, 20295]
```

Now that we've created a list object, we can assign it to a variable and do other calculations and summaries on the data.

The list comprehension is built around a central generator expression, called a **comprehension** in the language manual. The comprehension has two parts: the data expression clause and a for clause. The data expression clause is evaluated repeatedly, driven by the variables assigned in the for clause.

We can replace the enclosing [ and ] with the `list()` function. Using the explicit `list()` function had an advantage when we consider the possibility of changing the data structure. We can easily replace `list()` with `set()` or `Counter()` to make use of the core generator, but creating a distinct collection type.

## How it works...

A Python list object has a dynamic size. The size is adjusted when items are appended or inserted, or the list is extended with items from another sequence. Similarly, the size shrinks when items are popped or deleted.

In rare cases, we might want to create a list with a given initial size, and then set the values of the items separately. We can do this with a list comprehension, like this:

```
>>> sieve = [True for i in range(100)]
```

This will create a list with an initial size of 100 items, each of which is `True`. We might

need this kind of initialization to implement the Sieve of Eratosthenes algorithm:

```
>>> sieve[0] = sieve[1] = False
>>> for p in range(100):
...     if sieve[p]:
...         for n in range(p*2, 100, p):
...             sieve[n] = False
>>> prime = [p for p in range(100) if sieve[p]]
```

The sieve collection has a sequence of True and False values. The index position of each True is a prime number. Multiples of each prime,  $p$ , starting with  $p^2$ , are set to False. The prime collection is a sequence of values,  $p$  for which the expression `sieve[p]` is True.

## There's more...

A common goal for creating a list object is to be able to summarize it. We can use a variety of Python functions for this. Here are some examples:

```
>>> sizes = list(path.stat().st_size
...             for path in home.glob('*.*csv'))
>>> sum(sizes)
240925
>>> max(sizes)
20295
>>> min(sizes)
0
>>> from statistics import mean
>>> round(mean(sizes), 3)
8923.148
```

We've used the built-in `sum()`, `min()`, and `max()` methods to produce some descriptive statistics of these document sizes. Which of these index files is the smallest? We want to know the position of the minimum in the list of values. We can use the `index()` method for this:

```
>>> sizes.index(min(sizes))
1
```

We found the minimum, and then used the `index()` method to locate the position of that minimal value.

## Other ways to extend a list

We can extend a list object, as well as insert one into the middle or beginning of a list. We have two ways to extend a list: we can use the `+` operator or we can use the `extend()` method. Here's an example of creating two lists and putting them together with the `+` operator:

```
>>> home = Path.cwd() / "src"
>>> ch3 = list(path.stat().st_size
...           for path in home.glob('ch03/*.py'))
>>> ch4 = list(path.stat().st_size
...           for path in home.glob('ch04/*.py'))

>>> len(ch3)
16
>>> len(ch4)
6
>>> final = ch3 + ch4
>>> len(final)
22
>>> sum(final)
34853
```

We have created a list of sizes of documents with names like `Chapter_03/*.py`. We then created a second list of sizes of documents with a slightly different name pattern, `Chapter_04/*.py`. We then combined the two lists into a final list.

We can insert a value prior to any particular position in a list. The `insert()` method accepts the position of an item; the new value will be before the given position:

```
>>> p = [3, 5, 11, 13]
>>> p.insert(0, 2)
>>> p
[2, 3, 5, 11, 13]
```

```
>>> p.insert(3, 7)
>>> p
[2, 3, 5, 7, 11, 13]
```

We've inserted two new values into a list object. As with the `append()` and `extend()` methods, the `insert()` method does not return a value. It mutates the list object.

## See also

- Refer to the *Slicing and dicing a list* recipe for ways to copy lists and pick sublists from a list.
- Refer to the *Shrinking lists – deleting, removing, and popping* recipe for other ways to remove items from a list.
- In the *Reversing a copy of a list* recipe, we'll look at reversing a list.
- This article provides some insights into how Python collections work internally: <https://wiki.python.org/moin/TimeComplexity>.

When looking at the tables, it's important to note the expression  $O(1)$  means that the cost is essentially constant. The expression  $O(n)$  means the cost grows as the size of the collection grows.

## Slicing and dicing a list

There are many times when we'll want to pick items from a list. One of the most common kinds of processing is to treat the first item of a list as a special case. This leads to a kind of *head-tail* processing where we treat the head of a list differently from the items in the tail of a list.

We can use these techniques to make a copy of a list too.

## Getting ready

We have a spreadsheet that was used to record fuel consumption on a large sailboat. It has rows that look like this:



date	engine on engine off Other notes	fuel height fuel height
10/25/2013	08:24:00 AM	29
	01:15:00 PM	27
	calm seas – anchor solomon’s island	
10/26/2013	09:12:00 AM	27
	06:25:00 PM	22
	choppy – anchor in jackson’s creek	

Table 4.1: Example of sailboat fuel use

In this dataset, fuel is measured by height. This is because a **sight-gauge** is used, calibrated in inches of depth. For all practical purposes, the tank is rectangular, so the depth shown can be converted into volume since we know 31 inches of depth is about 75 gallons.

This example of spreadsheet data is not properly normalized. Ideally, all rows follow the **First Normal Form** for data: a row should have identical content, and each cell should have only atomic values. In this data, there are three subtypes of row:

1. The first row of a three-row group has engine on date, time, and a measurement.
2. The second row of a group has engine off time and a measurement.
3. The third row has some notes that aren’t too useful.

This kind of denormalized data includes the following two problems:

- The .csv file has four rows of headings. (The fourth row is a blank line that’s not shown here in this nicely formatted book.) This is something the csv module can’t deal with directly.
- Each day’s travel is spread across three rows. These rows must be combined to make it easier to compute an elapsed time and the number of inches of fuel used.

We can read the data with a function defined like this:

```
import csv
from pathlib import Path

def get_fuel_use(path: Path) -> list[list[str]]:
    with path.open() as source_file:
        reader = csv.reader(source_file)
        log_rows = list(reader)
    return log_rows
```

We've used the `csv` module to read the log details. The object returned by the `csv.reader()` function is iterable. In order to collect the items into a single list, we applied the `list()` function to the iterable; this creates a list object from the reader.

Each row of the original CSV file is a list. Here's what the first and last rows look like:

```
>>> log_rows[0]
['date', 'engine on', 'fuel height']

>>> log_rows[-1]
['', "choppy -- anchor in jackson's creek", '']
```

For this recipe, we'll use an extension of a list index expression to slice items from the list of rows. The slice, like an index expression, follows the list object in `[]` characters. Python offers several variations of the slice expression so that we can extract useful subsets of the list of rows.

## How to do it...

1. The first thing we need to do is remove the four lines of headings from the list of rows. We'll use two partial slice expressions to divide the list by the fourth row:

```
>>> head, tail = log_rows[:4], log_rows[4:]
>>> head[0]
['date', 'engine on', 'fuel height']
>>> head[-1]
['', '', '']
>>> tail[0]
```

```
['10/25/13', '08:24:00 AM', '29']
>>> tail[-1]
['', "choppy -- anchor in jackson's creek", '']
```

We've sliced the list into two sections using `log_rows[:4]` and `log_rows[4:]`. The first slice expression selects the first four lines; this is assigned to the `head` variable. The second slice expression selects rows from 4 to the end of the list. This is assigned to the `tail` variable. These are the rows of the sheet we care about.

2. We'll use slices with steps to pick the interesting rows. The `[start:stop:step]` version of a slice will pick rows in groups based on the step value. In our case, we'll take two slices. One slice starts on row zero—the “engine on” lines—and the other slice starts on row one—the “engine off” lines.

Here's a slice of every third row, starting with row zero:

```
>>> pprint(tail[0::3], width=64)
[['10/25/13', '08:24:00 AM', '29'],
 ['10/26/13', '09:12:00 AM', '27']]
```

We've used the `pprint()` function from the `pprint` module to make the output much easier to read.

There's additional data in every third row, starting with row one:

```
>>> pprint(tail[1::3], width=48)
[['', '01:15:00 PM', '27'],
 ['', '06:25:00 PM', '22']]
```

3. These two slices can then be zipped together to create a list of pairs:

```
>>> paired_rows = list(zip(tail[0::3], tail[1::3]))
>>> pprint(paired_rows)
[[('10/25/13', '08:24:00 AM', '29'), ['', '01:15:00 PM', '27']],
 [('10/26/13', '09:12:00 AM', '27'), ['', '06:25:00 PM', '22']]
```

This gives us a sequence that consists of pairs of three tuples. This is very close to something we can work with.

#### 4. Flatten the results:

```
>>> paired_rows = list(zip(tail[0::3], tail[1::3]))
>>> combined = [a+b for a, b in paired_rows]
>>> pprint(combined)
[['10/25/13', '08:24:00 AM', '29', '', '01:15:00 PM', '27'],
 ['10/26/13', '09:12:00 AM', '27', '', '06:25:00 PM', '22']]
```

We've used a list comprehension from the *Building lists – literals, appending, and comprehensions* recipe to combine the two elements in each pair of rows to create a single row. This has more properly normalized data describing each leg of the voyage.

From the resulting list, we can now compute the difference in times to get the running time for the boat. We can compute the difference in heights to estimate the fuel consumed during each leg of the journey. This flat list with five useful items—date, time, height, time, and height—has all the needed data in a single row. It also has a column that will generally contain an empty string.

## How it works...

The slice operator has several different forms:

- `[:]`: The start and stop are implied. The expression `S[:]` will create a copy of sequence `S`.
- `[:stop]`: This makes a new list from the beginning to just before the stop index.
- `[start:]`: This makes a new list from the given start to the end of the sequence.
- `[start:stop]`: This picks a sublist, starting from the start index and stopping just before the stop index. Python works with half-open intervals. The start is included, while the stop index is not included.
- `[:]::step]`: The start and stop are implied and include the entire sequence. The

step—generally not equal to one—means we’ll skip through the list from the start using the step. For a given step,  $s$ , and a list of size  $|L|$ , the index values are  $i \in \{s \times n \mid n \in \mathbb{N} \text{ and } 0 \leq s \times n < |L|\}$ .

- `[start : : step]`: The start is given, but the stop is implied. The idea is that the start is an offset, and the step applies to that offset. For a given start,  $a$ , step,  $s$ , and a list of size  $|L|$ , the index values are  $i \in \{s \times n + a \mid n \in \mathbb{N} \text{ and } 0 \leq s \times n + a < |L|\}$ .
- `[ : stop : step]`: This is used to prevent processing the last few items in a list. Since the step is given, processing begins with element zero.
- `[start : stop : step]`: This will pick elements from a subset of the sequence. Items prior to start and from stop to the end will not be used.

The slicing technique works for lists, tuples, strings, and any other kind of sequence. Slicing does not cause the collection to be mutated; rather, slicing will make a copy of some part of the sequence. The items within the source collection are now shared between collections.

## There’s more...

In the *Reversing a copy of a list* recipe, we’ll look at an even more sophisticated use of slice expressions.

The copy of a sequence is called a **shallow copy** because there will be two collections that each contain references to the same underlying objects. We’ll look at this in detail in the *Making shallow and deep copies of objects* recipe.

For this specific example, we have another way of restructuring multiple rows of data into single rows of data: we can use a generator function. We’ll look at functional programming techniques online in *Chapter 9*.

## See also

- Refer to the *Building lists – literals, appending, and comprehensions* recipe for ways to create lists.
- Refer to the *Shrinking lists – deleting, removing, and popping* recipe for other ways to

remove items from a list.

- In the *Reversing a copy of a list* recipe, we'll look at reversing a list.
- The pandas package offers some additional ways to work with CSV files.

## Shrinking lists – deleting, removing, and popping

There will be many times when we'll want to remove items from a list collection. We might delete items from a list, and then process the items that are left over.

Removing unneeded items has a similar effect to using `filter()` to create a copy that has only the needed items. The distinction is that a filtered copy will use more memory than deleting items from a list. We'll show both techniques for removing unwanted items from a mutable list.

### Getting ready

We have a spreadsheet that is used to record fuel consumption on a large sailboat. See *Table 4.1* for the data.

For more background on this data, refer to the *Slicing and dicing a list* recipe earlier in this chapter. The `get_fuel_use()` function will collect the raw data. It's important to note that the structure of this data—each fact spread among three separate rows—is perfectly awful and requires considerable care to reconstruct something more useful.

Each row of the original CSV file is a list. Each of those lists contains three items. It's essential to remove some rows with titles and uninformative data.

### How to do it...

We'll look at several ways to remove items from a list:

- The `del` statement.
- The `remove()` method.
- The `pop()` method.

- We can also replace items in a list using slice assignment.

## The del statement

We can remove items from a list using the `del` statement. We can provide an object and a slice to remove a group of rows from the list object. Here's how the `del` statement looks:

```
>>> del log_rows[:4]
>>> log_rows[0]
['10/25/13', '08:24:00 AM', '29']

>>> log_rows[-1]
['', "choppy -- anchor in jackson's creek", '']
```

The `del` statement removed the first four rows, leaving behind the rows that we really need to process. We can then combine these rows and summarize them using the *Slicing and dicing a list* recipe.

## The remove() method

We can remove items from a list using the `remove()` method. Given a specific value, this removes matching items from a list.

We might have a list that looks like this:

```
>>> row = ['10/25/13', '08:24:00 AM', '29', '', '01:15:00 PM', '27']
```

We can remove the useless `''` item from the list:

```
>>> row.remove('')
>>> row
['10/25/13', '08:24:00 AM', '29', '01:15:00 PM', '27']
```

Note that the `remove()` method does not return a value. It mutates the list in place.

As noted in the *Building lists – literals, appending, and comprehensions* recipe, the following code is incorrect:

```
a = ['some', 'data']
a = a.remove('data')
```

This is emphatically wrong. This will set `a` to `None`.

## The `pop()` method

We can remove items from a list using the `pop()` method. This removes items from a list based on their index.

We might have a list that looks like this:

```
>>> row = ['10/25/13', '08:24:00 AM', '29', '', '01:15:00 PM', '27']
```

This has a useless `''` string in it. We can find the index of the item to `pop` and then remove it. The code for this has been broken down into separate steps in the following example:

```
>>> target_position = row.index('')
>>> target_position
3

>>> row.pop(target_position)
''

>>> row
['10/25/13', '08:24:00 AM', '29', '01:15:00 PM', '27']
```

Note that the `pop()` method does two things:

- It mutates the list object to remove an item.
- It also returns the item that was removed.

This combination of mutation and returning a value is rare, making this method distinctive.

## Slice assignment

We can replace items in a list by using a slice expression on the left-hand side of the assignment statement. This lets us replace items in a list. When the replacement is a



different size, it lets us expand or contract a list. This leads to a technique for removing items from a list using slice assignment.

We'll start with a row that has an empty value in position 3. This looks like this:

```
>>> row = ['10/25/13', '08:24:00 AM', '29', '', '01:15:00 PM', '27']

>>> target_position = row.index('')
>>> target_position
3
```

We can assign an empty list to the slice that starts at index position 3 and ends just before index position 4. This will replace a one-item slice with a zero-item slice, removing the item from the list:

```
>>> row[3:4] = []
>>> row
['10/25/13', '08:24:00 AM', '29', '01:15:00 PM', '27']
```

The `del` statement and methods like `remove()` and `pop()` seem to clearly state the intent to eliminate an item from the collection. The slice assignment can be less clear because it doesn't have an obvious method name. It does work well, however, for removing a number of items that can be described by a slice expression.

## How it works...

Because a list is a mutable object, we can remove items from the list. This technique doesn't work for tuples or strings, because they are immutable.

We can only remove items with an index that's present in the list. If we attempt to remove an item with an index outside the allowed range, we'll get an `IndexError` exception.

The following example tries to delete an item with an index of three from a list where the index values are zero, one, and two:

```
>>> row = ['', '06:25:00 PM', '22']

>>> del row[3]
Traceback (most recent call last):
...
IndexError: list assignment index out of range
```

## There's more...

There are a few places in Python where deleting from a list object may become complicated. If we use a list object in a for statement, we can't delete items from the list. Doing so will lead to unexpected conflicts between the iteration control and the underlying object's internal state.

Let's say we want to remove all even items from a list. Here's an example that does not work properly:

```
>>> data_items = [1, 1, 2, 3, 5, 8, 10,
... 13, 21, 34, 36, 55]

>>> for f in data_items:
...     if f % 2 == 0:
...         data_items.remove(f)
>>> data_items
[1, 1, 3, 5, 10, 13, 21, 36, 55]
```

The source list had several even values. The result is clearly not correct; the values of 10 and 36 remained in the list. Why are some even-valued items left in the list?

Let's look at what happens when processing `data_items[5]`; it has a value of 8. When the `remove(8)` method is evaluated, the value will be removed, and all the subsequent values will slide forward one position in the list. The 10 value will be moved into position 5, the position formerly occupied by the 8 value. The iterator control value will advance to the next position, which will have 13 in it. The 10 value will never be processed.

We have several ways to avoid the *skip-when-delete* problem:

- Make a copy of the list:

```
>>> for f in data_items[:]:
...     if f % 2 == 0:
...         data_items.remove(f)
```

- Use a while statement and maintain the index value explicitly:

```
>>> position = 0
>>> while position != len(data_items):
...     f = data_items[position]
...     if f % 2 == 0:
...         data_items.remove(f)
...     else:
...         position += 1
```

We've designed a while statement to only increment the position variable if the value of `data_items[position]` is odd. If the value is even, then the value is removed, which also means the other items are moved forward one position in the list; it's essential the value of the position variable is left unchanged.

- We can also traverse the list in reverse order. The expression `range(len(row) - 1, -1, -1)` will produce index descending from -1. This works because negative index values work forward from the end of the list. The value `row[-1]` is the last item.

## See also

- Refer to the *Building lists – literals, appending, and comprehensions* recipe for ways to create lists.
- Refer to the *Slicing and dicing a list* recipe for ways to copy lists and pick sublists from a list.
- In the *Reversing a copy of a list* recipe, we'll look at reversing a list.

## Writing list-related type hints

The typing module provides a few essential type definitions for describing the contents of a list object. The primary type definition is `list`, which we can parameterize with the types of items in the list. It often looks like `list[int]`.

### Getting ready

We'll look at a list that has two kinds of tuples. Some tuples are simple RGB colors. Other tuples are RGB colors that are the result of some computations. These are built from float values instead of integers. We might have a heterogenous list structure that looks like this:

```
scheme = [  
    (' Brick_Red', (198, 45, 66)),  
    (' color1', (198.00, 100.50, 45.00)),  
    (' color2', (198.00, 45.00, 142.50)),  
]
```

Each item in the list is a two-tuple with a color name, and a tuple of RGB values. The RGB values are represented as a three-tuple of either integer or float values. This is potentially difficult to describe with type hints.

We have two related functions that work with this data. The first creates a color code from RGB values.

The essential rule is to treat each component, red, green, or blue, as an 8-bit number, a value between 0 and 255. These three are combined by shifting the red value by 16 bits and shifting the green value by 8 bits. The Python `<<` operator does the necessary bit shifting. The `|` operator performs an “or” operation, combining the shifted bits to create a new integer value.

The hints for this function aren't very complicated:

```
def hexify(r: float, g: float, b: float) -> str:  
    return f'#{int(r) << 16 | int(g) << 8 | int(b):06X}'
```

The `:06X` format specification produces a 6-position hexadecimal value.

An alternative is to treat each color as a separate pair of hex digits with an expression like `f"#{int(r):02X}{int(g):02X}{int(b):02X}"`. This uses three copies of the `:02X` format specification to produce 2-position hexadecimal values for each color component.

When we use this function to create a color string from an RGB number, it looks like this:

```
>>> hexify(198, 45, 66)
'#C62D42'
```

The other function, however, is potentially confusing. This function transforms a complex list of colors into another list with the hexadecimal color codes:

```
def source_to_hex_0(src):
    return [
        (n, hexify(*color)) for n, color in src
    ]
```

We need to add type hints to be sure this function properly transforms a list of colors from numeric form into string code form.

We've included a `_0` suffix on the function name to distinguish it from the examples that follow. This is not a best practice in general, but we find it helps clarify the code presented in a book like this.

## How to do it...

We'll start by adding type hints to describe the individual items of the input list, exemplified by the scheme variable, shown previously:

1. Define the resulting type first. It often helps to focus on the outcomes and work backward toward the source data required to produce the expected results. In this case, the result is a list of two-tuples with the color name and the hexadecimal code for the color. We could describe this as `list[tuple[str, str]]`, but that kind of summary hides some important details. We prefer to expose the details as follows:

```
ColorCode = tuple[str, str]
ColorCodeList = list[ColorCode]
```

This list can be seen as being homogeneous; each item will match the `ColorCode` type definition.

2. Define the source type. In this case, we have two slightly different kinds of color definitions. While they tend to overlap, they have different origins, and the processing history is sometimes helpful as part of a type hint:

```
from typing import Union
RGB_I = tuple[int, int, int]
RGB_F = tuple[float, float, float]
ColorRGB = tuple[str, Union[RGB_I, RGB_F]]
ColorRGBList = list[ColorRGB]
```

We've defined the two integer-based RGB three-tuple as `RGB_I`, and the float-based RGB three-tuple as `RGB_F`. These two alternative types are combined into the `ColorRGB` tuple definition. This is a two-tuple; the second element may be an instance of either the `RGB_I` type or the `RGB_F` type. The presence of a `Union` type means that this list is heterogenous.

We could also use `RGB_I | RGB_F` instead of `Union[RGB_I, RGB_F]`.

3. Update the function to include the type hints. The input will be a list like the schema object, shown previously. The result will be a list that matches the `ColorCodeList` type description:

```
def source_to_hex(src: ColorRGBList) -> ColorCodeList:
    return [
        (n, hexify(*color)) for n, color in src
    ]
```

## How it works...

The `list[T]` type hint requires a single value, `T`, to describe all of the object types that can be part of this list. For homogeneous lists, the type is stated directly. For heterogeneous lists, a `Union` must be used to define the various kinds of types that may be present.

The approach we've taken breaks type hinting down into two layers:

- A “foundation” layer that describes the individual items in a collection. We've defined three types of primitive items: the `RGB_I` and `RGB_F` types, as well as the resulting `ColorCode` type.
- A number of “composition” layers that combine foundational types into descriptions of composite objects. In this case, `ColorRGB`, `ColorRGBList`, and `ColorCodeList` are all composite type definitions.

Once the types have been named, then the names are used with definition functions, classes, and methods.

It's important to define types in stages to avoid long, complex type hints that don't provide any useful insight into the objects being processed. It's good to avoid type descriptions like this:

```
list[tuple[str, Union[tuple[int, int, int], tuple[float, float, float]]]]
```

While this is technically correct, it's difficult to understand because of its complexity. It helps to decompose complex types into useful component descriptions.

## There's more...

The type hints assume a single type for each item in the list. The syntax `list[T]` states that all items are of type `T`.

In the case of a heterogeneous list, with a number of distinct types, we need to define a union of types. We can import the `Union` type from the `typing` module. Or we can use `|` to provide the alternative types for a list.

Using a construct like `list[RGB_I | RGB_F]` describes a list that contains items with a mixture of types.

## See also

- In *Chapter 1*, the *Using NamedTuples to simplify item access in tuples* recipe provides some alternative ways to clarify types hints for tuples.
- The *Writing set-related type hints* recipe covers this from the view of set types.

## Reversing a copy of a list

Some algorithms produce results in a reversed order. It's common to collect the output in a list and then reverse the list. As an example, we'll look at the way numbers converted into a specific base are often generated from least-significant to most-significant digit. We generally want to display the values with the most-significant digit first. This leads to a need to reverse the sequence of digits in a list.

## Getting ready

Let's say we're doing a conversion among number bases. We'll look at how a number is represented in a base, and how we can compute that representation from a number.

Any value,  $v$ , can be defined as a polynomial function of the various digits,  $d_n$ , in a given base,  $b$ . A four-digit number would have  $\langle d_3, d_2, d_1, d_0 \rangle$  as the sequence of digits.

Note that the order we're using here is reversed from the usual order of items in a Python list.

The value,  $v$ , of this sequence of digits is given by the following polynomial:

$$v = d_n \times b^n + d_{n-1} \times b^{n-1} + d_{n-2} \times b^{n-2} + \dots + d_1 \times b + d_0$$

For example, the hexadecimal number `0xBEEF` has the following digits:  $\langle B = 11, E = 14, E = 14, F = 15 \rangle$ , with base  $b = 16$ :



$$48879 = 11 \times 16^3 + 14 \times 16^2 + 14 \times 16 + 15$$

There are many cases where the base isn't a consistent power of some number. The ISO date format, for example, has a mixed base that involves 7 days per week, 24 hours per day, 60 minutes per hour, and 60 seconds per minute.

Instead of  $b^4$ ,  $b^3$ ,  $b^2$ ,  $b^1 = b$ , and  $b^0 = 1$ , we have  $7 \times 24 \times 60 \times 60$ ,  $24 \times 60 \times 60$ ,  $60 \times 60$ , and 60 as the various values used to compute the polynomial.

Given a week number, a day of the week, an hour, a minute, and a second, we can compute a timestamp of seconds,  $t_s$ , within the given year:

$$t_s = (((w \times 7 + d) \times 24 + h) \times 60 + m) \times 60 + s$$

For example:

```
>>> week = 13
>>> day = 2
>>> hour = 7
>>> minute = 53
>>> second = 19

>>> t_s = (((week*7+day)*24+hour)*60+minute)*60+second
>>> t_s
8063599
```

This shows how we convert from the given moment into a timestamp. How do we invert this calculation? How do we get the various fields from the overall timestamp?

We'll need to use `divmod` style division. For some background, refer to the *Choosing between true division and floor division* recipe.

The algorithm for converting a timestamp in seconds,  $t_s$ , into individual week, day, and time fields looks like this:

$$t_m; s = \lfloor \frac{t_s}{60} \rfloor; t_s \bmod 60$$
$$t_h; m = \lfloor \frac{t_m}{60} \rfloor; t_m \bmod 60$$
$$t_d; h = \lfloor \frac{t_h}{24} \rfloor; t_h \bmod 24$$
$$w; d = \lfloor \frac{t_d}{7} \rfloor; t_d \bmod 7$$

This has a handy pattern that leads to an implementation. It has the consequence of producing the values in reverse order:

```
>>> t_s = 8063599
>>> fields = []
>>> for base in 60, 60, 24, 7:
...     t_s, f = divmod(t_s, base)
...     fields.append(f)
>>> fields.append(t_s)
>>> fields
[19, 53, 7, 2, 13]
```

We've applied the `divmod()` function four times to extract seconds, minutes, hours, days, and weeks from a timestamp, given in seconds. These are in the wrong order. How can we reverse them?

## How to do it...

We have three approaches: we can use the `reverse()` method, we can use a `[::-1]` slice expression, or we can use the `reversed()` built-in function. Here's the `reverse()` method:

```
>>> fields_copy1 = fields.copy()
>>> fields_copy1.reverse()
>>> fields_copy1
[13, 2, 7, 53, 19]
```

We made a copy of the original list so that we could keep an unmutated copy to compare with the mutated copy. This makes it easier to follow the examples. We applied the

`reverse()` method to reverse a copy of the list.

This will mutate the list. As with other mutating methods, it does not return a useful value. It's incorrect to use a statement like `a = b.reverse()`; the value of `a` will always be `None`.

Here's a slice expression with a negative step:

```
>>> fields_copy2 = fields[::-1]
>>> fields_copy2
[13, 2, 7, 53, 19]
```

In this example, we made a slice `[::-1]` that uses an implied start and stop, and a step of `-1`. This picks all the items in the list in reverse order to create a new list.

The original list is emphatically *not* mutated by this slice operation. This creates a copy. Check the value of the `fields` variable to see that it's unchanged.

Here's how we can use the `reversed()` function to create a reversed copy of a list of values:

```
>>> fields_copy3 = list(reversed(fields))
>>> fields_copy3
[13, 2, 7, 53, 19]
```

It's important to use the `list()` function in this example. The `reversed()` function is a generator, and we need to consume the items from the generator to create a new list.

## How it works...

As we noted in the *Slicing and dicing a list* recipe, the slice notation is quite sophisticated. Using a slice with a negative step size will create a copy (or a subset) with items processed in right to left order, instead of the default left to right order.

It's important to distinguish between these three methods:

- The `reverse()` method modifies the list object itself. As with methods like `append()` and `remove()`, there is no return value from this method. Because it changes the list, it doesn't return a value.

- The `[::-1]` slice expression creates a new list. This is a shallow copy of the original list, with the order reversed.
- The `reversed()` function is a generator that yields the values in reverse order. When the values are consumed by the `list()` function, it creates a copy of the list.

## See also

- Refer to the *Making shallow and deep copies of objects* recipe for more information on what a shallow copy is and why we might want to make a deep copy.
- Refer to the *Building lists – literals, appending, and comprehensions* recipe for ways to create lists.
- Refer to the *Slicing and dicing a list* recipe for ways to copy lists and pick sublists from a list.
- Refer to the *Shrinking lists – deleting, removing, and popping* recipe for other ways to remove items from a list.

## Building sets – literals, adding, comprehensions, and operators

If we've decided to create a collection based on only an item being present—a set—we have several ways of building this structure. Because of the narrow focus of sets, there's no ordering to the items—no relative positions—and items cannot be duplicated. We'll look at a number of ways we can assemble a set collection from a source of individual items.

The set operators parallel the operators defined by the mathematics of set theory. These can be helpful for doing bulk comparisons between sets. We'll look at these in addition to the methods of the set class.

Sets have an important constraint: they only contain immutable objects. Informally, immutable objects have no internal state that can be changed. Numbers are immutable, as are strings, and tuples of immutable objects. Formally, immutable objects have an internal hash value, and the `hash()` function will show this value.

Here's how this looks in practice:

```
>>> a = "string"
>>> hash(a)
... # doctest: +SKIP
4964286962312962439

>>> b = ["list", "of", "strings"]
>>> hash(b)
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'
```

The value of the `a` variable is an immutable string, which has a hash value. The `b` variable, on the other hand, is a mutable list, and doesn't have a hash value. We can create sets of immutable objects like strings, but the `TypeError` exception will be raised if we try to put mutable objects into a set.

## Getting ready

Let's say we need to do some analysis of the dependencies among modules in a complex application. Here's one part of the available data:

```
>>> import_details = [
... ('Chapter_12.ch12_r01', ['typing', 'pathlib']),
... ('Chapter_12.ch12_r02', ['typing', 'pathlib']),
... ('Chapter_12.ch12_r03', ['typing', 'pathlib']),
... ('Chapter_12.ch12_r04', ['typing', 'pathlib']),
... ('Chapter_12.ch12_r05', ['typing', 'pathlib']),
... ('Chapter_12.ch12_r06', ['typing', 'textwrap', 'pathlib']),
... ('Chapter_12.ch12_r07', ['typing', 'Chapter_12.ch12_r06',
... 'Chapter_12.ch12_r05', 'concurrent']),
... ('Chapter_12.ch12_r08', ['typing', 'argparse', 'pathlib']),
... ('Chapter_12.ch12_r09', ['typing', 'pathlib']),
... ('Chapter_12.ch12_r10', ['typing', 'pathlib']),
... ('Chapter_12.ch12_r11', ['typing', 'pathlib']),
... ('Chapter_12.ch12_r12', ['typing', 'argparse'])
```

Each item in this list names a module and a list of modules that it imports. There are a

number of questions we can ask about this collection of relationships among modules. We'd like to compute the short list of dependencies, thereby removing duplicates from this list.

## How to do it...

We have many ways to create set objects:

- **Literal:** We can create literal display of a set using a sequence of values surrounded by characters. It looks like this: `{value, ... }`. Python needs to match the `{` at the start of the literal and `}` at the end of the literal to see a complete logical line, so the literal can span physical lines. For more information, refer to the *Writing long lines of code* recipe in *Chapter 2*.

Note that we can't create an empty set with `{}`; this is an empty dictionary. We must use `set()` to create an empty set.

- **Conversion function:** We can convert some other data collection into a set using the `set()` function. We can convert a list of immutable items, or the keys of a dict, or a tuple of immutable items.
- **Add method:** The set method `add()` will add an item to a set. Additionally, sets can be created by a `union()` method or the `|` operator.
- **Comprehension:** A comprehension is a specialized generator expression that describes the items in a set using an expression to define membership. We'll look at this in detail in the *Writing a set comprehension* section of this recipe.

The first two ways to create sets are single Python expressions. The last two are more complex, and we'll show recipes for each of them.

### Building a set with the add method

Our source collection of data is a list with sublists. We want to summarize the items inside each of the sublists:

1. Create an empty set into which items can be added. Unlike lists, there's no abbreviated

syntax for an empty set, so we must use the `set()` function:

```
>>> all_imports = set()
```

2. Write a `for` statement to iterate through each two-tuple in the `import_details` collection. This needs a nested `for` statement to iterate through each name in the list of imports in each pair. Use the `add()` method of the `all_imports` set to create a complete set with duplicates removed:

```
>>> for item, import_list in import_details:
...     for name in import_list:
...         all_imports.add(name)
>>> all_imports == {'Chapter_12.ch12_r06', 'textwrap',
...                 'Chapter_12.ch12_r05', 'pathlib', 'concurrent',
...                 'argparse', 'typing'}
True
```

This result summarizes many lines of details, showing the set of distinct items imported. Note that the order here is arbitrary and can vary each time the example is executed.

The arbitrary ordering means a **doctest** example to confirm the correctness of this code can't simply show the expected result. See the *Handling common doctest issues* recipe in *Chapter 15 for more advice on using doctest*.

## Writing a set comprehension

The goal of a set comprehension is to create an object that occupies a syntax role, similar to a set literal:

1. Write the wrapping braces that surround the set object to be built:

```
>>> {}
{}

```

2. Write the source of the data. This will include the target variable. We have two nested lists, so we'll need to use two `for` clauses. Note that there's no `:` at the end of the `for` clause because we're not writing a complete statement:

```
>>> {...
...     for item, import_list in import_details
...         for name in import_list
...     }
{Ellipsis}
```

For now, we've written the result of the expression as the special `Ellipsis` object. Once we finish this expression, we will replace it with something more useful.

3. Prefix the `for` clause with the expression to evaluate that creates each value of the target collection. In this case, we only want the name from the import list within each pair of items in the overall import details list-of-lists:

```
>>> names = {name
...     for item, import_list in import_details
...         for name in import_list}
>>> names == {'Chapter_12.ch12_r06', 'Chapter_12.ch12_r05',
...     'typing', 'concurrent', 'argparse', 'textwrap', 'pathlib'}
True
```

A set comprehension cannot have duplicates, so this will always have distinct values.

As with the list comprehension, a set comprehension is built around a central generator expression. The generator expression at the heart of the comprehension has a data expression clause and a `for` clause.

We can replace the enclosing `{` and `}` syntax with the `set()` function. Using the explicit `set()` function had an advantage when we consider the possibility of changing the data structure. We can easily replace `set()` with `frozenset()`, `list()`, or `Counter()`.

## How it works...

A set is a collection of immutable objects. Each immutable Python object has a hash value, and these numeric hash codes are used to optimize locating items in a set. We can imagine the implementation relies on an array of buckets, and the numeric hash value directs us to a bucket to see if the item is present in that bucket or not.



Hash values are not necessarily unique. The array of hash buckets is finite, meaning hash collisions are possible. A collision occurs when two distinct objects both have the same hash value. This leads to some overhead to handle any collisions.

We can create two integers that will have a hash collision:

```
>>> import sys
>>> v1 = 7
>>> v2 = 7+sys.hash_info.modulus
>>> v1
7
>>> v2
2305843009213693958

>>> hash(v1)
7
>>> hash(v2)
7
```

In spite of these two objects having the same hash value, hash collision processing will keep these two objects separate from each other in a set.

## There's more...

We have several ways to add items to a set:

- The example used the `add()` method. This works with a single item.
- We can use the `union()` method. This method is like an operator—it creates a new result set. It does not mutate either of the operand sets.
- We can use the `update()` method to update one set with items from another set. This mutates a set and does not return a value.

For most of these techniques, we'll need to create a singleton set from the item we're going to add. Here are some examples:

```
>>> collection = {1}
>>> collection
{1}

>>> item = 3
>>> collection.union({item})
{1, 3}

>>> collection
{1}
```

In the preceding example, we've created a singleton set, `{item}`, from the value of the `item` variable. We then used the `union()` method to compute a new set, which is the union of the `collection` set and the `{item}` set.

Note that `union()` creates a new object and leaves the original `collection` untouched. Here is yet another alternative that uses the union operator, `|`:

```
>>> collection = collection | {item}
>>> collection
{1, 3}
```

We can also use the `update()` method to mutate the set:

```
>>> collection.update({4})
>>> collection
{1, 3, 4}
```

Methods like `update()` and `add()` mutate the set object. Because they mutate the set, they do not return a value. This is similar to the way methods of the `list` collection work. Generally, a method that mutates the collection does not return a value. The only exception to this pattern is the `pop()` method, which both mutates the set object and returns the popped value.

Python has a number of set operators. These are ordinary operator symbols that we can use in complex set expressions:

- | for set union, often typeset as  $A \cup B$
- & for set intersection, often typeset as  $A \cap B$
- ^ for set symmetric difference, often typeset as  $A \triangle B$
- - for set subtraction, often typeset as  $A \setminus B$

## See also

- In the *Shrinking sets – remove(), pop(), and difference* recipe, we'll look at how we can update a set by removing or replacing items.

## Shrinking sets – remove(), pop(), and difference

Python gives us several ways to remove items from a set collection. We can use the `remove()` method to remove a specific item. We can use the `pop()` method to remove (and return) an arbitrary item.

Additionally, we can compute a new set using the set intersection, difference, and symmetric difference operators: `&`, `-`, and `^`. These will produce a new set that is a subset of a given input set.

## Getting ready

Sometimes, we'll have log files that contain lines with complex and varied formats. Here's a small snippet from a long, complex log:

```
>>> log = """
... [2016-03-05T09:29:31-05:00] INFO: Processing ruby_block[print IP] action
run (@recipe_files:~/home/slott/ch4/deploy.rb line 9)
... [2016-03-05T09:29:31-05:00] INFO: Installed IP: 111.222.111.222
... [2016-03-05T09:29:31-05:00] INFO: ruby_block[print IP] called
...

```

```
... (Skipping some details)
```

```
... ""
```

We need to find all of the text similar to IP: 111.222.111.222 in this log. These are IPv4 addresses with 4 numeric fields.

Here's how we can create a set of matches:

```
>>> import re
>>> pattern = re.compile(r"IP: \d+\.\d+\.\d+\.\d+")
>>> matches = set(pattern.findall(log))
>>> matches
{'IP: 111.222.111.222'}
```

The problem we have with this log is extraneous matches. The log file also has text that looks similar but are dummy or placeholder values we need to ignore. In the full log, we'll also find lines containing text like IP: 1.2.3.4, which is a placeholder, not a meaningful address. It turns out that there is a small set of irrelevant values.

This is a place where set intersection and set subtraction can be very helpful.

## How to do it...

1. Create a set of items we'd like to ignore as a set literal:

```
>>> to_be_ignored = {'IP: 0.0.0.0', 'IP: 1.2.3.4'}
```

2. Collect all entries from the log. We'll use the re module for this, as shown earlier. We'll see results like the following:

```
>>> matches = {'IP: 111.222.111.222', 'IP: 1.2.3.4'}
```

3. Remove items from the set of matches using set subtraction. Here are two examples:

```
>>> matches - to_be_ignored
{'IP: 111.222.111.222'}
>>> matches.difference(to_be_ignored)
```

```
{'IP: 111.222.111.222'}
```

Both of these are operators that return new sets as their results. Neither of these will mutate the underlying set objects.

It turns out the `difference()` method can work with any iterable collection, including lists and tuples. While permitted, mixing sets and lists can be confusing, and it can be challenging to write type hints for them.

We'll often use these in statements, like this example:

```
>>> valid_matches = matches - to_be_ignored
>>> valid_matches
{'IP: 111.222.111.222'}
```

This will assign the resulting set to a new variable, `valid_matches`, so that we can do the required processing on this new set.

We can also use the `remove()` and `pop()` methods to remove specific items. The `remove()` method raises an exception when an item cannot be removed. We can use this behavior to both confirm that an item is in the set and remove it.

## How it works...

A set object tracks membership of items. An item is either in the set or not. We specify the item we want to remove. Removing an item doesn't depend on an index position or a key value.

The set operators permit sophisticated set computations. We can remove any of the items in one set from a target set computing set difference or set subtraction.

## There's more...

We have several other ways to remove items from a set:

- In this example, we used the `difference()` method and the `-` operator. The `difference()`

method behaves like an operator and creates a new set.

- We can also use the `difference_update()` method. This will mutate a set in place. It does not return a value.
- We can remove an individual item with the `remove()` method.
- We can also remove an arbitrary item with the `pop()` method. This doesn't apply to this example very well because we can't control which item is popped from a set.

Here's how the `difference_update()` method looks:

```
>>> valid_matches = matches.copy()
>>> valid_matches.difference_update(to_be_ignored)
>>> valid_matches
{'IP: 111.222.111.222'}
```

We applied the `difference_update()` method to remove the undesirable items from the `valid_matches` set. Since the `valid_matches` set was mutated, no value is returned. Also, since the set is a copy, this operation doesn't modify the original `matches` set.

We could do something like the following example to use the `remove()` method. Note that `remove()` will raise an exception if an item is not present in the set:

```
>>> valid_matches = matches.copy()
>>> for item in to_be_ignored:
...     if item in valid_matches:
...         valid_matches.remove(item)

>>> valid_matches
{'IP: 111.222.111.222'}
```

We tested to see if the item was in the `valid_matches` set before attempting to remove it. Using an `if` statement is one way to avoid raising a `KeyError` exception. An alternative is to use a `try:` statement to silence the exception that's raised when an item is not present.

We can also use the `pop()` method to remove an arbitrary item. This method is unusual

in that it both mutates the set and returns the item that was removed. However, we can't control which item is popped, making it inappropriate for this example.

## Writing set-related type hints

The typing module provides a few essential type definitions for describing the contents of a set object. The primary type definition is `set`, which we can parameterize with the types of items in the set. We'll use `set[int]` to describe a set composed of integers. This parallels the *Writing list-related type hints* recipe.

## Getting ready

A dice game like *Zonk* (also called *10,000* or *Greed*) requires a random collection of dice to be grouped into “hands.” While rules vary, there are several patterns for hands, including:

- Three of a kind.
- A “small straight” of five ascending dice (1-2-3-4-5 or 2-3-4-5-6 are the two combinations).
- A “large straight” of six ascending dice.
- An “ace” hand. This has at least one 1 die that's not part of a three of a kind or straight.
- Six of a kind. While rare, it's not impossible.

We'll use the following class and function definitions to create the hands of dice:

```
import random

class Die(str, Enum):
    d_1 = "\u2680"
    d_2 = "\u2681"
    d_3 = "\u2682"
    d_4 = "\u2683"
    d_5 = "\u2684"
    d_6 = "\u2685"
```

```
def zonk(n: int = 6) -> tuple[Die, ...]:
    faces = list(Die)
    return tuple(random.choice(faces) for _ in range(n))
```

The Die class definition enumerates the six faces of a standard die by providing the Unicode character with the appropriate value.

When we evaluate the `zonk()` function, it looks like this.

```
>>> zonk()
(<Die.d_6: '⚰'>, <Die.d_1: '⚱'>, <Die.d_1: '⚱'>,
 <Die.d_6: '⚰'>, <Die.d_3: '⚳'>, <Die.d_2: '⚲'>)
```

This shows us a hand with two sixes, two ones, a two, and a three. When examining the hand for patterns, we will often create complex sets of objects.

## How to do it...

A function to do analysis of the patterns of dice works by creating a set [Die] object from the six dice instances. This set reveals a great deal of information:

- When there is one die in the set of unique values, then all six dice have the same value.
- When there are five distinct dice in the set of unique values, then this could be a small straight. This requires an additional check to see if the set of unique values is 1-5 or 2-6, which are the two valid small straights.
- When there are six distinct items in the set of unique values, then this must be a large straight.
- For two unique dice values there will be at least one three of a kind. There may be a four or five of a kind, but these are scored as a three of a kind and the remaining dice are non-scoring.
- For three or four unique dice in the set, there may be a three of a kind. More detailed



analysis of the set is required to see the exact pattern.

We can distinguish many of the patterns by looking at the cardinality of the set of distinct dice. The remaining distinctions can be made by looking at the pattern of counts. For this, a `collections.Counter` object will be useful.

Here's how to write this set-based analysis:

1. Define the type for each item in the set. In this example, the `Die` class is the item class. We'll work with `set[Die]` and `Counter[Die]` types.
2. Create the set object with the unique values from the hand of `Die` instances. Here's how the evaluation function can begin:

```
import collections

def eval_zonk_6(hand: tuple[Die, ...]) -> str:
    assert len(hand) == 6, "Only works for 6-dice zonk."
    unique: set[Die] = set(hand)
```

3. There are two small straight definitions: 1-5 and 2-6:

```
faces = list(Die)
small_straights = [
    set(faces[:-1]), set(faces[1:])
]
```

We can build these two sets in the body of the analysis function to show how they're used. Pragmatically, the value of `small_straights` should be computed only once.

We can't build a set of these two set instances because set objects are mutable. We could build a set of two `frozenset` objects instead of building a list.

4. Examine the simple cases. The number of distinct elements in the set identifies several kinds of hands directly:

```

if len(unique) == 6:
    return "large straight"
elif len(unique) == 5 and unique in small_straights:
    return "small straight"
elif len(unique) == 2:
    return "three of a kind"
elif len(unique) == 1:
    return "six of a kind"

```

5. When there are three or four distinct values, the patterns can be summarized using the counts. This pattern of frequency counts can be summarized as a set[int]:

```

elif len(unique) in {3, 4}:
    # 4 unique: wwwxyz (good) or wwxyz (bad)
    # 3 unique: xxxxyz, xxxyyz (good) or xxyyzz (bad)
    frequencies: set[int] = set(
        collections.Counter(hand).values())

```

6. For the cases of three or four distinct Die values, these can form a variety of patterns. If at least one of the Die has a frequency of three or four, that's a scoring combination. If nothing else matches and there's a die showing a one, that's a minimal score:

```

if 3 in frequencies or 4 in frequencies:
    return "three of a kind"
elif Die.d_1 in unique:
    return "ace"

```

7. Are there any conditions left over? Does this cover all the possible cardinalities of dice and frequencies of dice? The remaining cases includes collections of pairs and singletons without any "one" showing. After the above if statement, we can provide a single return statement to collect all other cases into a single, non-scoring *Zonk*:

```

return "Zonk!"

```

This shows two ways of using sets to evaluate the pattern of a collection of data items. The first set, set[Die], looked at the overall pattern of unique Die values. The second set,

set[int], looked at the pattern of frequencies of Die values.

## How it works...

The essential property of a set is membership. When we compute a set from a collection of Die instances, we used the set[Die] type hint to describe this structure.

Similarly, when we look at the distribution of frequencies, there are only a few distinct patterns. Transforming the counts into a set of values into a collection with the set[int] type hint described this additional structure.

## There's more...

A set of items that don't have a single, uniform type is potentially confusing. We can use set[T1 | T2] to describe a set where items can be any of the types T1 or T2.

Computing the score of the hand of dice depends on which dice were part of the winning pattern. This means the evaluation function needs to return a more complex result when the outcome is a three of a kind. To determine the points, there are three cases we need to consider:

- Which value of the Dice class occurred three or six times. This determines the base score. Often, 1s are given 1,000 points, and 2 through 6 are given 200 through 600 points.
- It's possible to roll two triples; this pattern must be distinguished, too. This often scores 2,000 points, irrespective of the numbers shown.
- For the straights and the "ace" hand, simple fixed scores are assigned. A small straight might be 1,000 points. A large straight 2,000. A lone 1 might score only 50 points.

We have two separate conditions to identify the patterns of unique values indicating a three-of-a-kind pattern. The function needs some refactoring to properly identify the values of the dice occurring three or more times and the values of the dice which were ignored. We've left this additional design as an exercise for the reader.

## See also

- Refer to the *Writing list-related type hints* recipe in this chapter for more about type hints for lists.

## Join our community Discord space

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

<https://packt.link/dHrHU>





# 5

## Built-In Data Structures Part 2: Dictionaries

Starting with *Chapter 4, Built-In Data Structures Part 1: Lists and Sets*, we began looking at Python’s rich collection of built-in data structures. These data structures are sometimes called “containers” or “collections” because they contain a collection of individual items.

In this chapter, we’ll cover the dictionary structure. A dictionary is a mapping from keys to values, sometimes called an associative array. It seems sensible to separate mappings from the two sequences – lists and sets.

This chapter will also look at some more advanced topics related to how Python handles references to mutable collection objects. This has consequences in the way functions need to be defined.

In this chapter, we’ll look at the following recipes, all related to Python’s built-in data structures:

- *Creating dictionaries – inserting and updating*

- *Shrinking dictionaries – the pop() method and the del statement*
- *Writing dictionary-related type hints*
- *Understanding variables, references, and assignment*
- *Making shallow and deep copies of objects*
- *Avoiding mutable default values for function parameters*

We'll start with how to create a dictionary.

## Creating dictionaries – inserting and updating

A dictionary is one kind of Python mapping. The built-in type `dict` provides a number of foundational features. There are some common variations on these features defined in the `collections` module.

As we noted in the *Choosing a data structure* recipe at the beginning of *Chapter 4*, we'll use a dictionary when we have a key that we need to map the key to a given value. For example, we might want to map a single word to a long, complex definition of the word, or perhaps map some value to a count of the number of times that value has occurred in a dataset.

### Getting ready

We'll look at an algorithm for locating the various stages in transaction processing. This relies on assigning a unique ID to each request and including that ID with each log record written during the transaction. Because a multi-threaded server may be handling a number of requests concurrently, the stages for each request's transaction will be interleaved unpredictably. Reorganizing the log by request ID helps isolate each transaction.

Here's a simulated sequence of log entries for three concurrent requests:

```
[2019/11/12:08:09:10,123] INFO #PJQXB^{eRwnEGG?2%32U path="/openapi.yaml" method=GET
[2019/11/12:08:09:10,234] INFO 9DiC!B^{nXxnEGG?2%32U path="/items?limit=x" method=GET
[2019/11/12:08:09:10,235] INFO 9DiC!B^{nXxnEGG?2%32U error="invalid query"
```

```
[2019/11/12:08:09:10,345] INFO #PJQXB^{eRwnEGG?2%32U status="200" bytes="11234"
[2019/11/12:08:09:10,456] INFO 9DiC!B^{nXxnEGG?2%32U status="404" bytes="987"
[2019/11/12:08:09:10,567] INFO >~UL>~PB_R>&nEGG?2%32U path="/category/42" method=GET
```

The lines are long, and may be wrapped haphazardly to fit within the book's margins. Each line has a timestamp. The severity level is INFO for each record shown in the example. The next string of 20 characters is a transaction ID. This is followed by log information for unique to a step in the transaction.

The following regular expression defines the log records:

```
import re
log_parser = re.compile(r"\[(.*?)\] (\w+) (\S+) (.*)" )
```

This pattern captures the four fields of each log entry. For more information on regular expression, see the *String parsing with regular expressions* recipe in *Chapter 1*.

Parsing these lines will produce a sequence of four-tuples. The resulting object looks like this:

```
[('2019/11/12:08:09:10,123',
  'INFO',
  '#PJQXB^{eRwnEGG?2%32U',
  'path="/openapi.yaml" method=GET'),
 ('2019/11/12:08:09:10,234',
  'INFO',
  '9DiC!B^{nXxnEGG?2%32U',
  'path="/items?limit=x" method=GET'),
```

... details omitted ...

```
('2019/11/12:08:09:10,567',
  'INFO',
  '>~UL>~PB_R>&nEGG?2%32U',
  'path="/category/42" method=GET')]
```

We need to know how often each unique path is requested. This means ignoring some log



records and collecting data from the other records. A mapping from the path string to a count is an elegant way to gather this data. We'll look at how to implement this in detail. Later, we'll look at some alternative implementations in the `collections` module.

## How to do it...

We have a number of ways to build dictionary objects:

- **Literal:** We can create a display of a dictionary by using a sequence of key/value pairs surrounded by `{}` characters. We use a `:` between a key and the associated value. Literals look like this: `{"num": 355, "den": 113}`.
- **Conversion function:** A sequence of two-tuples can be turned into a dictionary like this: `dict([('num', 355), ('den', 113)])`. Each two-tuple becomes a key-value pair. The keys must be immutable objects like strings, numbers, or tuples of immutable objects. We can also build dictionaries like this: `dict(num=355, den=113)`. Each of the parameter names becomes a key. This limits the dictionary keys to strings that are also valid Python variable names.
- **Insertion:** We can use the dictionary `[key] = value` syntax to set or replace a value in a dictionary. We'll look at this later in this recipe.
- **Comprehensions:** Similar to lists and sets, we can write a dictionary comprehension to build a dictionary from some source of data.

## Building a dictionary by setting items

We build a dictionary by creating an empty dictionary and then setting items to it:

1. Create an empty dictionary to map paths to counts. We can also use `dict()` to create an empty dictionary. Since we're going to create a histogram that counts the number of times a path is used, we'll call it `histogram`:

```
>>> histogram = {}
```

We can also use the function `dict()` instead of the literal value `{}` to create an empty

dictionary.

2. For each of the log lines, filter out the ones that do not have a value that starts with path in the item with an index of 3:

```
>>> for line in log_lines:
...     path_method = line[3] # group(4) of the original match
...     if path_method.startswith("path"):
```

3. If the path is not in the dictionary, we need to add it. Once the value of the path\_method string is in the dictionary, we can increment the value in the dictionary, based on the key from the data:

```
...         if path_method not in histogram:
...             histogram[path_method] = 0
...             histogram[path_method] += 1
```

This technique adds each new path\_method value to the dictionary. Once it has been established that the path\_method key is in the dictionary, we can increment the value associated with the key.

## Building a dictionary as a comprehension

The last field of each log line had one or two fields inside. There may have been a value like path="/openapi.yaml" method=GET with two attributes, path and method, or a value like error="invalid query" with only one attribute, error.

We can use the following regular expression to decompose the final field of each line:

```
param_parser = re.compile(
    r'(\w+)=([".*?"]|\w+)'
)
```

The findall() method of this regular expression will provide a sequence of two-tuples based on the matching text. We can then build a dictionary from the sequence of matched groups:

1. For each of the log lines, apply the regular expression to create a sequence of pairs:

```
>>> for line in log_lines:
...     name_value_pairs = param_parser.findall(line[3])
```

2. Use a dictionary comprehension to use the first matching group as the key and the second matching group as the value:

```
...     params = {match[0]: match[1] for match in name_value_pairs}
```

We can print the `params` values and we'll see the dictionaries like the following examples:

```
{'path': '/openapi.yaml', 'method': 'GET'}
{'path': '/items?limit=x', 'method': 'GET'}
{'error': '"invalid query"'}
```

Using a dictionary for the final fields of each log record makes it easier to separate the important pieces of information.

## How it works...

The core feature of a dictionary is a mapping from an immutable key to a value object of any kind. In the first example, we've used an immutable string as the key, and an integer as the value. We describe it as `dict[str, int]` in the type hint.

It's important to understand how the `+=` assignment statement works. The implementation of `+=` is essentially this:

```
histogram[customer] = histogram[customer] + 1
```

The `histogram[customer]` value is fetched from the dictionary, a new value is computed, and the result is used to update the dictionary.

It's essential that dictionary key objects be immutable. We cannot use a list, set, or dictionary as the key in a dictionary mapping. We can, however, transform a list into an immutable

tuple, or make a set into a frozenset so that we can use one of these more complex objects as a key. In the examples shown in this recipe, we had immutable strings as the keys to each dictionary.

## There's more...

We don't have to use an `if` statement to add missing keys. We can use the `setdefault()` method of a dictionary instead. It's even easier to use one of the classes from the `collections` module.

Here's the version using the `defaultdict` class from the `collections` module:

```
>>> from collections import defaultdict

>>> histogram = defaultdict(int)
>>> for line in log_lines:
...     path_method = line[3] # group(4) of the match
...     if path_method.startswith("path"):
...         histogram[path_method] += 1
```

We've created a `defaultdict` instance that will initialize any unknown key values using the `int()` function. We provide `int`—the function object—to the `defaultdict` constructor. The `defaultdict` instance will evaluate the given function to create default values.

This allows us to use `histogram[path_method] += 1`. If the value associated with the `path_method` key was previously in the dictionary, the value will be incremented and put back into the dictionary. If the `path_method` key was not in the dictionary, the `int()` function is called with no argument; this default value will be incremented and put into the dictionary.

The other way we can accumulate frequency counts is by creating a `Counter` object. We can build the `Counter` object from the raw data as follows:

```
>>> from collections import Counter

>>> filtered_paths = (
```

```
...     line[3]
...     for line in log_lines
...     if line[3].startswith("path")
... )
>>> histogram = Counter(filtered_paths)
>>> histogram
Counter({'path="/openapi.yaml" method=GET': 1, 'path="/items?limit=x"
method=GET': 1, 'path="/category/42" method=GET': 1})
```

First, we used a generator expression to create an iterator over the filtered path data; this was assigned to `filtered_paths`. Then we created a `Counter` from the source of data; the `Counter` class will scan the data and count the distinct occurrences.

## See also

- In the *Shrinking dictionaries – the `pop()` method and the `del` statement* recipe, we'll look at how dictionaries can be modified by removing items.

## Shrinking dictionaries – the `pop()` method and the `del` statement

A common use case for a dictionary is as an **associative store**: it keeps an association between key and value objects. This means that we may be doing any of the **CRUD** operations on an item in the dictionary:

- Create a new key and value pair.
- Retrieve the value associated with a key.
- Update the value associated with a key.
- Delete the key (and the corresponding value) from the dictionary.

## Getting ready

A great deal of processing supports the need to group items around one (or more) different common values. We'll return to the log data shown in the *Creating dictionaries – inserting*

and *updating* recipe in this chapter.

We'll use an iterator algorithm that uses the transaction ID as a key in a dictionary. The value for this key will be the sequence of steps for the transaction. With a very long log, we don't — generally — want to save every transaction in a gigantic dictionary. When we reach the termination of a transaction's sequence, we can yield the list of log entries for the transaction. A function can consume this iterator, processing each batch of transactions independently.

## How to do it...

The context for this recipe will require an `if` statement with the condition `match := log_parser.match(line)`. This will apply the regular expression, and collect the result in the `match` variable. Given that context, the processing to update or delete from a dictionary is as follows:

1. This function uses the `defaultdict` class, and two additional type hints, `Iterable` and `Iterator`:

```
from collections import defaultdict
from collections.abc import Iterable, Iterator
```

2. Define a `defaultdict` object to hold transaction steps. The keys are 20-character strings. The values are lists of log records. In this case, each log record will have been parsed from the source text into a tuple of individual strings:

```
LogRec = tuple[str, ...]

def request_iter_t(source: Iterable[str]) -> Iterator[list[LogRec]]:
    requests: defaultdict[str, list[LogRec]] = defaultdict(list)
```

3. Define the key for each cluster of log entries:

```
for line in source:
    if match := log_parser.match(line):
        id = match.group(3)
```

4. Update a dictionary item with a log record:

```
requests[id].append(tuple(match.groups()))
```

5. If this log record completes a transaction, yield the group as part of a generator function. Then remove the transaction from the dictionary, since it's complete:

```
if match.group(4).startswith('status'):
    yield requests[id]
    del requests[id]
```

6. At the end, there may be a non-empty requests dictionary. This reflects a transaction that was in process when the log file was switched.

## How it works...

Because a dictionary is a mutable object, we can remove keys from a dictionary. A `del` statement will delete both the key and the value object associated with the key. In this example, the key is removed when the data reveals the transaction is complete. A moderately busy web server handling an average of 10 transactions per second will see 864,000 transactions in a 24-hour period. If there are an average of 2.5 log entries per transaction, there will be at least 2,160,000 lines in the file.

If we only want to know the elapsed time per resource, we don't want to keep the entire dictionary of 864,000 transactions in memory. We'd rather transform the log into an intermediate summary file for further analysis.

This idea of transient data leads us to accumulate the parsed log lines into a list instance. Each new line is appended to the appropriate list for the transaction in which the line belongs. When the final line has been found, the group of lines can be purged from the dictionary.

## There's more...

In the example, we used the `del` statement. The `pop()` method can also be used. The `del` statement will raise a `KeyError` exception if the given item cannot be found in the

dictionary.

The `pop()` method would look like this:

```
requests.pop(id)
```

This will mutate the dictionary in place, removing the item if it exists, or raising a `KeyError` exception.

The `pop()` method, when provided with a default value, can return the given default value instead of raising an exception when a key is not found. In either case, the key will no longer be in the dictionary. Note that this method both mutates the collection *and* returns a value.

The `popitem()` method will remove a key and value pair from the dictionary. The pairs are returned in **Last-In-First-Out (LIFO)** order. This means a dictionary is also a kind of stack.

## See also

- In the *Creating dictionaries – inserting and updating* recipe, we look at how we create dictionaries and fill them with keys and values.

## Writing dictionary-related type hints

When we look at sets and lists, we generally expect each item within a list (or a set) to be the same type. When we look at object-oriented class designs, in *Chapter 7*, we'll see how a common superclass can be the common type for a closely related family of object types. While it's possible to have heterogeneous types in a list or set collection, it often becomes quite complex to process, requiring the `match` statement to do proper type matching. A dictionary, however, can be used to create a *discriminated union* of types. A particular key value may be used to define which other keys are present in the dictionary. This means a simple `if` statement can discriminate between heterogeneous types.



## Getting ready

We'll look at two kinds of dictionary type hints, one for homogeneous value types and the other for heterogeneous value types. We'll look at data that starts out as one of these kinds of dictionaries but is transformed to have more complex type definitions.

We'll be starting with the following CSV file:

```
date,engine on,fuel height on,engine off,fuel height off
10/25/13,08:24:00,29,13:15:00,27
10/26/13,09:12:00,27,18:25:00,22
10/28/13,13:21:00,22,06:25:00,14
```

This describes three separate legs of a multi-day trip on a sailboat. The fuel is measured by the height in the tank, rather than some indirect method using a float or other gauges. Because the tank is approximately rectangular, 31 inches of depth is about 75 gallons of fuel.

## How to do it...

The initial use of `csv.DictReader` will lead to dictionaries with homogeneous type definitions:

1. Locate the type of the keys in the dictionary. When reading CSV files, the keys are strings, with the type `str`.
2. Locate the type of the values in the dictionary. When reading CSV files, the values are strings, with the type `str`.
3. Combine the types using the `dict` type hint. This yields `dict[str, str]`.

Here's an example function for reading data from a CSV file:

```
import csv
from pathlib import Path

def get_fuel_use(source_path: Path) -> list[dict[str, str]]:
```

```
with source_path.open() as source_file:
    rdr = csv.DictReader(source_file)
    data: list[dict[str, str]] = list(rdr)
return data
```

The `get_fuel_use()` function yields values that match the source data. In this case, it's a dictionary that maps string column names to string cell values.

This data, by itself, is difficult to work with. A common second step is to apply transformations to the source rows to create more useful data types. We can describe the results with a type hint:

1. Identify the various value types that will be needed. In this example, there are five fields with three different types, shown here:
  - The `date` field is a `datetime.date` object.
  - The `engine_on` field is a `datetime.time` object.
  - The `fuel_height_on` field is an integer, but we know that it will be used in a float context, so we'll create a float directly.
  - The `engine_off` field is a `datetime.time` object.
  - The `fuel_height_off` field is also a float value.
2. Import the `TypedDict` type definition from the `typing` module.
3. Define the subclass of `TypedDict` with the new heterogeneous dictionary types:

```
import datetime
from typing import TypedDict

class History(TypedDict):
    date: datetime.date
    start_time: datetime.time
    start_fuel: float
    end_time: datetime.time
    end_fuel: float
```

This is, in part, a teaser for *Chapter 7*. It shows a very simple kind of class definition. In this case, the class is dictionaries with five specific keys, all of which are required and must have values of the given types.

In this example, we've also renamed the fields to make them into names that are valid Python names. Replacing punctuation with `_` is the obvious first step. We also changed a few because the column names in the CSV file seemed awkward.

The function to perform the transformation can look like the following example:

```
from collections.abc import Iterable, Iterator

def make_history(source: Iterable[dict[str, str]]) -> Iterator[History]:
    for row in source:
        yield dict(
            date=datetime.datetime.strptime(
                row['date'], "%m/%d/%y").date(),
            start_time=datetime.datetime.strptime(
                row['engine on'], '%H:%M:%S').time(),
            start_fuel=float(row['fuel height on']),
            end_time=datetime.datetime.strptime(
                row['engine off'], '%H:%M:%S').time(),
            end_fuel=float(row['fuel height off']),
        )
```

This function consumes instances of the initial `dict[str, str]` dictionary and creates instances of the dictionary described by the `History` class. Here's how these two functions work together:

```
>>> from pprint import pprint

>>> source_path = Path("data/fuel2.csv")
>>> fuel_use = make_history(get_fuel_use(source_path))
>>> for row in fuel_use:
...     pprint(row)
{'date': datetime.date(2013, 10, 25),
 'end_fuel': 27.0,
 'end_time': datetime.time(13, 15),
 'start_fuel': 29.0,
```

```
'start_time': datetime.time(8, 24)}
{'date': datetime.date(2013, 10, 26),
 'end_fuel': 22.0,
 'end_time': datetime.time(18, 25),
 'start_fuel': 27.0,
 'start_time': datetime.time(9, 12)}
{'date': datetime.date(2013, 10, 28),
 'end_fuel': 14.0,
 'end_time': datetime.time(6, 25),
 'start_fuel': 22.0,
 'start_time': datetime.time(13, 21)}
```

This shows how the output from the `get_fuel_use()` function can be processed by the `make_history()` function to create an iterable sequence of dictionaries. Each of the resulting dictionaries has the source data converted to a more useful type.

## How it works...

The core type hint for a dictionary names the key type and the value type, in the form `dict[key, value]`. The `TypedDict` class lets us be more specific about bindings between dictionary keys and a broad domain of values.

It's important to note that type hints are only checked by programs like **mypy**. These hints have no runtime impact. We could, for example, write a statement like the following:

```
result: History = {'date': 42}
```

This statement claims that the result dictionary will match the type hints in the `History` type definition. The dictionary literal, however, has the wrong type for the `'date'` field and a number of other fields are missing. While this will execute, it will raise errors from **mypy**.

Running the **mypy** program reveals the error as shown in the following listing:

```
(cookbook3) % python -m mypy src/ch05/recipe_04_bad.py
src/ch05/recipe_04_bad.py:18: error: Missing keys ("start_time", "start_fuel", "
    end_time", "end_fuel") for TypedDict "History" [typeddict-item]
```

Found 1 error in 1 file (checked 1 source file)

For run-time validation of data, a project like **Pydantic** can be very helpful.

## There's more...

One of the common cases for heterogeneity in dictionary keys is optional items. The type hint `Optional[str]` or `str | None` describes this. This is rarely needed with a dictionary, since it can be simpler to omit the *key-value* pair entirely.

Let's assume we need two variants of the `History` type:

- The variant shown earlier in this recipe, where all fields are present.
- Two “incomplete” records, one without an engine off time and ending fuel height, and another variant without an engine on time or starting fuel height. These two records might be used for an overnight passage under power.

In this case, we might need to use the `NotRequired` annotation for these fields. The resulting class definition would look like this:

```
from typing import TypedDict, NotRequired

class History2(TypedDict):
    date: datetime.date
    start_time: NotRequired[datetime.time]
    start_fuel: NotRequired[float]
    end_time: NotRequired[datetime.time]
    end_fuel: NotRequired[float]
```

This record permits a great deal of variability in the dictionary values. It requires the use of `if` statements to determine the mix of fields present in the data. Furthermore, it also requires somewhat more clever processing in the `make_history()` function to create these variant records based on empty columns in the CSV file.

There are some parallels between `TypedDict` and the `NamedTuple` type definitions. Changing `TypedDict` to `NamedTuple` will create a named tuple class instead of a typed dictionary class.

Because a `NamedTuple` class has an `_asdict()` method, it's possible to produce a dictionary that matches the `TypedDict` structure from a named tuple.

A dictionary that matches the `TypedDict` hint is mutable. A subclass of `NamedTuple`, however, is immutable. This is one central difference between these two type hints. More importantly, a dictionary uses `row['date']` syntax to refer to one item using the key 'date'. A named tuple uses `row.date` syntax to refer to one item using a name.

## See also

- The *Using NamedTuples to simplify item access in tuples* recipe provides more details on the `NamedTuple` type hint.
- See the *Writing list-related type hints* recipe in *Chapter 4* for more about type hints for lists.
- The *Writing set-related type hints* recipe, also in *Chapter 4*, covers this from the view of set types.
- For runtime validation of data, a project like **Pydantic** can be very helpful. See <https://docs.pydantic.dev/latest/>.

## Understanding variables, references, and assignment

How do variables really work? What happens when we assign a mutable object to two variables? When two variables are sharing references to a common mutable object, the behaviors can be confusing.



This is the core principle: **Python shares references; it doesn't copy data.**

To see what this rule on reference sharing means, we'll create two data structures: one is mutable and one is immutable.

## Getting ready

We'll look at the two kinds of sequences, although we could do something similar with two kinds of sets:

```
>>> mutable = [1, 1, 2, 3, 5, 8]
>>> immutable = (5, 8, 13, 21)
```

We'll look at what happens when references to these objects are shared.

We can do a similar comparison with a set and a frozenset. We can't easily do this with a mapping because Python doesn't offer a handy immutable mapping.

## How to do it...

This recipe will show how to observe the “spooky action at a distance” when there are two references to an underlying mutable object. We'll look at ways to prevent this in the *Making shallow and deep copies of objects* recipe. Here are the steps for seeing the difference between mutable and immutable collections:

1. Assign each collection to an additional variable. This will create two references to the structure:

```
>>> mutable_b = mutable
>>> immutable_b = immutable
```

We now have two references to the list [1, 1, 2, 3, 5, 8] and two references to the tuple (5, 8, 13, 21).

2. We can confirm this using the `is` operator. This determines if two variables refer to the same underlying object:

```
>>> mutable_b is mutable
True
>>> immutable_b is immutable
True
```

3. Make a change to one of the two references to the collection. For the list type, we

have methods like `extend()` or `append()`. For this example we'll use the `+` operator:

```
>>> mutable += [mutable[-2] + mutable[-1]]
```

We can do a similar thing with the immutable structure:

```
>>> immutable += (immutable[-2] + immutable[-1],)
```

4. Look at the other two variables that reference the mutable structure. Because the two variables are references to the same underlying list object, each variable shows the current state:

```
>>> mutable_b
[1, 1, 2, 3, 5, 8, 13]
>>> mutable is mutable_b
True
```

5. Look at the two variables referring to immutable structures. Initially, the two variables shared a common object. When the assignment statement was executed, a new tuple was created and only one variable changed to refer to the new tuple:

```
>>> immutable_b
(5, 8, 13, 21)
>>> immutable
(5, 8, 13, 21, 34)
```

## How it works...

The two variables, `mutable` and `mutable_b`, still refer to the same underlying object. Because of that, we can use either variable to change the object and see the change reflected in the other variable's value.

The two variables, `immutable_b` and `immutable`, started out referring to the same object. Because the object cannot be mutated in place, a change to one variable means that a new object is assigned to that variable. The other variable remains firmly attached to the original object.



In Python, a variable is a label that's attached to an object. We can think of them like adhesive notes in bright colors that we stick on an object temporarily. Multiple labels can be attached to an object. It's the assignment statement that places a variable name on an object.

Consider the following statement:

```
immutable += (immutable[-2] + immutable[-1],)
```

This has the same effect as this statement:

```
immutable = immutable + (immutable[-2] + immutable[-1],)
```

The expression on the right side of = creates a new tuple from the previous value of the `immutable` tuple. The assignment statement then assigns the label `immutable` to the newly-minted object.

Assigning to a variable has two possible actions:

- For mutable objects that provide definitions for appropriate in-place assignment operators like +=, the assignment is transformed into a special method; in this case, `__iadd__()`. The special method will mutate the object's internal state.
- For immutable objects that do not provide definitions for assignment like +=, the assignment is transformed into = and +. A new object is built by the + operator and the variable name is attached to that new object. Other variables that previously referred to the object being replaced are not affected; they will continue to refer to old objects.

Python counts the number of places from which an object is referenced. When the count of references becomes zero, the object is no longer used anywhere and can be removed from memory.

## There's more...

Some languages have primitive types in addition to objects. In these languages, a `+=` statement may leverage a feature of the hardware instructions to tweak the value of a primitive type.

Python doesn't have this kind of optimization. Numbers are immutable objects; there are no special instructions to tweak their values. Consider the following assignment statements:

```
a = 355
a += 113
```

The processing does not tweak the internal state of the object 355. The `int` class does not provide an `__iadd__()` special method. A new immutable integer object is created. This new object is given the label `a`. The old value previously assigned to `a` is no longer needed, and the storage can be reclaimed.

## See also

- In the *Making shallow and deep copies of objects* recipe, we'll look at ways we can copy mutable structures to prevent shared references.
- Also, see *Avoiding mutable default values for function parameters* for another consequence of the way references are shared in Python.
- For the CPython implementation, a few objects can be immortal. See PEP 683 for more on this implementation detail.

## Making shallow and deep copies of objects

Throughout this chapter, we've talked about how assignment statements share references to objects. Objects are not normally copied.

Consider this assignment statement:

```
a = b
```

This creates two references to the same underlying object. If the value of the `b` variable has a mutable type, like the list, set, or dict types, then a change using either `a` or `b` will update the underlying mutable object. For more background, see the *Understanding variables, references, and assignment* recipe.

Most of the time, this is the behavior we want. This is ideal for providing mutable objects to functions and having a local variable in the function mutate an object created outside the function. There are rare situations in which we want to actually have two independent objects created from one original object.

There are two ways to break the connection that exists when two variables are references to the same underlying object:

- Making a shallow copy of the structure
- Making a deep copy of the structure

## Getting ready

Python does not automatically make a copy of an object. We've seen several kinds of syntax for making a copy:

- Sequences – list, as well as the str, bytes, and tuple types: we can use `sequence[:]` to copy a sequence by using an empty slice expression. This is a special case for sequences.
- Almost all collections have a `copy()` method.
- Calling a type, with an instance of the type as the only argument, returns a copy. For example, if `d` is a dictionary, `dict(d)` will create a shallow copy of `d`.

What's important is that these are all *shallow* copies. When two collections are shallow copies, they each contain references to the same underlying objects. If the underlying objects are immutable, such as tuples, numbers, or strings, this distinction doesn't matter.

For example, if we have `a = [1, 1, 2, 3]`, we can't perform any mutation on `a[0]`. The number 1 in `a[0]` has no internal state. We can only replace the object.

Questions arise, however, when we have a collection that involves mutable objects. First, we'll create an object, then we'll create a copy:

```
>>> some_dict = {'a': [1, 1, 2, 3]}
>>> another_dict = some_dict.copy()
```

This example created a shallow copy of the dictionary. The two copies will look alike because they both contain references to the same objects. There's a shared reference to the immutable string 'a' and a shared reference to the mutable list `[1, 1, 2, 3]`. We can display the value of `another_dict` to see that it looks like the `some_dict` object we started with:

```
>>> another_dict
{'a': [1, 1, 2, 3]}
```

Here's what happens when we update the shared list that's inside the copy of the dictionary. We'll change the value of `some_dict` and see the results are also present in `another_dict`:

```
>>> some_dict['a'].append(5)
>>> another_dict
{'a': [1, 1, 2, 3, 5]}
```

We can see that the item is shared by using the `id()` function:

```
>>> id(some_dict['a']) == id(another_dict['a'])
True
```

Because the two `id()` values are the same, these are the same underlying object. The value associated with the key 'a' is the same mutable list in both `some_dict` and `another_dict`. We can also use the `is` operator to see that they're the same object.

This mutation of a shallow copy works for list collections that contain all other mutable

object types as items as well:

Because we can't make a set of mutable objects, we don't really have to consider making shallow copies of sets that share items.



A tuple can contain mutable objects. While the tuple is immutable, the objects inside it are mutable.

The immutability of a tuple does not magically propagate to the items within the tuple.

What if we want to completely disconnect two copies? How do we make a deep copy instead of a shallow copy?

## How to do it...

Python generally works by sharing references. It makes copies of objects reluctantly. The default behavior is to make a shallow copy, sharing references to the items within a collection. Here's how we make deep copies:

1. Import the copy module:

```
>>> import copy
```

2. Use the `copy.deepcopy()` function to duplicate an object and all of the mutable items contained within that object:

```
>>> some_dict = {'a': [1, 1, 2, 3]}
>>> another_dict = copy.deepcopy(some_dict)
```

This will create copies that have no shared references. A change to one copy's mutable internal items won't have any effect anywhere else:

```
>>> some_dict['a'].append(5)
>>> some_dict
{'a': [1, 1, 2, 3, 5]}
>>> another_dict
{'a': [1, 1, 2, 3]}
```

We updated an item in `some_dict` and it had no effect on the copy in `another_dict`. We can see that the objects are distinct with the `id()` function:

```
>>> id(some_dict['a']) == id(another_dict['a'])
False
```

Since the `id()` values are different, these are distinct objects. We can also use the `is` operator to see that they're distinct objects.

## How it works...

Making a shallow copy is relatively easy. We can even write our own version of the algorithm using comprehensions (containing generator expressions):

```
>>> copy_of_list = [item for item in some_list]
>>> copy_of_dict = {key:value for key, value in some_dict.items()}
```

In the list case, the items for the new list are references to the items in the source list. Similarly, in the dict case, the keys and values are references to the keys and values of the source dictionary.

The `deepcopy()` function uses a recursive algorithm to look inside each item that's a mutable collection.

For an object with a list type, the conceptual algorithm is something like this:

```
from typing import Any

def deepcopy_json(some_obj: Any) -> Any:
    match some_obj:
        case int() | float() | tuple() | str() | bytes() | None:
            return some_obj
        case list() as some_list:
            list_copy: list[Any] = []
            for item in some_list:
                list_copy.append(deepcopy_json(item))
            return list_copy
        case dict() as some_dict:
            dict_copy: dict[Any, Any] = {}
            for key in some_dict:
                dict_copy[key] = deepcopy_json(some_dict[key])
            return dict_copy
        case _:
            raise ValueError(f"can't copy {type(some_obj)}")
```

This can be used for the collection of types used in JSON documents. For the immutable types in the first case clause, there's no need to make a copy; an object of one of these types cannot be mutated. For the two mutable types used in JSON documents, empty structures are built, and then copies of each item are inserted. The processing involves recursion to assure that – no matter how deeply nested – all items that are mutable are copied.

The actual implementation of the `deepcopy()` function handles additional types, not part of the JSON specification. The point of this example is to show the general idea of a deep copy function.

## See also

- In the *Understanding variables, references, and assignment* recipe, we look at how Python prefers to create references to objects.

## Avoiding mutable default values for function parameters

In *Chapter 3*, we looked at many aspects of Python function definitions. In the *Designing functions with optional parameters* recipe, we showed a recipe for handling optional parameters. At the time, we didn't dwell on the issue of providing a reference to a mutable structure as a default. We'll take a close look at the consequences of a mutable default value for a function parameter.

### Getting ready

Let's imagine a function that either creates or updates a mutable Counter object. We'll call it `gather_stats()`.

Ideally, a small data gathering function could look like this:

```
from collections import Counter
from random import randint, seed

def gather_stats_bad(
    n: int,
    samples: int = 1000,
    summary: Counter[int] = Counter()
) -> Counter[int]:
    summary.update(
        sum(randint(1, 6)
            for d in range(n)) for _ in range(samples)
    )
    return summary
```

This shows a *bad* design for a function. It has two scenarios:

1. The first scenario offers no argument value for the `summary` parameter. When this is omitted, the function creates and returns a collection of statistics. Here's the example of this story:



```
>>> seed(1)
>>> s1 = gather_stats_bad(2)
>>> s1
Counter({7: 168, 6: 147, 8: 136, 9: 114, 5: 110, 10: 77, 11: 71, 4: 70,
3: 52, 12: 29, 2: 26})
```

2. The second scenario allows us to provide an explicit argument value for the summary parameter. When this argument is provided, this function updates the given object. Here's an example of this story:

```
>>> seed(1)
>>> mc = Counter()
>>> gather_stats_bad(2, summary=mc)
Counter...
>>> mc
Counter({7: 168, 6: 147, 8: 136, 9: 114, 5: 110, 10: 77, 11: 71, 4: 70,
3: 52, 12: 29, 2: 26})
```

We've set the random number seed to be sure that the two sequences of random values are identical. We provided a Counter object to confirm that the results are identical.

The problem arises when we do the following operation after the first scenario shown above:

```
>>> seed(1)
>>> s3b = gather_stats_bad(2)
>>> s3b
Counter({7: 336, 6: 294, 8: 272, 9: 228, 5: 220, 10: 154, 11: 142, 4: 140,
3: 104, 12: 58, 2: 52})
```

The values in this example are incorrect. They're doubled. Something has gone wrong. This only happens when we use the default scenario more than once. This code can pass a simple unit test suite and *appear* correct.

As we saw in the *Making shallow and deep copies of objects* recipe, Python prefers to share

references. A consequence of that sharing is the object referenced by the `s1` variable and the object referenced by the `s3b` variable are the same object:

```
>>> s1 is s3b
True
```

This means the value of the object referred to by the `s1` variable changed when the object for the `s3b` variable was created. From this, it should be apparent the function is updating a single, shared collection object and returning the reference to the shared collection.

The default value used for the `summary` parameter of this `gather_stats_bad()` function leads to result values built from a single, shared object. How can we avoid this?

## How to do it...

There are two approaches to solving this problem of a mutable default parameter:

- Provide an immutable default.
- Change the design.

We'll look at the immutable default first. Changing the design is generally a better idea. In order to see why it's better to change the design, we'll show a purely technical solution.

When we provide default values for functions, the default object is created exactly once and shared forever after. Here's the alternative:

1. Replace any mutable default parameter value with `None`:

```
def gather_stats_good(
    n: int,
    samples: int = 1000,
    summary: Counter[int] | None = None
) -> Counter[int]:
```

```
def gather_stats_good(
    n: int,
    summary: Counter[int] | None = None,
```

```
    samples: int = 1000,  
    ) -> Counter[int]:
```

2. Add an `if` statement to check for an argument value of `None` and replace it with a fresh, new mutable object of the proper type:

```
    if summary is None:  
        summary = Counter()
```

This will assure us that every time the function is evaluated with no argument value for a parameter, we create a fresh, new mutable object. We will avoid sharing a single mutable object over and over again.

## How it works...

As we noted earlier, Python prefers to share references. It rarely creates copies of objects without explicit use of the `copy` module or the `copy()` method of an object. Therefore, default values for function parameter values will be shared objects. Python does not create fresh, new objects for default parameter values.



Never use mutable defaults for default values of function parameters.

Instead of a mutable object (for example, `set`, `list`, or `dict`) as a default, use `None`.

In most cases, we should consider changing the design to not offer a default value at all. Instead, define two separate functions. One function updates a parameter value, and a second function uses this function but provides a fresh, empty mutable object.

For this example, they might be called `create_stats()` and `update_stats()`, with unambiguous parameters:

```
def update_stats(
    n: int,
    summary: Counter[int],
    samples: int = 1000,
) -> Counter[int]:
    summary.update(
        sum(randint(1, 6)
            for d in range(n) for _ in range(samples))
    )
    return summary

def create_stats(n: int, samples: int = 1000) -> Counter[int]:
    return update_stats(n, Counter(), samples)
```

Note that the `summary` parameter to the `update_stats()` function is not optional. Similarly, there is no `summary` object parameter defined for the `create_stats()` function.

The idea of optional mutable arguments was not a good idea because the mutable object provided as a default value for a parameter is reused.

## There's more...

In the standard library, there are some examples of a cool technique that shows how we can create fresh default objects. A number of places use a *factory function* as a parameter. This function can be used to create a fresh, new mutable object.

In order to leverage this design pattern, we need to modify the design of our `update_stats()` function. We will no longer update an existing `Counter` object in the function. We'll always create a fresh, new object.

Here's a function that calls a factory function to create a useful default value:

```
from collections import Counter
from collections.abc import Callable, Iterable, Hashable
from typing import TypeVar, TypeAlias

T = TypeVar('T', bound=Hashable)
Summarizer: TypeAlias = Callable[[Iterable[T]], Counter[T]]
```

```
def gather_stats_flex(
    n: int,
    samples: int = 1000,
    summary_func: Summarizer[int] = Counter
) -> Counter[int]:
    summary = summary_func(
        sum(randint(1, 6)
            for d in range(n) for _ in range(samples))
    )
    return summary
```

For this version, we've defined the `Summarizer` type to be a function of one argument that will create a `Counter` object. The default value uses the `Counter` class as the one-argument function. We can override the `summary_func` function with any one-argument function that will collect details instead of summarizing.

Here's an example using `list` instead of `collections.Counter`:

```
>>> seed(1)
>>> gather_stats_flex(2, 12, summary_func=list)
[7, 4, 5, 8, 10, 3, 5, 8, 6, 10, 9, 7]
```

In this case, we provided the `list` function to create a list with the individual random samples in it.

Here's an example without an argument value. It will create a new `collections.Counter` object each time it's used:

```
>>> seed(1)
>>> gather_stats_flex(2, 12)
Counter({7: 2, 5: 2, 8: 2, 10: 2, 4: 1, 3: 1, 6: 1, 9: 1})
```

In this case, we've evaluated the function using the default value for `summary_func`, which creates a `collections.Counter` object from the random samples.

## See also

- See the *Creating dictionaries – inserting and updating* recipe, which shows how the `defaultdict` collection works.

## Join our community Discord space

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

<https://packt.link/dHrHU>





# 6

## User Inputs and Outputs

The key purpose of software is to produce useful output. Of the many possible outputs, one simple type of output is text displaying a useful result. Python supports this with the `print()` function.

The `input()` function has a parallel with the `print()` function. The `input()` function reads text from a console, allowing us to provide data to our programs. The use of `print()` and `input()` creates an elegant symmetry between input and output from an application.

There are a number of other common ways to provide input to a program. Parsing the command line is helpful for many applications. We sometimes need to use configuration files to provide useful input. Data files and network connections are yet more ways to provide input. Each of these methods is distinct and needs to be looked at separately. In this chapter, we'll focus on the fundamentals of `input()` and `print()`.

In this chapter, we'll look at the following recipes:

- *Using the features of the `print()` function*
- *Using `input()` and `getpass()` for user input*



- *Debugging with f"{value=}" strings*
- *Using argparse to get command-line input*
- *Using invoke to get command-line input*
- *Using cmd to create command-line applications*
- *Using the OS environment settings*

It seems best to start with the `print()` function and show several of the things it can do. After all, it's often the output from an application that is most useful.

## Using the features of the `print()` function

In many cases, the `print()` function is the first function we learn about. The first script is often a variation on the following:

```
>>> print("Hello, world.")
Hello, world.
```

The `print()` function can display multiple values, with helpful spaces between items.

When we write this:

```
>>> count = 9973
>>> print("Final count", count)
Final count 9973
```

We can see that a space separator is included for us. Additionally, a line break, usually represented by the `\n` character, is printed after the values provided in the function.

Can we control this formatting? Can we change the extra characters that are supplied?

## Getting ready

Consider this spreadsheet, used to record fuel consumption on a large sailboat. The CSV file has rows that look like this:

```
date,engine on,fuel height on,engine off,fuel height off
10/25/13,08:24:00,29,13:15:00,27
10/26/13,09:12:00,27,18:25:00,22
10/28/13,13:21:00,22,06:25:00,14
```

For more information on this data, refer to the *Shrinking sets – remove(), pop(), and difference* and *Slicing and dicing a list* recipes in *Chapter 4*. Instead of a sensor inside the tank, the depth of fuel is observed through a glass panel on the side of the tank. Knowing the tank is approximately rectangular, with a full depth of about 31 inches and a volume of about 72 gallons, it's possible to convert depth to volume.

Here's an example of using this CSV data. This function reads the file and returns a list of fields built from each row:

```
from pathlib import Path
import csv

def get_fuel_use(source_path: Path) -> list[dict[str, str]]:
    with source_path.open() as source_file:
        rdr = csv.DictReader(source_file)
        return list(rdr)
```

Here's an example of reading and printing rows from the CSV file:

```
>>> source_path = Path("data/fuel2.csv")
>>> fuel_use = get_fuel_use(source_path)
>>> for row in fuel_use:
...     print(row)
{'date': '10/25/13', 'engine on': '08:24:00', 'fuel height on': '29',
'engine off': '13:15:00', 'fuel height off': '27'}
{'date': '10/26/13', 'engine on': '09:12:00', 'fuel height on': '27',
'engine off': '18:25:00', 'fuel height off': '22'}
{'date': '10/28/13', 'engine on': '13:21:00', 'fuel height on': '22',
'engine off': '06:25:00', 'fuel height off': '14'}
```

The output from the `print()` function, shown here in long lines, is challenging to use. Let's look at how we can improve this output using additional features of the `print()` function.

## How to do it...

We have two ways to control the `print()` function output format:

- Set the inter-field separator string, `sep`. The default value is a single-space character.
- Set the end-of-line string, `end`. The default value is the `\n` character.

This recipe will show several variations:

1. Read the data:

```
>>> fuel_use = get_fuel_use(Path("data/fuel2.csv"))
```

2. For each item in the data, make any useful data conversions:

```
>>> for leg in fuel_use:  
...     start = float(leg["fuel height on"])  
...     finish = float(leg["fuel height off"])
```

3. The following alternatives show different ways to include separators:

- Print labels and fields using the default values of `sep` and `end`:

```
...     print("On", leg["date"], "from", leg["engine on"],  
...           "to", leg["engine off"],  
...           "change", start-finish, "in.")  
On 10/25/13 from 08:24:00 to 13:15:00 change 2.0 in.  
On 10/26/13 from 09:12:00 to 18:25:00 change 5.0 in.  
On 10/28/13 from 13:21:00 to 06:25:00 change 8.0 in.
```

When we look at the output, we can see where a space was inserted between each item.

- When preparing data, we might want to use a format that's similar to CSV, perhaps using a column separator that's not a simple comma. We can print labels and fields using a string value of `" | "` for the `sep` parameter:

```
...     print(leg["date"], leg["engine on"],
...           leg["engine off"], start-finish, sep=" | ")
10/25/13 | 08:24:00 | 13:15:00 | 2.0
10/26/13 | 09:12:00 | 18:25:00 | 5.0
10/28/13 | 13:21:00 | 06:25:00 | 8.0
```

In this case, we can see that each column has the given separator string. Since there were no changes to the end setting, each `print()` function produces a distinct line of output.

- Here's how we might change the default punctuation to emphasize the field name and value. We can print labels and fields using a string value of "=" for the `sep` parameter and ", " for the `end` parameter:

```
...     print("date", leg["date"], sep="=", end=", ")
...     print("on", leg["engine on"], sep="=", end=", ")
...     print("off", leg["engine off"], sep="=", end=", ")
...     print("change", start-finish, sep="=")
date=10/25/13, on=08:24:00, off=13:15:00, change=2.0
date=10/26/13, on=09:12:00, off=18:25:00, change=5.0
date=10/28/13, on=13:21:00, off=06:25:00, change=8.0
```

Since the string used at the end of the line was changed to ", ", each use of the `print()` function no longer produces separate lines. In order to see a proper end of line, the final `print()` function has a default value for `end`. We could also have used an argument value of `end="\n"` to make the presence of the newline character explicit.

## How it works...

The `print()` function has a definition that includes several parameters that must be provided as keywords. Two of these are the `sep` and `end` keyword parameters, with default values of space and newline, respectively.

Using the `print()` function's `sep` and `end` parameters can get quite complex for anything more sophisticated than these simple examples. Rather than working with a complex

sequence of `print()` function requests, we can use the `format()` method of a string, or use an f-string.

## There's more...

The `sys` module defines the two standard output files that are always available: `sys.stdout` and `sys.stderr`. Generally, the `print()` function can be thought of as a handy wrapper around `sys.stdout.write()`.

We can use the `file=` keyword argument to write to the standard error file instead of writing to the standard output file:

```
>>> import sys
>>> print("Red Alert!", file=sys.stderr)
```

We've imported the `sys` module so that we have access to the standard error file. We used this to write a message that will not be part of the standard output stream.

Because these two files are always available, using OS file redirection techniques often works out nicely. When our program's primary output is written to `sys.stdout`, it can be redirected at the OS level. A user might enter a shell command line like this:

```
% python myapp.py < input.dat > output.dat
```

This will provide the `input.dat` file as the input to `sys.stdin`. When this Python program writes to `sys.stdout`, the output will be redirected by the OS to the `output.dat` file.

In some cases, we need to open additional files. In that case, we might see programming like this:

```
>>> from pathlib import Path
>>> target_path = Path("data")/"extra_detail.log"
>>> with target_path.open('w') as target_file:
...     print("Some detailed output", file=target_file)
```

```
...     print("Ordinary log")
Ordinary log
```

In this example, we've opened a specific path for the output and assigned the open file to `target_file`, using the `with` statement. We can then use this as the `file=` value in a `print()` function to write to this file. Because a file is a context manager, leaving the `with` statement means that the file will be closed properly; all of the OS resources will be released from the application. All file operations should be wrapped in a `with` statement context to ensure that the resources are properly released.

## See also

- For more formatting options, see the *Debugging with f"{value=}" strings* recipe.
- For more information on the input data in this example, refer to the *Shrinking sets – remove(), pop(), and difference* and *Slicing and dicing a list* recipes in *Chapter 4*.
- For more information on file operations in general, refer to *Chapter 8*

## Using input() and getpass() for user input

Some Python scripts depend on gathering input from a user. There are several ways to do this. One popular technique is to use the console to prompt a user for input interactively.

There are two relatively common situations:

- **Ordinary input:** This will provide a helpful echo of the characters being entered.
- **Secure, no echo input:** This is often used for passwords. The characters entered aren't displayed, providing a degree of privacy. We use the `getpass()` function in the `getpass` module for this.

As an alternative to interactive input, we'll look at some other approaches in the *Using argparse to get command-line input* recipe later in this chapter.

The `input()` and `getpass()` functions are just two implementation choices to read from the

console. It turns out that getting the string of characters is only the first step in gathering useful data. The input also needs to be validated.

## Getting ready

We'll look at a technique to read a complex structure from a person. In this case, we'll use year, month, and day as separate items. These items are then combined to create a complete date.

Here's a quick example of user input that omits all of the validation considerations. This is poor design:

```
from datetime import date

def get_date1() -> date:
    year = int(input("year: "))
    month = int(input("month [1-12]: "))
    day = int(input("day [1-31]: "))
    result = date(year, month, day)
    return result
```

While it is very easy to use the `input()` function, it lacks a number of helpful features. When the user enters an invalid date, this will raise a potentially confusing exception.

We often need to wrap the `input()` function with data validation processing to make it more useful. The calendar is complex, and we'd hate to accept February 31 without warning the user that it is not a proper date.

## How to do it...

1. If the input is a password or something equally subject to redaction, the `input()` function isn't the best choice. If passwords or other secrets are involved, then use the `getpass.getpass()` function. This means we need the following import when secrets are involved:

```
from getpass import getpass
```

Otherwise, when secret input is not required, we'll use the built-in `input()` function, and no additional import is required.

2. Determine which prompt will be used. In our example, we provided a field name and a hint about the type of data expected as the prompt string argument to the `input()` or `getpass()` functions. It can help to separate the input from the text-to-integer conversion. This recipe doesn't follow the snippet shown previously; it breaks the operation into two separate steps. First, get the text value:

```
year_text = input("year: ")
```

3. Determine how to validate each item in isolation. The simplest case is a single value with a single rule that covers everything. In more complex cases – like this one – each individual element is a number with a range constraint. In a later step, we'll look at validating the composite item:

```
year = int(year_text)
```

Wrap the input and validation into a `while-try` block that looks like this:

```
year = None
while year is None:
    year_text = input("year: ")
    try:
        year = int(year_text)
    except ValueError as ex:
        print(ex)
```

This applies a single validation rule, the `int(year_text)` expression, to ensure that the input is an integer. The `while` statement leads to a repeat of the input and conversion sequence of steps until the value of the `year` variable is not `None`.

Raising an exception for faulty input allows us some flexibility. We can extend this with additional exception classes for other conditions the input must meet.

This processing only covers the year field. We still need to get values for the month and



day fields. This means we'll need three nearly identical loops for each of these three fields of a complex date object. Rather than copying and pasting nearly identical code, we need to restructure this processing.

We'll define a new function, `get_integer()`, for general-purpose input of a numeric value. Here's the complete function definition:

```
def get_integer(prompt: str) -> int:
    while True:
        value_text = input(prompt)
        try:
            value = int(value_text)
            return value
        except ValueError as ex:
            print(ex)
```

We can combine this into an overall process to get the three integers of a date. This will involve a similar *while-try* design pattern but applied to the composite object. It will look like this:

```
def get_date2() -> date:
    while True:
        year = get_integer("year: ")
        month = get_integer("month [1-12]: ")
        day = get_integer("day [1-31]: ")
        try:
            result = date(year, month, day)
            return result
        except ValueError as ex:
            print(f"invalid, {ex}")
```

This uses individual *while-try* processing sequences around the sequence of `get_integer()` functions to get the individual values that make up a date. Then, it uses the `date()` constructor to create a date object from the individual fields. If the date object – as a whole – can't be built because the pieces are invalid, then the year, month, and day must be reentered to create a valid date.

## How it works...

We need to decompose the input problem into several separate but closely related problems. To do this, imagine a tower of conversion steps. At the bottom layer is the initial interaction with the user. We identified two of the common ways to handle this:

- `input()`: This prompts and reads from a user
- `getpass.getpass()`: This prompts and reads input (like passwords) without a visible echo

These two functions provide the essential console interaction. There are other libraries that can provide more sophisticated interactions, if that's required. For example, the Click project has some helpful prompting capabilities. See <https://click.palletsprojects.com/en/7.x/>.

The Rich project has extremely sophisticated terminal interaction. See <https://rich.readthedocs.io/en/latest/>.

On top of the foundation, we've built several tiers of validation processing. The tiers are as follows:

- A **data type** validation: This uses built-in conversion functions such as `int()` or `float()`. These raise a `ValueError` exception for invalid text.
- A **domain** validation: This uses an `if` statement to determine whether values fit any application-specific constraints. For consistency, this should also raise a `ValueError` exception if the data is invalid.
- **Composite object** validation: This is application-specific checking. For our example, the composite object was an instance of `datetime.date`. This also tends to raise `ValueError` exceptions for dates that are invalid.

There are a lot of potential kinds of constraints that might be imposed on values. We've used the valid date constraint because it's particularly complicated.

## There's more...

We have several alternatives for user input that involve slightly different approaches. We'll look at these two topics in detail:

- **Complex text:** This will involve the simple use of `input()` with more sophisticated parsing of the source text. Instead of prompting for individual fields, it might be better to accept a string in the `yyyy-mm-dd` format and use the `strptime()` parser to extract a date. This doesn't change the design pattern; it replaces an `int()` or `float()` with something a bit more complicated.
- **Interaction via the `cmd` module:** This involves a more complex class to control interaction. We'll look at this closely in the *Using `cmd` to create command-line applications* recipe.

A list of potential input validation rules can be extracted from JSON schema definitions. This list of types includes Boolean, integer, float, and string. A number of common string formats defined in JSON schema include date-time, time, date, email, hostname, IP addresses in the version 4 and version 6 formats, and URIs.

Another source of user input validation rules can be found in the definition of the HTML5 `<input>` tag. This list includes color, date, datetime-local, email, file, month, number, password, telephone numbers, time, URL, and week-year.

## See also

- See the *Using `cmd` to create command-line applications* recipe in this chapter for complex interaction.
- See the *Using `argparse` to get command-line input* recipe to gather user input from the command line.
- In the reference material for the SunOS operating system, which is now owned by Oracle, there is a collection of commands that prompt for different kinds of user inputs: <https://docs.oracle.com/cd/E19683-01/816-0210/6m6nb7m5d/index.html>

## Debugging with f“{value=}” strings

One of the most important debugging and design tools available in Python is the `print()` function. The two formatting options shown in the *Using the features of the `print()` function* recipe don't offer a lot of flexibility. We have more flexibility with f"string" formatting. We'll build on some of the recipes shown in *Chapter 1, Numbers, Strings, and Tuples*.

### Getting ready

Let's look at a multi-step process that involves some moderately complex calculations. We'll compute the mean and standard deviation of some sample data. Given these values, we'll locate all items that are more than one standard deviation above the mean:

```
>>> import statistics
>>> size = [2353, 2889, 2195, 3094,
...        725, 1099, 690, 1207, 926,
...        758, 615, 521, 1320]
>>> mean_size = statistics.mean(size)
>>> std_size = statistics.stdev(size)
>>> sig1 = round(mean_size + std_size, 1)
>>> [x for x in size if x > sig1]
[2353, 2889, 3094]
```

This calculation has several working variables. The final list comprehension involves three other variables, `mean_size`, `std_size`, and `sig1`. With so many values used to filter the `size` list, it's difficult to visualize what's going on. It's often helpful to know the steps in the calculation; showing the values of the intermediate variables can be very helpful.

### How to do it...

The f“{name=}” string will have both the literal string `name=` and the value for the `name` expression. This is often a variable, but any expression can be used. Using this with a `print()` function looks as follows:

```
>>> print(
...     f"{mean_size:.2f}, {std_size:.2f}"
... )
mean_size=1414.77, std_size=901.10
```

We can use `{name=}` to put any variable into the f-string and see the value. These examples in the code above include a suffix of `:.2f` as the format specification to show the values rounded to two decimal places. Another common suffix is `!r` to show the internal representation of the object; we might use `f"{name=!r}"`.

## How it works...

For more background on the formatting options, refer to the *Building complicated strings with f-strings* recipe in *Chapter 1*.

There is a very handy extension to this capability. We can use any expression on the left of the `=` in the f-string. This will show the expression and the value computed by the expression, providing us with even more debugging information.

## There's more...

We can use the extended expression capability of f-strings to include additional calculations that aren't simply the values' local variables:

```
>>> print(
...     f"{mean_size:.2f}, {std_size:.2f},"
...     f" {mean_size + 2 * std_size:.2f}"
... )
mean_size=1414.77, std_size=901.10, mean_size + 2 * std_size=3216.97
```

We've computed a new value, `mean_size+2*std_size`, that appears only inside the formatted output. This lets us display intermediate computed results without having to create an extra variable.

## See also

- Refer to the *Building complicated strings with f-strings* recipe in *Chapter 1*, for more of the things that can be done with f-strings and the `format()` method of a string.
- Refer to the *Using the features of the `print()` function* recipe earlier in this chapter for other formatting options.

## Using argparse to get command-line input

For some applications, it can be better to get the user input from the OS command line without a lot of human interaction. We'd prefer to parse the command-line argument values and either perform the processing or report an error.

For example, at the OS level, we might want to run a program like this:

```
% python ch06/distance_app.py -u KM 36.12,-86.67 33.94,-118.40
From 36.12,-86.67 to 33.94,-118.4 in KM = 2886.90
```

At the OS prompt of `%`, we entered a command, `python ch06/distance_app.py`. This command had an optional argument, `-u KM`, and two positional arguments of `36.12, -86.67` and `33.94, -118.40`.

If the user enters something incorrect, the interaction might look like this:

```
% python ch06/distance_app.py -u KM 36.12,-86.67 33.94,-118asd
usage: distance_app.py [-h] [-u {NM,MI,KM}] p1 p2
distance_app.py: error: argument p2: could not convert string to float:
'-118asd'
```

An invalid argument value of `-118asd` leads to an error message. A user can hit the up-arrow key to get the previous command line back, make a change, and run the program again. The interactive user experience is delegated to OS command-line processing.

## Getting ready

The first thing we need to do is to refactor our code to create three separate functions:

- A function to get the arguments from the command line.
- A function that does the real work. The intent is to define a function that can be reused in a variety of contexts, one of which is with parameters from the command line.
- A main function that gathers arguments and invokes the *real work* function with the appropriate argument values.

Here's our *real work* function, `display()`:

```
from ch06.distance_computation import haversine, MI, NM, KM

def display(
    lat1: float, lon1: float, lat2: float, lon2: float, r: str
) -> None:
    r_float = {"NM": NM, "KM": KM, "MI": MI}[r]
    d = haversine(lat1, lon1, lat2, lon2, R=r_float)
    print(f"From {lat1},{lon1} to {lat2},{lon2} in {r} = {d:.2f}")
```

We've imported the core calculation, `haversine()`, from the `ch06.distance_computation` module. We've based this on the calculations shown in the examples in the *Picking an order for parameters based on partial functions* recipe in *Chapter 3*:

Here's how the function looks when it's used inside Python:

```
>>> display(36.12, -86.67, 33.94, -118.4, 'NM')
From 36.12,-86.67 to 33.94,-118.4 in NM = 1558.53
```

This function has two important design features. The first feature is that it avoids references to features of the `argparse.Namespace` object that's created by argument parsing. Our goal is to have a function that we can reuse in a number of alternative contexts. We need to keep the input and output elements of the user interface separate.

The second design feature is that this function displays a value computed by another function. This is a decomposition of a larger problem into two smaller problems. We've

separated the user experience of printed output from the essential calculation. (Both aspects are quite small, but the principle of separating these two aspects is important.)

## How to do it...

1. Define the overall argument parsing function:

```
def get_options(argv: list[str]) -> argparse.Namespace:
```

2. Create the parser object:

```
parser = argparse.ArgumentParser()
```

3. Add the various types of arguments to the parser object. Sometimes, this is difficult because we're still refining the user experience. It's difficult to imagine all the ways in which people will use a program and all of the questions they might have. For our example, we have two mandatory, positional arguments and an optional argument:

- Point 1 latitude and longitude
- Point 2 latitude and longitude
- Optional units of distance; we'll provide nautical miles as the default:

```
parser.add_argument("-u", "--units",  
                    action="store", choices=("NM", "MI", "KM"), default="NM")  
parser.add_argument("p1", action="store", type=point_type)  
parser.add_argument("p2", action="store", type=point_type)
```

We've added a mix of optional and mandatory arguments. The `-u` argument starts with a dash to mark it as optional. A longer double dash version, `--units`, is supported as an alternative.

The mandatory, positional arguments are named without a prefix.

4. Evaluate the `parse_args()` method of the parser object created in *step 2*:



```
options = parser.parse_args(argv)
```

By default, the parser uses the values from `sys.argv`, which are the command-line argument values entered by the user. Testing is much easier when we can provide an explicit argument value.

Here's the final function:

```
def get_options(argv: list[str]) -> argparse.Namespace:
    parser = argparse.ArgumentParser()
    parser.add_argument("-u", "--units",
        action="store", choices=("NM", "MI", "KM"), default="NM")
    parser.add_argument("p1", action="store", type=point_type)
    parser.add_argument("p2", action="store", type=point_type)
    options = parser.parse_args(argv)
    return options
```

This relies on a `point_type()` function to both validate the strings and convert the string to a (latitude, longitude) two-tuple. Here's the definition of this function:

```
def point_type(text: str) -> tuple[float, float]:
    try:
        lat_str, lon_str = text.split(",")
        lat = float(lat_str)
        lon = float(lon_str)
        return lat, lon
    except ValueError as ex:
        raise argparse.ArgumentTypeError(ex)
```

If anything goes wrong, an exception will be raised. From this exception, we'll raise an `ArgumentTypeError` exception. This is caught by the `argparse` module and causes it to report the error to the user.

Here's the main script that combines the option parser and the output display functions:

```
def main(argv: list[str] = sys.argv[1:]) -> None:
    options = get_options(argv)
    lat_1, lon_1 = options.p1
    lat_2, lon_2 = options.p2
    display(lat_1, lon_1, lat_2, lon_2, r=options.r)

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

This main script connects the user inputs to the displayed output. The details of error messages and help processing are delegated to the `argparse` module.

## How it works...

The argument parser works in three stages:

1. Define the overall context by creating a parser object as an instance of the `ArgumentParser` class.
2. Add individual arguments with the `add_argument()` method. These can include optional arguments as well as required arguments.
3. Parse the actual command-line inputs, often based on `sys.argv`.

Some simple programs will have a few optional arguments. A more complex program may have many optional arguments.

It's common to have a filename as a positional argument. When a program reads one or more files, the filenames can be provided in the command line, as follows:

```
% python some_program.py *.rst
```

We've used the Linux shell's **globbing** feature: the `*.rst` string is expanded into a list of all files that match the naming rule. This is a feature of the Linux shell, and it happens before the Python interpreter starts. This list of files can be processed using an argument, defined as follows:

```
parser.add_argument('file', type=Path, nargs='*')
```

All of the arguments on the command line that do *not* start with the - character are positional arguments, and they are collected into the `file` value in the object built by the parser.

We can then use the following to process each of the given files:

```
for filename in options.file:
    process(filename)
```

For Windows programs, the shell doesn't glob filenames from wildcard patterns. This means the application must deal with filenames that contain wildcard characters like "\*" and "?" in them. The Python `glob` module can help with this. Also, the `pathlib` module can create `Path` objects, which include globbing features to locate matching filenames in a directory.

## There's more...

What kinds of arguments can we process? There are a lot of argument styles in common use. All of these variations are defined using the `add_argument()` method of a parser:

- **Simple options:** Arguments of the form `-o` or `--option` often define optional features. These use the `'store_true'` or `'store_false'` actions.
- **Options with values:** We showed `-r unit` as an option with a value. The `'store'` action is how the value is saved.
- **Options that increment a counter:** The action `'count'` and `default=0` permit an option that can be repeated. The verbose and very verbose logging options, `-v` and `-vv`, respectively, are an example.
- **Options that accumulate a list:** The action `'append'` and `default=[]` can accumulate multiple option values.

- **Show the version number:** A special action of 'version' can be used to create an argument that will show the version number and exit.
- Positional arguments do not have a leading '-' in their name. They must be defined in the order they will be used.

The `argparse` module uses `-h`, and `--help` will display a help message and exit. These options are available unless changed by an argument that uses the 'help' action.

This covers most of the common cases for command-line argument processing. Generally, we'll try to leverage these common styles of arguments when we write our own applications. If we strive to follow the widely used argument styles, our users are more likely to understand how our application works.

## See also

- We looked at how to get interactive user input in the *Using `input()` and `getpass()` for user input* recipe.
- We'll look at a way to add even more flexibility to this in the *Using the OS environment settings* recipe.

## Using `invoke` to get command-line input

The `invoke` package is not part of the standard library. It needs to be installed separately. Generally, this is done with the following terminal command:

```
(cookbook3) % python -m pip install invoke
```

Using the `python -m pip` command ensures that we will use the `pip` command that goes with the currently active virtual environment, shown as `cookbook3`.

See the *Using `argparse` to get command-line input* recipe in this chapter. It describes a command-line application that works something like the following:

```
% RECIPE=7 # invoke
```

The command will always be `invoke`. The Python path information is used to locate a module file named `tasks.py` to provide the definitions of the commands that can be invoked. The remaining command-line values are provided to a function defined in the `tasks` module.

## Getting ready

Often, we'll create a two-tiered design when using **invoke**. These two tiers are:

- A function to get the arguments from the command line, do any validation or conversion required, and invoke the function to perform the real work. This function will be decorated with `@task`.
- A function that does the real work. It helps if this function is designed so that it makes no reference to the command-line options in any direct way. The intent is to define a function to be reused in a variety of contexts, one of which is with parameters from the command line.

In some cases, these two functions can be merged into one. This often happens when Python is used as a wrapper to provide a simple interface for an underlying application that is bewilderingly complicated. In this kind of application, the Python wrapper may do very little processing, with no useful distinction between validating parameter values and doing the “real work” of the application.

In the *Using `argparse` to get command-line input* recipe in this chapter, the `display()` function is defined. This function does the “real work” of the application. When working with **invoke**, this design will continue to be used.

## How to do it...

1. Define a function that describes a task that can be invoked. It's often essential to provide some help for the various parameters, which is done by providing a dictionary

of parameter names and help text to the `@task` decorator:

```
import sys
from invoke.tasks import task
from invoke.context import Context

@task(
    help={
        'p1': 'Lat,Lon',
        'p2': 'Lat,Lon',
        'u': 'Unit: KM, MI, NM'})
def distance(
    context: Context, p1: str, p2: str, u: str = "KM"
) -> None:
    """Compute distance between two points.
    """
```

The docstring for the function becomes the help text provided by the `invoke distance --help` command. It's very important to provide something that will help the user understand what the various commands will do and how to use them.

The `Context` parameter is required, but it won't be used in this example. This object provides a consistent context when invoking multiple separate tasks. It also provides methods to run external applications.

2. Perform any needed conversions on the various parameter values. Evaluate the “real-work” function with cleaned values:

```
try:
    lat_1, lon_1 = point_type(p1)
    lat_2, lon_2 = point_type(p2)
    display(lat_1, lon_1, lat_2, lon_2, r=u)
except (ValueError, KeyError) as ex:
    sys.exit(f"{ex}\nFor help use invoke --help distance")
```

We've used `sys.exit()` to produce a failure message. It's also possible to raise an exception, but this will show long traceback displays that may not be helpful.

## How it works...

The **invoke** package examines the parameters for a given Python function and builds the necessary command-line parsing options. The parameter names become the names for options. In the example `distance()` function, the parameters of `p1`, `p2`, and `u` become the command-line options of `--p1`, `--p2`, and `-u`, respectively. This lets us provide the parameters flexibly when running the application. The values can be provided positionally, or by using the option flags.

## There's more...

The most important feature of **invoke** is its ability to act as a wrapper for other binary applications. The `Context` object that's provided to each task provides ways to change the current working directory and run an arbitrary OS command. This includes options to update the environment of the subprocess, capture the output and error streams, provide an input stream, and many other features.

We can use **invoke** to combine multiple applications under a single wrapper. This can help to simplify complicated collections of applications by providing a uniform interface via a single module of task definitions.

We can, for example, combine an application that computes distance between two points, and a separate application that processes a CSV file with an entire route of a connected series of points.

The overall design might look like this:

```
@task
def distance(context: Context, p1: str, p2: str, u: str) -> None:
    ... # Shown earlier

@task
def route(context: Context, filename: str) -> None:
    if not path(filename).exists():
        sys.exit(f"File not found {filename}")
    context.run("python some_app.py {filename}", env={"APP_UNITS": "NM"})
```

The `context.run()` method will invoke an arbitrary OS-level command. The `env` parameter value provides updated environment variables to the command that's executed.

## See also

- Additional recipes for application integration are shown in *Chapter 14, Application Integration: Combination*.
- The <https://www.pyinvoke.org> web page contains all of the documentation on `invoke`.

## Using `cmd` to create command-line applications

There are several ways to create interactive applications. The *Using `input()` and `getpass()` for user input* recipe looked at functions such as `input()` and `getpass.getpass()`. The *Using `argparse` to get command-line input* recipe showed us how to use the `argparse` module to create applications with which a user can interact from the OS command line.

We have another way to create interactive applications: using the `cmd` module. This module will prompt the user for input and then invoke a specific method of the class we provide.

Here's an example of the interaction:

```
] dice 5
Rolling 5 dice
] roll
[5, 6, 6, 1, 5]
]
```

We entered the `dice 5` command to set the number of dice. After that, the `roll` command showed the results of rolling five dice. The `help` command will show the available commands.

## Getting ready

The core feature of a `cmd.Cmd` application is a **read-evaluate-print loop (REPL)**. This kind of application works well when there are a number of individual state changes and a



number of closely related commands to make those state changes.

We'll make use of a simple, stateful dice game. The idea is to have a handful of dice, some of which can be rolled and some of which are frozen. This means our `Cmd` class definition must have some attributes that describe the current state of a handful of dice.

Commands will include the following:

- **dice** to set the number of dice
- **roll** to roll the dice
- **reroll** to re-roll selected dice, leaving the others untouched

## How to do it...

1. Import the `cmd` module to make the `cmd.Cmd` class definition available. Since this is a game, the `random` module will also be needed:

```
import cmd
import random
```

2. Define an extension to `cmd.Cmd`:

```
class DiceCLI(cmd.Cmd):
```

3. Define any initialization required in the `preloop()` method:

```
def preloop(self) -> None:
    self.n_dice = 6
    self.dice: list[int] | None = None # no roll has been made.
    self.reroll_count = 0
```

This method is evaluated once when the processing starts.

Initialization can also be done in an `__init__()` method. However, doing this is a bit more complicated because it must collaborate with the `Cmd` class initialization.

4. For each command, create a `do_command()` method. The name of the method will be

the command, prefixed by the characters `do_`. Any user input text after the command will be provided as an argument value to the method. The docstring comment in the method definition is the help text for the command. Here is the `roll` command, defined by the `do_roll()` method:

```
def do_roll(self, arg: str) -> bool:
    """Roll the dice. Use the dice command to set the number of
    dice."""
    self.dice = [random.randint(1, 6) for _ in range(self.n_dice)]
    print(f"{self.dice}")
    return False
```

5. Parse and validate the arguments to the commands that use them. The user's input after the command will be provided as the value of the first positional argument to the method. Here is the `dice` command, defined by the `do_dice()` method:

```
def do_dice(self, arg: str) -> bool:
    """Sets the number of dice to roll."""
    try:
        self.n_dice = int(arg)
    except ValueError:
        print(f"{arg!r} is invalid")
        return False
    self.dice = None
    print(f"Rolling {self.n_dice} dice")
    return False
```

6. Write the main script. This will create an instance of this class and execute the `cmdloop()` method:

```
if __name__ == "__main__":
    game = DiceCLI()
    game.cmdloop()
```

The `cmdloop()` method handles the details of prompting, collecting input, and executing the proper method.

## How it works...

The `Cmd` class contains a large number of built-in features to display a prompt, read input from a user, and then locate the proper method based on the user's input.

For example, when we enter a command like `dice 5`, the built-in methods of the `Cmd` superclass will strip the first word from the input, `dice`, and prefix this with `do_`. It will then try to evaluate the method with the argument value of the rest of the line, `5`.

If we enter a command for which there's no matching `do_*` method, the command processor writes an error message. This is done automatically; we don't need to write any code to handle invalid command input.

Some methods, such as `do_help()`, are already part of the application. These methods will summarize the other `do_*` methods. When one of our methods has a docstring, this will be displayed by the built-in help feature.

The `Cmd` class relies on Python's facilities for introspection. An instance of the class can examine the method names to locate all of the methods that start with `do_`. Introspection is an advanced topic, one that will be touched on in *Chapter 8*.

## There's more...

The `Cmd` class has a number of additional places where we can add interactive features:

- We can define specific `help_*` methods that become part of the help topics.
- When any of the `do_*` methods return a non-`False` value, the loop will end. We might want to add a `do_quit()` method to return `True`.
- If the input stream is closed, an EOF command will be provided. In Linux, using `ctrl-d` will close the input file. This leads to the `do_EOF()` method, which should use `return True`.
- We might provide a method named `emptyline()` to respond to blank lines.
- The `default()` method is evaluated when the user's input does not match any of the `do_*` methods.

- The `postloop()` method can be used to do some processing just after the loop finishes. This would be a good place to write a summary.

Also, there are a number of attributes we can set. These are class-level variables that would be peers of the method definitions:

- The `prompt` attribute is the prompt string to write. The `intro` attribute is the introductory text to write before the first prompt. For our example, we can do the following:

```
class DiceCLI2(cmd.Cmd):
    prompt = "] "
    intro = "A dice rolling tool. ? for help."
```

- We can tailor the help output by setting `doc_header`, `undoc_header`, `misc_header`, and `ruler` attributes.

The goal is to be able to create a tidy class that handles user interaction as directly as possible.

## See also

- We'll look at class definitions in *Chapter 7* and *Chapter 8*.

## Using the OS environment settings

There are several ways to look at inputs provided by the users of our software:

- **Interactive input:** This is provided by the user as required by the application. See the *Using `input()` and `getpass()` for user input* recipe.
- **Command-line arguments:** These are provided once, when the program is started. See the *Using `argparse` to get command-line input* and *Using `invoke` to get command-line input* recipes.
- **Environment variables:** These are OS-level settings. There are several ways these can be set:

- At the command line, when running the application.
- Set in a configuration file for the user’s selected shell. For example, if using **zsh**, these files are the `~/.zshrc` file and the `~/.profile` file. There can also be system-wide files, like the `/etc/zshrc` file.
- In Windows, there’s the Advanced Settings option for environment variables.
- **Configuration files:** These are unique to an application. They are the subject of *Chapter 13*.

The environment variables are available through the `os` module.

## Getting ready

In the *Using `argparse` to get command-line input* recipe, we wrapped the `haversine()` function in a simple application that parsed command-line arguments. We created a program that worked like this:

```
% python ch06/distance_app.py -u KM 36.12,-86.67 33.94,-118.40
From 36.12,-86.67 to 33.94,-118.4 in KM = 2886.90
"""
```

After using this version of the application for a while, we may find that we’re often using nautical miles to compute distances from where our boat is anchored. We’d really like to have default values for one of the input points as well as the `-r` argument.

Since a boat can be anchored in a variety of places, we need to change the default without having to tweak the actual code. The idea of a “slowly changing” argument value fits well with the OS environment variables. They can be persistent but are also relatively easy to change.

We’ll use two OS environment variables:

- `UNITS` will have the default distance units.
- `HOME_PORT` can have an anchor point.

We want to be able to do the following:

```
% UNITS=NM
% HOME_PORT=36.842952,-76.300171
% python ch06/distance_app.py 36.12,-86.67

From 36.12,-86.67 to 36.842952,-76.300171 in NM = 502.23
```

## How to do it...

1. Import the `os` module. The default set of command-line arguments to parse comes from `sys.argv`, so it's important to also import the `sys` module. The application will also depend on the `argparse` module:

```
import os
import sys
import argparse
```

2. Import any other classes or objects needed for the application:

```
from ch03.recipe_11 import haversine, MI, NM, KM
from ch06.recipe_04 import point_type, display
```

3. Define a function that will use the environment values as defaults for optional command-line arguments:

```
def get_options(argv: list[str] = sys.argv[1:]) -> argparse.Namespace:
```

4. Gather default values from the OS environment settings. This includes any validation required:

```
default_units = os.environ.get("UNITS", "KM")
if default_units not in ("KM", "NM", "MI"):
    sys.exit(f"Invalid UNITS, {default_units!r} not KM, NM, or
            MI")
default_home_port = os.environ.get("HOME_PORT")
```

Note that using `os.environ.get()` permits the application to include a default value for cases where the environment variable is not set.

5. Create the parser object. Provide the default values for the relevant arguments extracted from the environment variables:

```
parser = argparse.ArgumentParser()
parser.add_argument("-u", "--units",
                    action="store", choices=("NM", "MI", "KM"),
                    default=default_units
)
parser.add_argument("p1", action="store", type=point_type)
parser.add_argument(
    "p2", nargs="?", action="store", type=point_type,
    default=default_home_port
)
```

6. Do any additional validation to ensure that arguments are set properly. In this example, it's possible to have no value for `HOME_PORT` and no value provided for the second command-line argument.

This requires an if statement and a call to `sys.exit()`:

```
options = parser.parse_args(argv)
if options.p2 is None:
    sys.exit("Neither HOME_PORT nor p2 argument provided.")
```

7. Return the final options object with the set of valid arguments:

```
return options
```

This will allow the `-u` argument and the second point to be optional. The argument parser will use the configuration information to supply default values if these are omitted from the command line.

There is a nuanced distinction in the error code provided by `sys.exit()`. When applications fail because of command-line problems, it's common to return a status code of 2, but `sys.exit()` will set the value to 1. A slightly better approach is to use the `parser.error()`

method. Doing this requires refactoring to create the `ArgumentParser` instance *before* acquiring and validating values from environment variables.

## How it works...

We've used the OS environment variables to create default values that can be overridden by command-line arguments. If the environment variable is set, that string is provided as the default to the argument definition. If the environment variable is not set, then an application-level default value will be used. In the case of the `UNITS` variable, in this example, the application uses kilometers as the default if the OS environment variable is not set.

We've used the OS environment to set default values that can be overridden by the command-line argument values. This supports the idea of the environment providing a general context that might be shared by a number of commands.

## There's more...

The *Using `argparse` to get command-line input* recipe shows a slightly different way to handle the default command-line arguments available from `sys.argv`. The first of the arguments is the name of the Python application being executed and is not often relevant to argument parsing.

The value of `sys.argv` will be a list of strings:

```
['ch06/distance_app.py', '-u', 'NM', '36.12,-86.67']
```

We have to skip the initial value in `sys.argv[0]` at some point in the processing. Generally, the application needs to provide `sys.argv[1:]` to the parser. This can be done inside the `get_options()` function. It can be done when the `main()` function evaluates the `get_options()` function. As shown in this example, it can be done when creating the default argument values for the `get_options()` function.

The `argparse` module allows us to provide type information for an argument definition. Providing type information can be used to validate argument values. In many cases, there



may be a finite list of choices for a value, and this set of allowed choices can be provided as part of the argument definition. Doing this creates better error and help messages, improving the user experience when running the application.

## See also

- We'll look at numerous ways to handle configuration files in *Chapter 13*.

## Join our community Discord space

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

<https://packt.link/dHrHU>



# 7

## Basics of Classes and Objects

The point of computing is to process data. We often encapsulate the processing and the data into a single definition. We can organize objects into classes with a common collection of attributes to define their internal state and common behavior. Each instance of a class is a distinct object with unique internal state and behavior.

This concept of state and behavior applies particularly well to the way games work. When building something like an interactive game, the user's actions update the game state. Each of the player's possible actions is a method to change the state of the game. In many games this leads to a lot of animation to show the transition from state to state. In a single-player arcade-style game, the enemies or opponents will often be separate objects, each with an internal state that changes based on other enemy actions and the player's actions.

On the other hand, if we consider a card or dice game, there may be very few states possible. A game like *Zonk* involves a player rolling (and rerolling) dice as long as their

score improves. If a subsequent roll fails to improve their hand of dice, their turn is over. The hand's state is the pool of dice that comprise the scoring subset, often pushed to one side of the table. In a six-dice game, there will be from one to six scoring dice as distinct states. Additionally, when all dice are scoring, the player can begin the rolling process again by rerolling all of the dice. This leads to an additional "over-the-top" state that the players must also bear in mind.

The point of object-oriented design is to define the current state with the attributes of an object. Each object is defined as an instance of a class of similar objects. We write the class definitions in Python and use these to create objects. The methods defined in the class cause the state changes on an object.

In this chapter, we will look at the following recipes:

- *Using a class to encapsulate data and processing*
- *Essential type hints for class definitions*
- *Designing classes with lots of processing*
- *Using typing.NamedTuple for immutable objects*
- *Using dataclasses for mutable objects*
- *Using frozen dataclasses for immutable objects*
- *Optimizing small objects with `__slots__`*
- *Using more sophisticated collections*
- *Extending a built-in collection – a list that does statistics*
- *Using properties for lazy attributes*
- *Creating contexts and context managers*
- *Managing multiple contexts with multiple resources*

The subject of object-oriented design is quite large. In this chapter, we'll cover some of the

essentials. We'll start with some foundational concepts, such as how a class definition encapsulates state and processing details for all instances of a class.

## Using a class to encapsulate data and processing

Class design is influenced by the SOLID design principles. The Single Responsibility and Interface Segregation principles offer helpful advice. Taken together, these principles advise us that a class should have methods narrowly focused on a single, well-defined responsibility.

Another way of considering a class is as a group of closely-related functions working with common data. We call these methods for working with the data. A class definition should contain the smallest collection of methods for working with the object's data.

We'd like to create class definitions based on a narrow allocation of responsibilities. How can we define responsibilities effectively? What's a good way to design a class?

### Getting ready

Let's look at a simple, stateful object – a pair of dice. The context for this is an application that simulates a simple game like *Craps*.

A software object can be viewed as analogous to a thing – a noun. The behaviors of the class can then be viewed as verbs. This identification with nouns and verbs gives us a hint as to how we can proceed to design classes to work effectively.

This leads us to several steps of preparation. We'll provide concrete examples of these steps using a pair of dice for game simulation. We proceed as follows:

1. Write down simple sentences that describe what an instance of the class does. We can call these the problem statements. It's essential to focus on single-verb sentences, with a focus on only the nouns and verbs. Here are some examples:
  - The game of *Craps* has two standard dice.
  - Each die has six faces, with point values from one to six.

- Dice are rolled by a player. While writers and editors prefer the active voice version, “A player rolls the dice,” the dice are often acted upon by other objects, making passive-voice sentences slightly more useful.
  - The total of the dice changes the state of the *Craps* game. Those rules are separate from the dice.
  - If the two dice match, the number was described as being rolled “the hard way”. If the two dice do not match, the roll was described as being made “the easy way”.
2. Identify all of the nouns in the sentences. In this example, the nouns include dice, faces, point values, and a player. The nouns identify different classes of objects that may be **collaborators**, like player and game. Nouns may also identify attributes of objects, like face and point value.
  3. Identify all the verbs in the sentences. Verbs often become methods of the class in question. In this example, verbs include roll and match.

This information helps define the state and behavior of the objects. Having this background information will help us write the class definition.

## How to do it...

Since the simulation we’re writing involves random throws of dice, we’ll depend on from random `import randint` to provide the useful `randint()` function. The steps for defining a class are as follows:

1. Start writing the class with the `class` statement:

```
class Dice:
```

2. Initialize the object’s attributes within the body of an `__init__()` method. We’ll model the internal state of the dice with a `faces` attribute. A `self` variable is required to be sure that we’re referencing an attribute of a given instance of a class. We’ll

provide a type hint on each attribute to be sure it's used properly throughout the class definition:

```
def __init__(self) -> None:
    self.faces: tuple[int, int] = (0, 0)
```

3. Define the object's methods based on the verbs in the description. When the player rolls the dice, a `roll()` method can set the values shown on the faces of the two dice. We implement this with a method to set the `faces` attribute of the `self` object:

```
def roll(self) -> None:
    self.faces = (randint(1,6), randint(1,6))
```

This method mutates the internal state of the object. We've elected to not return a value.

4. After a player rolls the dice, a `total()` method helps compute the total of the dice:

```
def total(self) -> int:
    return sum(self.faces)
```

5. Additional methods can provide answers to questions about the state of the dice. In this case, the total was made “the hard way” when both dice match:

```
def hardway(self) -> bool:
    return self.faces[0] == self.faces[1]
def easyway(self) -> bool:
    return self.faces[0] != self.faces[1]
```

## How it works...

The core idea here is to use ordinary rules of grammar – nouns, verbs, and adjectives – as a way to identify basic features of a class. In our example, dice are real things. We try to avoid using abstract terms such as randomizers or event generators. It's easier to describe the tangible features of real things, and then define an implementation to match the tangible features.

The idea of rolling the dice is an example physical action that we can model with a method definition. This action of rolling the dice changes the state of the object. In rare cases – 1 time in 36 – the next state will happen to match the previous state.

Here's an example of using the Dice class:

1. First, we'll seed the random number generator with a fixed value so that we can get a fixed sequence of results:

```
>>> import random
>>> random.seed(1)
```

2. We'll create a Dice object, and assign it to a variable, d1. We can then set its state with the roll() method. We'll then look at the total() method to see what was rolled. We'll examine the state by looking at the faces attribute:

```
>>> d1 = Dice()
>>> d1.roll()
>>> d1.total()
7
>>> d1.faces
(2, 5)
```

## There's more...

Capturing the essential internal state and methods that cause state change is the first step in good class design. We can summarize some helpful design principles using the acronym **SOLID**:

- **Single Responsibility Principle:** A class should have one clearly defined responsibility.
- **Open/Closed Principle:** A class should be open to extension – generally via inheritance – but closed to modification. We should design our classes so that we don't need to tweak the code to add or change features.
- **Liskov Substitution Principle:** We need to design inheritance so that a subclass

can be used in place of the superclass.

- **Interface Segregation Principle:** When writing a problem statement, we want to be sure that collaborating classes have as few dependencies as possible. In many cases, this principle will lead us to decompose large problems into many small class definitions.
- **Dependency Inversion Principle:** It's less than ideal for a class to depend directly on other classes. It's better if a class depends on an abstraction, and a concrete implementation class is substituted for the abstract class.

The goal is to create classes that have the necessary behavior and also adhere to the design principles so they can be extended and reused.

## See also

- See the *Using properties for lazy attributes* recipe, where we'll look at the choice between an eager attribute and a lazy property.
- In *Chapter 8*, we'll look in more depth at class design techniques.
- See *Chapter 15*, for recipes on how to write appropriate unit tests for the class.

## Essential type hints for class definitions

A class name is also a type hint, allowing a direct reference between a variable and the class that should define the objects associated with the variable. This relationship lets tools such as **mypy** reason about our programs to be sure that object references and method references appear to match the type hints in our code.

In addition to the class name, we'll use type hints in three common places within a class definition:

- In method definitions, we'll use type hints to annotate the parameters and the return type.
- In the `__init__()` method, we may need to provide hints for the instance variables



that define the state of the object.

- Inside the attributes of the class overall. These are not common and type hints are rare here.

## Getting ready

We're going to examine a class with a variety of type hints. In this example, our class will model a handful of dice. We'll allow rerolling selected dice, making the instance of the class stateful.

The collection of dice can be set by a first roll, where all the dice are rolled. The class allows subsequent rolls of a subset of dice. The number of rolls is counted as well.

The type hints will reflect the nature of the collection of dice, the integer counts, a floating-point average value, and a string representation of the hand as a whole. This will show a number of type hints and how to write them.

## How to do it...

1. This definition will involve random numbers as well as type hints for sets and lists.

We import the random module:

```
import random
```

2. Define the class. This creates a new type:

```
class Dice:
```

3. It's rare for class-level variables to require a type hint. They're almost always created with assignment statements that make the type information clear to a person or a tool like **mypy**. In this case, we want all instances of our class of dice to share a common random number generator object:

```
RNG = random.Random()
```

- The `__init__()` method creates the instance variables that define the state of the object. In this case, we'll save some configuration details, plus some internal state. The `__init__()` method also has the initialization parameters. Generally, we'll put the type hints on these parameters. Other internal state variables may require type hints to show what kinds of values will be assigned by other methods of the class. In this example, the `faces` attribute has no initial value; we state that when it is set, it will be a `List[int]` object:

```
def __init__(self, n: int, sides: int = 6) -> None:
    self.n_dice = n
    self.sides = sides
    self.faces: list[int]
    self.roll_number = 0
```

- Methods that compute new derived values can be annotated with their return type information. Here are three examples to return a string representation, compute the total, and also compute an average of the dice. These functions have return types of `str`, `int`, and `float`, as shown:

```
def __str__(self) -> str:
    return ", ".join(
        f"{i}: {f}"
        for i, f in enumerate(self.faces)
    )

def total(self) -> int:
    return sum(self.faces)

def average(self) -> float:
    return sum(self.faces) / self.n_dice
```

- For methods with parameters, we include type hints on the parameters as well as a return type. In this case, the methods that change the internal state also return values. The return value from both methods is a list of dice faces, described as `list[int]`. The parameter for the `reroll()` method is a set of dice to be rolled again. This is shown as a `set[int]` requiring a set of integers. Python is a little more flexible than

this, and we'll look at some alternatives:

```
def first_roll(self) -> list[int]:
    self.roll_number = 0
    self.faces = [
        self.RNG.randint(1, self.sides)
        for _ in range(self.n_dice)
    ]
    return self.faces

def reroll(self, positions: set[int]) -> list[int]:
    self.roll_number += 1
    for p in positions:
        self.faces[p] = self.RNG.randint(1, self.sides)
    return self.faces
```

## How it works...

The type hint information is used by programs such as **mypy** to be sure that the instances of the class are used properly throughout the application.

If we try to write a function like the following:

```
def example_mypy_failure() -> None:
    d = Dice(2.5)
    d.first_roll()
    print(d)
```

The attempt to create an instance of the Dice class using a float value for the `n` parameter represents a conflict with the type hints. The hint for the Dice class's `__init__()` method claimed the argument value should be an integer. The **mypy** program reports the following:

```
src/ch07/recipe_02_bad.py:9: error: Argument 1 to "Dice" has incompatible type "float"; expected "int" [arg-type]
```

If we try to execute the application, it will raise a `TypeError` exception in another place. The error will manifest when evaluating the `d.first_roll()` method. The exception is raised here because the body of the `__init__()` method works well with values of any type.

The hints claim specific types are expected, but at runtime, any object can be provided. The hints are not checked during execution.

Similarly, when we use other methods, the **mypy** program checks to be sure our use of the method matches the expectations defined by the type hints. Here's another example:

```
r1: list[str] = d.first_roll()
```

This assignment statement has a type hint for the `r1` variable that doesn't match the type hint for the return type from the `first_roll()` method. This conflict is found by **mypy** and reported as an `Incompatible types in assignment` error.

## There's more...

One of the type hints in this example is too specific. The function for re-rolling the dice, `reroll()`, has a `positions` parameter. The `positions` parameter is used in a `for` statement, which means the object must be some kind of iterable object.

The mistake was providing a type hint, `set[int]`, which is only one of many kinds of iterable objects. We can generalize this definition by switching the type hint from the very specific `set[int]` to the more general `Iterable[int]`.

Relaxing the hint means that any `set`, `list`, or `tuple` object can be a valid argument value for this parameter. The only other code change required is to import `Iterable` from the `collections.abc` module.

The `for` statement has a specific protocol for getting the iterator object from an iterable collection, assigning values to a variable, and executing the indented body. This protocol is defined by the `Iterable` type hint. There are many such protocol-based types, and they allow us to provide type hints that match Python's inherent flexibility with respect to type.

## See also

- In *Chapter 3*, in the *Function parameters and type hints* recipe, a number of similar concepts are shown.

- In *Chapter 4*, the *Writing list-related type hints* and *Writing set-related type hints* recipes address additional detailed type hinting.
- In *Chapter 5*, the *Writing dictionary-related type hints* recipe also addresses type hinting.

## Designing classes with lots of processing

Some of the time, an object will contain all of the data that defines its internal state. There are cases, however, where a class doesn't hold the data, but instead is designed to consolidate processing for data held in separate containers.

Some prime examples of this design are statistical algorithms, which are often outside the data being analyzed. The data might be in a built-in list or Counter object; the processing is defined in a class separate from the data container.

### Getting ready

It's quite common to do analysis on data that's already been summarized into groups or bins. We might, for example, have a vast data file with a large number of measurements of an industrial process.

For background, see the NIST Aerosol Particle Size case study: <https://www.itl.nist.gov/div898/handbook/pmc/section6/pmc62.htm>

Rather than analyze the voluminous raw data, it's often much faster to first summarize the important variables, then analyze the summarized data. The summary data can be kept in a Counter object. The data looks like this:

```
data = Counter({7: 80,  
               6: 67,  
               8: 62,  
               9: 50,
```

```
... Details omitted ...
```

```
2: 3,  
3: 2,  
1: 1})
```

The keys (7, 6, 8, 9, and so on) are codes reflecting the particle size. The actual sizes varied from 109 to 119. Computing the actual size,  $c$ , from the  $s$  code is done with  $c = \lfloor 2(s - 109) \rfloor$ . (The units aren't provided in the NIST background information. Since a lot of the data reflects electronic chip wafers and fabrication, the units are likely something very small.)

We want to compute some statistics on this Counter object without being forced to work with the original voluminous dataset. In general, there are two general design strategies for designing classes to store and process data:

- **Extend** the storage class definition, Counter in this case, to add statistical processing. We'll cover this in detail in the *Extending a built-in collection – a list that does statistics* recipe.
- **Wrap** a Counter object in a class that provides the additional features required. When we do this, though, we have two more choices:
  - Expose the underlying Counter object. We'll focus on this.
  - Write special methods to make the wrapper appear to also be a collection, encapsulating the Counter object. We'll look at this in *Chapter 8*.

For this recipe, we'll focus on the **wrap** variant where we define a statistical computation class that exposes a Counter object. We have two ways to design this compute-intensive processing:

- An **Eager** implementation computes the statistics as soon as possible. The values become simple attributes. We'll focus on this choice.
- A **Lazy** approach doesn't compute anything until the value is required via a method function or property. We'll look at this in the *Using properties for lazy attributes* recipe.

The essential algorithm for both designs is the same. The only question is when the work of the computation gets done.

## How to do it...

1. Import the appropriate class from the collections module. The computation uses `math.sqrt()`. Be sure to add the needed `import math` also:

```
from collections import Counter
import math
```

2. Define the class with a descriptive name:

```
class CounterStatistics:
```

3. Write the `__init__()` method to include the object where the data is located. In this case, the type hint is `Counter[int]` because the keys used in the Counter object will be integers:

```
def __init__(self, raw_counter: Counter[int]) -> None:
    self.raw_counter = raw_counter
```

4. Initialize any other local variables in the `__init__()` method that might be useful. Since we're going to calculate values eagerly, the most eager possible time is when the object is created. We'll write references to some yet to be defined functions:

```
self.mean = self.compute_mean()
self.stddev = self.compute_stddev()
```

5. Define the required methods for the various values. Here's the calculation of the mean:

```
def compute_mean(self) -> float:
    total, count = 0.0, 0
    for value, frequency in self.raw_counter.items():
        total += value * frequency
```

```
        count += frequency
    return total / count
```

6. Here's how we can calculate the standard deviation:

```
def compute_stddev(self) -> float:
    total, count = 0.0, 0
    for value, frequency in self.raw_counter.items():
        total += frequency * (value - self.mean) ** 2
        count += frequency
    return math.sqrt(total / (count - 1))
```

Note that this calculation requires that the mean is computed first and the `self.mean` instance variable has been created. This internal state change from no known mean to a known mean to a known standard deviation is a potential complication that requires clear documentation.

The raw data for this example is at <https://www.itl.nist.gov/div898/handbook//datasets/NEGIZ4.DAT>. This file has an awkwardly complicated layout because there are 50 lines of header text in front of the data. Further, the file isn't in a common CSV format. For these reasons, it's easier to work with summarized data.

The repository of code for this book includes a file named `data/binned.csv` that has the binned summary data. This data has three columns: `size_code`, `size`, and `frequency`. We're only interested in `size_code` and `frequency`.

Here's how we can build a suitable Counter object from this file:

```
>>> from pathlib import Path
>>> import csv
>>> from collections import Counter

>>> data_path = Path.cwd() / "data" / "binned.csv"
>>> with data_path.open() as data_file:
...     reader = csv.DictReader(data_file)
...     extract = {
...         int(row['size_code']): int(row['frequency'])
```



```
...         for row in reader
...     }
>>> data = Counter(extract)
```

We've used a dictionary comprehension to create a mapping from `size_code` to the frequency of that code value. This is then provided to the `Counter` class to build a `Counter` object named `data` from this existing summary. We can provide this data to the `CounterStatistics` class to get useful summary statistics from the binned data. This looks like the following example:

```
>>> stats = CounterStatistics(data)
>>> print(f"Mean: {stats.mean:.1f}")
Mean: 10.4
>>> print(f"Standard Deviation: {stats.stddev:.2f}")
Standard Deviation: 4.17
```

We provided the data object to create an instance of the `CounterStatistics` class. Creating this instance will also immediately compute the summary statistics. No additional explicit method evaluations are required. These values are available as the `stats.mean` and `stats.stddev` attributes.

The processing cost to compute the statistics is paid initially. As we'll see below, a tiny incremental cost can be associated with any change to the underlying data.

## How it works...

This class encapsulates two complex algorithms, but doesn't include any of the data for those algorithms. The data is kept separately, in a `Counter` object. We wrote a high-level specification for the processing and placed it in the `__init__()` method. Then we wrote methods to implement the processing steps that were specified. We can set as many attributes as are needed, making this a very flexible approach.

The advantage of this design is that the attribute values can be used repeatedly. The cost of computation for the mean and standard deviation is paid once; each time an attribute

value is used, no further processing is required.

The disadvantage of this design is that any changes to the state of the underlying Counter object will render the CounterStatistics object's state obsolete and incorrect. If, for example, we added a few hundred more data values, the mean and standard deviation would need to be recomputed. A design that eagerly computes values is appropriate when the underlying Counter object isn't going to change.

## There's more...

If we need to do computations on stateful, mutable objects, we have several choices:

- Encapsulate the Counter object and make changes via the CounterStatistics class. This requires some care to expose enough methods of the data collection. We'll defer this kind of design until *Chapter 8*.
- Use lazy computations. See the *Using properties for lazy attributes* recipe in this chapter.
- Add a method to implement the computation of mean and standard deviation, so these can be recomputed after changing the underlying Counter object. This leads to refactoring the `__init__()` method to use this new computation method. We'll leave this as an exercise for the reader.
- Write documentation explaining the requirement to create a new CounterStatistics instance each time the underlying Counter object changes. This involves no code, merely an explicit statement of the constraints on the object's state.

## See also

- In the *Extending a built-in collection – a list that does statistics* recipe, we'll look at a different design approach where these new summary functions are used to extend a class definition.
- We'll look at a different approach in the *Using properties for lazy attributes* recipe. This alternative recipe will use properties to compute the attributes as needed.

- The wrap=extend design choice is also looked at in *Chapter 8*.

## Using `typing.NamedTuple` for immutable objects

In some cases, an object is a container of rather complex data, but doesn't really do very much processing on that data. Indeed, in many cases, we'll define a class that doesn't require any unique method functions. These classes are relatively passive containers of data items, without a lot of processing.

In many cases, Python's built-in container classes – `list`, `set`, or `dict` – can cover your use cases. The small problem is that the syntax for accessing an item in a dictionary or a list isn't quite as elegant as the syntax for accessing an attribute of an object.

How can we create a class that allows us to use `object.attribute` syntax instead of the more elaborate `object['attribute']`?

### Getting ready

There are two cases for any kind of class design:

- Is it stateless (or immutable)? Does it embody attributes with values that never change? This is a good example of a `NamedTuple`.
- Is it stateful (or mutable)? Will there be state changes for one or more attributes? This is the default for Python class definitions. An ordinary class is stateful. We can simplify creating stateful objects using the *Using dataclasses for mutable objects* recipe.

We'll define a class to describe simple playing cards that have a rank and a suit. Since a card's rank and suit don't change, we'll create a small stateless class for this. The `typing.NamedTuple` class serves as a handy base class for these kinds of class definitions.

### How to do it...

1. We'll define stateless objects as a subclass of `typing.NamedTuple`:

```
from typing import NamedTuple
```

2. Define the class name as an extension to `NamedTuple`. Include the attributes with their individual type hints:

```
class Card(NamedTuple):  
    rank: int  
    suit: str
```

Here's how we can use this class definition to create `Card` objects:

```
>>> eight_hearts = Card(rank=8, suit='\N{White Heart Suit}')  
>>> eight_hearts  
Card(rank=8, suit='♥')  
  
>>> eight_hearts.rank  
8  
  
>> eight_hearts.suit  
'♥'  
  
>>> eight_hearts[0]
```

We've created a new class, named `Card`, which has two attribute names: `rank` and `suit`. After defining the class, we can create an instance of the class. We built a single `Card` object, `eight_hearts`, with a rank of eight and a suit of ♥.

We can refer to attributes of this object with their name or their position within the tuple. When we use `eight_hearts.rank` or `eight_hearts[0]`, we'll see the value of the rank attribute because this attribute is defined first in the sequence of attribute names.

This kind of object is immutable. Here's an example of attempting to change the instance attributes:

```
>>> eight_hearts.suit = '\N{Black Spade Suit}'  
Traceback (most recent call last):  
...  
AttributeError: can't set attribute
```

We attempted to change the `suit` attribute of the `eight_hearts` object. This raised an `AttributeError` exception showing that instances of `NamedTuple` are immutable.



A tuple can contain objects of any type.

When a tuple contains mutable items, like lists, sets, or dictionaries, those objects remain mutable.

Only the top-level containing tuple is immutable. Lists, sets, or dictionaries within a tuple are mutable.

## How it works...

The `typing.NamedTuple` class lets us define a new subclass that has a well-defined list of attributes. A number of methods are created automatically to provide a minimal level of Python behavior. We can see an instance will display a readable text representation showing the values of the various attributes.

In the case of a `NamedTuple` subclass, the behavior is based on the way a built-in tuple instance works. The order of the attributes defines the comparison between tuples. Our definition of `Card`, for example, lists the `rank` attribute first. This means that we can easily sort cards by rank. For two cards of equal rank, the suits will be sorted into order. Because a `NamedTuple` is also a tuple, it works well as a member of a set or a key for a dictionary.

The two attributes, `rank` and `suit` in this example, are named as part of the class definition, but are implemented as instance variables. A variation on the tuple's `__new__()` method is created for us. This method has two parameters matching the instance variable names. The automatically created method will assign argument values to the instance variables when the object is created.

## There's more...

We can add methods to this class definition. For example, if each card has a number of points, we might want to extend the class to look like this example:

```
class CardPoints(NamedTuple):
    rank: int
    suit: str

    def points(self) -> int:
        if 1 <= self.rank < 10:
            return self.rank
        else:
            return 10
```

We've written a `CardPoints` class with a `points()` method that returns the points assigned to each rank. This point rule applies to games like *Cribbage*, not to games like *Blackjack*.

Because this is a tuple, the methods cannot add new attributes or change the attributes. In some cases, we build complex tuples from other tuples.

## See also

- In the *Designing classes with lots of processing* recipe, we looked at a class that is entirely processing and almost no data. It acts as the polar opposite of this class.

## Using dataclasses for mutable objects

We've noted two general kinds of objects in Python:

- **Immutable:** During design, we'll ask if something has attributes with values that never change. If the answer is yes, see the *Using `typing.NamedTuple` for immutable objects* recipe, which offers a way to build class definitions for immutable objects.
- **Mutable:** Will there be state changes for one or more attributes? In this case, we can either build a class from the ground up, or we can leverage the `@dataclass` decorator to create a class definition from a few attributes and type hints. This case is the focus of this recipe.

How can we leverage the `dataclasses` library to help design mutable objects?

## Getting ready

We'll look closely at a mutable object with an internal state to represent a hand of cards. While individual cards are immutable, they can be inserted into a hand and removed from a hand. In a game like Cribbage, the hand has a number of state changes. Initially, six cards are dealt to both players. The players will each lay away a pair of cards to create the crib. The remaining four cards are then played alternately to create scoring opportunities. The hands are then counted in isolation, with a slightly different mix of scoring opportunities. The dealer gets the score from counting the cards in the crib as an extra hand. (Yes, it's unfair initially, but the deal alternates, so it's eventually fair.)

We'll look at a simple collection to hold the cards and discard two that form the crib.

## How to do it...

1. To define data classes, we'll import the `@dataclass` decorator:

```
from dataclasses import dataclass
```

2. Define the new class using the `@dataclass` decorator:

```
@dataclass
class CribbageHand:
```

3. Define the various attributes with appropriate type hints. For this example, we'll expect a player to have a collection of cards represented by `list[CardPoints]`. Because each card is unique, we could also use a `set[CardPoints]` type hint:

```
cards: list[CardPoints]
```

4. Define any methods that change the state of the object:

```
def to_crib(self, card1: CardPoints, card2: CardPoints) -> None:
    self.cards.remove(card1)
```

Here's the complete class definition, properly indented:

```
@dataclass
class CribbageHand:
    cards: list[CardPoints]

    def to_crib(self, card1: CardPoints, card2: CardPoints) -> None:
        self.cards.remove(card1)
        self.cards.remove(card2)
```

This definition provides a single instance variable, `self.cards`, that can be used by any method that is written. Because we provided a type hint, the mypy program can check the class to be sure that it is being used properly.

Here's how it looks when we create an instance of this `CribbageHand` class:

```
>>> cards = [
... CardPoints(rank=3, suit='\N{WHITE DIAMOND SUIT}'),
... CardPoints(rank=6, suit='\N{BLACK SPADE SUIT}'),
... CardPoints(rank=7, suit='\N{WHITE DIAMOND SUIT}'),
... CardPoints(rank=1, suit='\N{BLACK SPADE SUIT}'),
... CardPoints(rank=6, suit='\N{WHITE DIAMOND SUIT}'),
... CardPoints(rank=10, suit='\N{WHITE HEART SUIT}')]
>>> ch1 = CribbageHand(cards)

>>> from pprint import pprint
>>> pprint(ch1)
CribbageHand(cards=[CardPoints(rank=3, suit='♠'),
                    CardPoints(rank=6, suit='♣'),
                    CardPoints(rank=7, suit='♠'),
                    CardPoints(rank=1, suit='♣'),
                    CardPoints(rank=6, suit='♠'),
                    CardPoints(rank=10, suit='♥')])
```



```
>>> [c.points() for c in ch1.cards]
[3, 6, 7, 1, 6, 10]
```

In the following example, the player decided (perhaps unwisely) to lay away the 3♦ and A♠ cards for the crib:

```
>>> ch1.to_crib(
...     CardPoints(rank=3, suit='\N{WHITE DIAMOND SUIT}'),
...     CardPoints(rank=1, suit='\N{BLACK SPADE SUIT}'))

>>> pprint(ch1)
CribbageHand(cards=[CardPoints(rank=6, suit='♠'),
                    CardPoints(rank=7, suit='♦'),
                    CardPoints(rank=6, suit='♦'),
                    CardPoints(rank=10, suit='♥')])

>>> [c.points() for c in ch1.cards]
[6, 7, 6, 10]
```

After the `to_crib()` method removed two cards from the hand, the remaining four cards were displayed. Another list comprehension was created with the point values of the remaining four cards.

## How it works...

The `@dataclass` decorator helps us define a class with several useful methods as well as a list of attributes drawn from the named variables and their type hints. We can see that an instance displays a readable text representation showing the values of the various attributes.

The attributes are named as part of the class definition, but are actually implemented as instance variables. In this example, there's only one attribute, `cards`. A very sophisticated `__init__()` method is created for us. In this example, it will have a parameter that matches the name of each instance variable and will assign the argument value to a matching instance variable.

The `@dataclass` decorator has a number of options to help us choose what features we want in the class. Here are the options we can select from and the default settings:

- `init=True`: By default, an `__init__()` method will be created with parameters to match the instance variables.
- `repr=True`: By default, a `__repr__()` method will be created to return a string showing the state of the object.
- `eq=True`: By default, the `__eq__()` and `__ne__()` methods are provided. These methods implement the `==` and `!=` operators.
- `order=False`: The `__lt__()`, `__le__()`, `__gt__()`, and `__ge__()` methods are not created automatically. These methods implement the `<`, `<=`, `>`, and `>=` operators.
- `unsafe_hash=False`: Normally, mutable objects do not have hash values, and cannot be used as keys for dictionaries or elements of a set. It's possible to have a `__hash__()` method added automatically, but this is rarely a sensible choice for mutable objects, which is why the option is called an “unsafe” hash.
- `frozen=False`: This creates an immutable object. See the *Using frozen dataclasses for immutable objects* recipe in this chapter for more details.

Because a great deal of code is written for us, we can focus on the attributes of the class definition. We can write the methods that are truly distinctive and avoid writing “boilerplate” methods that have obvious definitions.

## There's more...

A hand of cards requires an initialization method to provide the collection of `Card` objects. A default `__init__()` method can populate the collection.

Consider creating a deck of cards, in contrast to a hand of cards. The initial deck of cards is an example of a dataclass that doesn't need an initialization method to set the instance variables. Instead, a deck of cards needs a customized `__init__()` method without any parameters; it always creates the same collection of 52 `Card` objects. This means we'll use

`init=False` in the `@dataclass` decorator to define this method for a `Deck` class definition.

The general pattern for `@dataclass` definitions is to provide class-level names, which are used to both define the instance variables and also create the initialization method, `__init__()`. This covers a common use case for stateful objects.

In some cases, however, we want to define a class-level variable that is *not* used to create instance variables, but will remain a class-level variable. This is done with the `ClassVar` type hint. A `ClassVar` type indicates a class-level variable that is not part of the instance variables or the `__init__()` method.

In the following example, we'll create a class variable with a sequence of suit strings:

```
import random

from typing import ClassVar

@dataclass(init=False)
class Deck:
    SUITS: ClassVar[tuple[str, ...]] = (
        '\N{Black Club Suit}',
        '\N{White Diamond Suit}',
        '\N{White Heart Suit}',
        '\N{Black Spade Suit}'
    )

    cards: list[CardPoints]

    def __init__(self) -> None:
        self.cards = [
            CardPoints(rank=r, suit=s)
            for r in range(1, 14)
```

This example class definition provides a class-level variable, `SUITS`, which is part of the `Deck` class. This variable is a tuple of the characters used to define the suits.

The `cards` variable has a hint claiming it will have the `list[CardPoints]` type. This information is used by the `mypy` program to confirm that the body of the `__init__()`

method performs the proper initialization of this attribute. It also confirms that this attribute is used appropriately by other classes.

## See also

- See the *Using typing.NamedTuple for immutable objects* recipe for a way to build class definitions for stateless objects.
- The *Using a class to encapsulate data and processing* recipe covers techniques for building a class without the additional methods created by the `@dataclass` decorator.

## Using frozen dataclasses for immutable objects

In the *Using typing.NamedTuple for immutable objects* recipe, we saw how to define a class that has a fixed set of attributes. The attributes can be checked by the `mypy` program to ensure that they're being used properly. In some cases, we might want to make use of the slightly more flexible dataclass to create an immutable object.

One potential reason for using a dataclass is because it can have more complex field definitions than a `NamedTuple` subclass. Another potential reason is the ability to customize the initialization and the hashing function that is created. Because a `NamedTuple` is essentially a tuple, there's limited ability to fine-tune the behavior of the instances in this class.

## Getting ready

We'll revisit the idea of defining simple playing cards with rank and suit. The rank can be modeled by an integer between 1 (ace) and 13 (king.) The suit can be modeled by a single Unicode character from the set `{ '♠', '♥', '♦', '♣' }`. Since a card's rank and suit don't change, we'll create a small, frozen dataclass for this.

## How to do it...

1. From the `dataclasses` module, import the `dataclass` decorator:

```
from dataclasses import dataclass
```

2. Start the class definition with the `@dataclass` decorator, using the `frozen=True` option to ensure that the objects are immutable. We've also included `order=True` so that the comparison operators are defined, allowing instances of this class to be sorted into order:

```
@dataclass(frozen=True, order=True)
class Card:
```

3. Provide the attribute names and type hints for the attributes of each instance of this class:

```
    rank: int
    suit: str
```

We can use these objects in code as follows:

```
>>> eight_hearts = Card(rank=8, suit='\N{White Heart Suit}')
>>> eight_hearts
Card(rank=8, suit='♥')
>>> eight_hearts.rank
8
>>> eight_hearts.suit
'♥'
```

We've created an instance of the `Card` class with a specific value for the `rank` and `suit` attributes. Because the object is immutable, any attempt to change the state will result in an exception that looks like the following example:

```
>>> eight_hearts.suit = '\N{Black Spade Suit}'
Traceback (most recent call last):
...
dataclasses.FrozenInstanceError: cannot assign to field 'suit'
```

This shows an attempt to change an attribute of a frozen dataclass instance.

The `dataclasses.FrozenInstanceError` exception is raised to signal that this kind of operation is not permitted.

## How it works...

This `@dataclass` decorator adds a number of built-in methods to a class definition. As we noted in the *Using dataclasses for mutable objects* recipe, there are a number of features that can be enabled or disabled. Each feature may lead us to include one or several individual methods in the class definition.

## There's more...

The `@dataclass` initialization method is quite sophisticated. We'll look at one feature that's sometimes handy for defining optional attributes.

Consider a class that can hold a hand of cards. While the common use case provides a set of cards to initialize the hand, we can also have hands that might be built incrementally, starting with an empty collection and adding cards during the game.

We can define this kind of optional attribute using the `field()` function from the `dataclasses` module. The `field()` function lets us provide a function to build default values, called `default_factory`. We'd use it as shown in the following example:

```
from dataclasses import dataclass, field

@dataclass(frozen=True, order=True)
class Hand:
    cards: list[CardPoints] = field(default_factory=list)
```

The `Hand` dataclass has a single attribute, `cards`, which is a list of `CardPoints` objects. The `field()` function provides a default factory: in the event no initial value is provided, the `list()` function will be executed to create a new, empty list.

We can create two kinds of hands with this dataclass. Here's the conventional example, where we deal six cards:

```

>>> cards = [
... CardPoints(rank=3, suit='\N{WHITE DIAMOND SUIT}'),
... CardPoints(rank=6, suit='\N{BLACK SPADE SUIT}'),
... CardPoints(rank=7, suit='\N{WHITE DIAMOND SUIT}'),
... CardPoints(rank=1, suit='\N{BLACK SPADE SUIT}'),
... CardPoints(rank=6, suit='\N{WHITE DIAMOND SUIT}'),
... CardPoints(rank=10, suit='\N{WHITE HEART SUIT}')]
>>>
>>> h = Hand(cards)

```

The `Hand()` type expects a single attribute, matching the definition of the attributes in the class. This is optional, and we can build an empty hand as shown in this example:

```

>>> crib = Hand()
>>> d3 = CardPoints(rank=3, suit='\N{WHITE DIAMOND SUIT}')
>>> h.cards.remove(d3)
>>> crib.cards.append(d3)

>>> from pprint import pprint
>>> pprint(crib)
Hand(cards=[CardPoints(rank=3, suit='♦')])

```

In this example, we've created a `Hand()` instance with no argument values, assigned to the `crib` variable. Because the `cards` attribute was defined with a field that provided a `default_factory`, the `list()` function will be used to create an empty list for the `cards` attribute.

## See also

- The *Using dataclasses for mutable objects* recipe covers some additional topics on using dataclasses to avoid some of the complexities of writing class definitions.

## Optimizing small objects with `__slots__`

The general case for an object allows a dynamic collection of attributes. There's a special case for an object with a fixed collection of attributes based on the tuple class. We looked at both of these in the *Using typing.NamedTuple for immutable objects* recipe.

There's a middle ground. We can also define an object with a fixed number of attributes, but the values of the attributes can be changed. By changing the class from an unlimited collection of attributes to a fixed set of attributes, it turns out that we can also save memory and processing time.

How can we create optimized classes with a fixed set of attributes?

## Getting ready

Generally, Python allows adding attributes to an object. This can be undesirable, particularly when working with a large number of objects. The flexibility of the way most class definitions use a dictionary has a cost in memory use. Using specific `__slots__` names limits the class to the named attributes, saving memory.

The card game of Cribbage, for example, has a few components:

- A deck of cards.
- Two players, who will alternate in the role of dealer and opponent.

This small domain of things seems like a candidate for a class definition. Each player has a hand of cards and a score. The player's role is an interesting complication. There are important differences in the two roles.

- The player who is the dealer gets the crib cards.
- If the starter card is a Jack, the player in the dealer role gets points for this.
- The opponent plays the first card.
- The opponent counts their hand first.
- The dealer plays from their hand, but counts their hand and the crib.

The specific order of play and counting hands is important because the first player to pass 120 points is the winner, no matter what state the game is in.

It seems like the Cribbage game includes a deck of cards and two players. The crib – which



belongs to the dealer – can be seen as a feature of the game overall. We'll look at ways to switch the role between dealer and opponent when a new round of play starts.

## How to do it...

We'll leverage the `__slots__` special name when creating the class:

1. Define the class with a descriptive name:

```
class Cribbage:
```

2. Define the list of attribute names. This identifies the only two attributes that are allowed for instances of this class. Any attempt to add another attribute will raise an `AttributeError` exception:

```
__slots__ = ('deck', 'players', 'crib', 'dealer', 'opponent')
```

3. Add an initialization method. This must create instance variables for the named slots:

```
def __init__(  
    self,  
    deck: Deck,  
    player1: Player,  
    player2: Player  
) -> None:  
    self.deck = deck  
    self.players = [player1, player2]  
    random.shuffle(self.players)  
    self.dealer, self.opponent = self.players  
    self.crib = Hand()
```

The `Deck` class definition is shown in the *Using dataclasses for mutable objects* recipe in this chapter.

4. Add methods to update the collection. For this example, we've defined a method to switch roles.

```
def new_deal(self) -> None:
    self.deck.shuffle()
    self.players = list(reversed(self.players))
    self.dealer, self.opponent = self.players
    self.crib = Hand()
```

Here's how we can use this class to build a hand of cards. We'll need the definition of the Card class based on the example in the *Using typing.NamedTuple for immutable objects* recipe:

```
>>> deck = Deck()
>>> c = Cribbage(deck, Player("1"), Player("2"))
>>> c.dealer
Player(name='2')
>>> c.opponent
Player(name='1')
>>> c.new_deal()
>>> c.dealer
Player(name='1')
>>> c.opponent
Player(name='2')
```

The initial Cribbage object was created with a Deck and two Player instances. These three objects filled in the the deck and players slots. The `__init__()` method then randomized the players, making one of them the dealer and the other the opponent. The crib was initialized to an empty Hand instance.

The `new_deal()` method makes a number of changes to the state of the Cribbage instance. This is revealed when the dealer and opponent attributes are examined.

Here's what happens if we try to create a new attribute:

```
>>> c.some_other_attribute = True
Traceback (most recent call last):
...
AttributeError: 'Cribbage' object has no attribute 'some_other_attribute'
```

We attempted to create an attribute named `some_other_attribute` on the `Cribbage` object, `c`. This raised an `AttributeError` exception. Using `__slots__` means that new attributes cannot be added to an instance of the class.

## How it works...

When we create an object instance, the steps in the process are defined in part by the object's class and the built-in `type()` function. Implicitly, a class has a special `__new__()` method that handles the internal house-keeping required to create a new, empty object. After this, the `__init__()` method creates and initializes the attributes.

Python has three essential paths for creating instances of a class:

- The default behavior, defined by the built-ins `object` and `type()`, is used when we define a class without doing anything unusual. Each instance contains a `__dict__` attribute that is used to hold all other attributes. Because the object's attributes are kept in a dictionary, we can add, change, and delete attributes freely. This flexibility requires the use of additional memory for the dictionary object inside each instance.
- The `__slots__` behavior avoids creating the `__dict__` attribute. Because the object has only the attributes named in the `__slots__` sequence, we can't add or delete attributes. We can change the values of the defined attributes. This lack of flexibility means that less memory is used for each object.
- The subclass of `tuple` behavior defines immutable objects. An easy way to create these classes is with `typing.NamedTuple` as a parent class. Once built, the instances are immutable and cannot be changed. While it's possible to directly subclass `tuple`, the extra features of a `NamedTuple` seem to make this ideal.

A large application might be constrained by the amount of memory used, and switching the class with the largest number of instances to `__slots__` can lead to an improvement in performance.

## There's more...

It's possible to tailor the way the `__new__()` method works to replace the default `__dict__` attribute with a different kind of dictionary. This is an advanced technique because it exposes the inner workings of classes and objects.

Python relies on a metaclass to create instances of a class. The default metaclass is the type class. The idea is that the metaclass provides a few pieces of functionality that are used to create each object. Once the empty object has been created, then the class's `__init__()` method will initialize the empty object.

Generally, a metaclass will provide a definition of `__new__()`, and perhaps `__prepare__()`, if there's a need to customize the object. There's a widely used example in the Python Language Reference document that tweaks the namespace used to create a class.

For more details, see <https://docs.python.org/3/reference/datamodel.html#metaclass-example>.

## See also

- The more common cases of an immutable object or a completely flexible object are covered in the *Using `typing.NamedTuple` for immutable objects* recipe.

## Using more sophisticated collections

Python has a wide variety of built-in collections. In *Chapter 4*, we looked at them closely. In the *Choosing a data structure* recipe, we provided a kind of decision tree to help locate the appropriate data structure from the available choices.

When we consider built-in types and other data structures in the standard library, we have more choices, and more decisions to make. How can we choose the right data structure for our problem?

## Getting ready

Before we put data into a collection, we'll need to consider how we'll gather the data, and what we'll do with the collection once we have it. The big question is always how we'll identify a particular item within the collection. We'll look at a few key questions that we need to answer to help select a proper collection for our needs.

Here's an overview of some of the alternative collections. The `collections` module contains a number of variations on the built-in collections. These include the following:

- `deque`: A double-ended queue. This is a mutable sequence with optimizations for pushing and popping from each end. Note that the class name starts with a lowercase letter; this is atypical for Python.
- `defaultdict`: A mapping that can provide a default value for a missing key. Note that the class name starts with a lowercase letter; this is atypical for Python.
- `Counter`: A mapping that is designed to count the number of occurrences of distinct keys. This is sometimes called a multiset or a bag.
- `ChainMap`: A mapping that combines several dictionaries into a single mapping.

The `heapq` module includes a priority queue implementation. This is a specialized library that leverages the built-in list sequence to maintain items in a sorted order.

The `bisect` module includes methods for searching a sorted list. This creates some overlap between the dictionary features and the list features.

Additionally, there's an `OrderedDict` class in the `collections` module. Starting with Python 3.7, the dictionary keys for an ordinary dictionary are retained in the order they were created, making the `OrderedDict` class redundant.

## How to do it...

There are a number of questions we need to answer to decide if we need a library data collection instead of one of the built-in collections:

1. Is the structure a buffer between the producer and the consumer? Does some part of

the algorithm produce data items and another part consume the data items?

- A queue is used for **First-In-First-Out (FIFO)** processing. Items are inserted at one end and consumed from the other end. We can use `list.append()` and `list.pop(0)` to simulate this, though `collections.deque` will be more efficient; we can use `deque.append()` and `deque.popleft()`.
  - A stack is used for **Last-In-First-Out (LIFO)** processing. Items are inserted and consumed from the same end. We can use `list.append()` and `list.pop()` to simulate this, though `collections.deque` will be more efficient; we can use `deque.append()` and `deque.pop()`.
  - A priority queue (or heap queue) keeps the queue sorted in some order, distinct from the arrival order. We can try to simulate this by using the `list.append()`, `list.sort(key=lambda x:x.priority)`, and `list.pop(-1)` operations to keep items in priority order. Performing a sort after each insert can make it inefficient. Using the `heapq` module can be more efficient. The `heapq` module has functions for creating and updating heaps.
2. How do we want to deal with missing keys from a dictionary?
    - Raise an exception. This is the way the built-in `dict` class works.
    - Create a default item. This is how `collections.defaultdict` works. We must provide a function that returns the default value. Common examples include `defaultdict(int)` and `defaultdict(float)` to use a default value of 0 or 0.0. We can also use `defaultdict(list)` and `defaultdict(set)` to create dictionary-of-list or dictionary-of-set structures.
    - The `defaultdict(int)` used to create a dictionary for counting items is so common that the `collections.Counter` class does exactly this.
  3. How do we want to handle the order of keys in a dictionary? Generally, Python above version 3.6 keeps the keys in insertion order. If we want a different order, we'll have to sort them manually.

#### 4. How will we build the dictionary?

- We have a simple algorithm to create items. In this case, a built-in `dict` object may be sufficient.
- We have multiple dictionaries that will need to be merged. This can happen when reading configuration files. We might have an individual configuration, a system-wide configuration, and a default application configuration that all need to be merged into a single dictionary using a `ChainMap` collection.

### How it works...

There are two principle resource constraints on data processing:

- Storage
- Time

All of our programming must respect these constraints. In most cases, the two are inverses: anything we do to reduce storage use tends to increase processing time, and anything we do to reduce processing time increases storage use. Algorithm and data structure design seeks to find an optimal balance among the constraints.

The time aspect is formalized via a complexity metric. There are several ways to describe the complexity of an algorithm:

- Complexity  $O(1)$  doesn't change with the volume of data. For some collections, the actual overall long-term average is nearly  $O(1)$  with minor exceptions. Many dictionary operations are  $O(1)$ . Appending to a list, and popping from the end of a list is very fast, making a LIFO stack very efficient. Popping from the front of a list is  $O(n)$ , making a FIFO queue built from a simple list rather expensive; the `deque` class and `heapq` module remedy this with better designs.
- Complexity described as  $O(\log n)$  means the cost grows more slowly than the volume of data,  $n$ . The `bisect` module lets us search a sorted list more efficiently than the `list` class by dividing the list into halves. Note that sorting the list in the first place is

$O(n \log n)$ , so there needs to be a great many searches to amortize the cost of sorting.

- Complexity described as  $O(n)$  means the cost grows as the volume of data,  $n$ , grows. Finding an item in a list has this complexity. If the item is at the end of the list, all  $n$  items must be checked. Sets and mappings don't have this problem, and have nearly  $O(1)$  complexity.
- A complexity described as  $O(n \log n)$  grows more quickly than the volume of data. Sorting a list tends to have this complexity. For this reason, it helps to minimize or eliminate sorting large volumes of data.
- There are even worse cases. Some algorithms have a complexity of  $O(n^2)$ ,  $O(2^n)$ , or even  $O(n!)$ . We'd like to avoid these kinds of very expensive algorithms through clever design and good choice of data structure. These can be deceptive in practice. We may be able to work out an  $O(2^n)$  algorithm that seems to perform well on small test cases where  $n$  is 3 or 4. In these cases, there are only 8 or 16 combinations. If real data involves 70 items, the number of combinations is on the order of  $10^{22}$ , a number with 22 digits.

The various data structures available in the standard library reflect a number of time and storage trade-offs.

## There's more...

As a concrete and extreme example, let's look at searching a web log file for a particular sequence of events. We have two overall design strategies:

- Read all of the events into a list structure with something like `file.read().splitlines()`. We can then use a `for` statement to iterate through the list looking for the combination of events. While the initial read may take some time, the search will be very fast because the log is all in memory.
- Read and process each individual event from a log file. When a log entry is part of the searched-for pattern, it makes sense to save only this event in a subset of the log. We might use a `defaultdict` with a session ID or client IP address as the key and a



list of events as the value. This will take longer to read the logs, but the resulting structure in memory will be much smaller than a list of all log entries.

The first algorithm, reading everything into memory, can be wildly impractical. On a large web server, the logs might involve hundreds of gigabytes of data. Logs can easily be too large to fit into any computer's memory.

The second approach has a number of alternative implementations:

- **Single process:** The general approach to most of the Python recipes here assumes that we're creating an application that runs as a single process.
- **Multiple processes:** We might expand the row-by-row search into a multi-processing application using the `multiprocessing` or `concurrent.futures` packages. These packages let us create a collection of worker processes, each of which can process a subset of the available data and return the results to a consumer that combines the results. On a modern multiprocessor, multi-core computer, this can be a very effective use of resources.
- **Multiple hosts:** The extreme case requires multiple servers, each of which handles a subset of the data. This requires more elaborate coordination among the hosts to share result sets. Generally, it can work out well to use a framework such as **Dask** or **Spark** for this kind of processing. While the `multiprocessing` module is quite sophisticated, tools like **Dask** are even more suitable for large-scale computation.

We'll often decompose a large search into map and reduce processing. The map phase applies some processing or filtering to every item in the collection. The reduce phase combines map results into summary or aggregate objects. In many cases, there is a complex hierarchy of Map-Reduce stages applied to the results of previous Map-Reduce operations.

## See also

- See the *Choosing a data structure* recipe in *Chapter 4*, for a foundational set of decisions for selecting data structures.

## Extending a built-in collection – a list that does statistics

In the *Designing classes with lots of processing* recipe, we looked at a way to distinguish between a complex algorithm and a collection. We showed how to encapsulate the algorithm and the data into separate classes. The alternative design strategy is to extend the collection to incorporate a useful algorithm.

How can we extend Python’s built-in collections? How can we add features to the built-in list?

### Getting ready

We’ll create a sophisticated list class where each instance can compute the sums and averages of the items in the list. This will require an application to put only numbers in the list; otherwise, there will be `ValueError` exceptions raised.

We’re going to show methods that explicitly use generator expressions as places where additional processing can be included. Rather than use `sum(self)`, we’re going to emphasize `sum(v for v in self)` because there are two common future extensions: `sum(m(v) for v in self)` and `sum(v for v in self if f(v))`. These are the mapping and filtering alternatives where a mapping function, `m(v)`, is applied to each item; or a filter function, `f(v)`, is applied to pass or reject each item. Computing a sum of squares, for example, applies a mapping to compute the square of each value before summing.

### How to do it...

1. Pick a name for the list that also does simple statistics. Define the class as an extension to the built-in `list` class:

```
class StatsList(list[float]):
```

We can stick with a generic type hint of `list`. This is often too broad. Since the structure will contain numbers, it’s more sensible to use the narrower hint of `list[float]`.

When working with numeric data, **mypy** treats the `float` type as a superclass for

both float and int, saving us from having to define an explicit `Union[float, int]`.

2. Define the additional processing as methods. The `self` variable will be an object that has inherited all of the attributes and methods from the superclass. In this case, the superclass is `list[float]`. We'll use a generator expression here as a place where future changes might be incorporated. Here's a `sum()` method:

```
def sum(self) -> float:
    return sum(v for v in self)
```

3. Here's another method that we often apply to a list. This counts items and returns the size. We've used a generator expression to make it easy to add mappings or filter criteria if that ever becomes necessary:

```
def size(self) -> float:
    return sum(1 for v in self)
```

4. Here's the *mean* method:

```
def mean(self) -> float:
    return self.sum() / self.size()
```

5. Here are some additional methods. The `sum2()` method computes the sum of squares of values in the list. This is used to compute variance. The variance is then used to compute the standard deviation of the values in the list. Unlike with the previous `sum()` and `count()` methods, where there's no mapping, in this case, the generator expression includes a mapping transformation:

```
def sum2(self) -> float:
    return sum(v ** 2 for v in self)
def variance(self) -> float:
    return (
        (self.sum2() - self.sum() ** 2 / self.size())
        / (self.size() - 1)
    )
def stddev(self) -> float:
```

```
return math.sqrt(self.variance())
```

The `StatsList` class definition inherits all the features of a list object. It is extended by the methods that we added. Here's an example of creating an instance in this collection:

```
>>> subset1 = StatsList([10, 8, 13, 9, 11])
>>> data = StatsList([14, 6, 4, 12, 7, 5])
>>> data.extend(subset1)
```

We've created two `StatsList` objects, `subset1` and `data`, from literal lists of objects. We used the `extend()` method, inherited from the `list` superclass, to combine the two objects. Here's the resulting object:

```
>>> data
[14, 6, 4, 12, 7, 5, 10, 8, 13, 9, 11]
```

Here's how we can use the additional methods that we defined on this object:

```
>>> data.mean()
9.0
>>> data.variance()
11.0
```

We've displayed the results of the `mean()` and `variance()` methods. All the features of the built-in `list` class are also present in our extension.

## How it works...

One of the essential features of class definition is the concept of inheritance. When we create a superclass-subclass relationship, the subclass inherits all of the features of the superclass. This is sometimes called the generalization-specialization relationship. The superclass is a more generalized class; the subclass is more specialized because it adds or modifies features.

All of the built-in classes can be extended to add features. In this example, we added some statistical processing that created a subclass that's a specialized kind of list of numbers.

There's an important tension between the two design strategies:

- **Extending:** In this case, we extended a class to add features. The features are deeply entrenched with this single data structure, and we can't easily use them for a different kind of sequence.
- **Wrapping:** In designing classes with lots of processing, we kept the processing separate from the collection. This leads to some more complexity in juggling two objects.

It's difficult to suggest that one of these is inherently superior to the other. In many cases, we'll find that wrapping may have an advantage because it seems to be a better fit to the SOLID design principles. However, there will often be cases where it's appropriate to extend a built-in collection.

## There's more...

The idea of generalization can lead to superclasses that are abstractions. Because an abstract class is incomplete, it requires a subclass to extend it and provide missing implementation details. We can't make an instance of an abstract class because it would be missing features that make it useful.

As we noted in the *Choosing a data structure* recipe in *Chapter 4*, there are abstract superclasses for all of the built-in collections. Rather than starting from a concrete class, we can also start our design from an abstract base class.

We could, for example, start a class definition like this:

```
from collections.abc import MutableMapping

class MyFancyMapping(MutableMapping[int, int]):
    ... # etc.
```

In order to finish this class, we'll need to provide an implementation for a number of special methods:

- `__getitem__()`
- `__setitem__()`
- `__delitem__()`
- `__iter__()`
- `__len__()`

Each of these methods is missing from the abstract class; they have no concrete implementation in the `Mapping` class. Once we've provided workable implementations for each method, we can then make instances of the new subclass.

## See also

- In the *Designing classes with lots of processing* recipe, we took a different approach. In that recipe, we left the complex algorithms in a separate class.

## Using properties for lazy attributes

In the *Designing classes with lots of processing* recipe, we defined a class that eagerly computed a number of attributes of the data in a collection. The idea there was to compute the values as soon as possible, so that the attributes would have no further computational cost.

We described this as **eager** processing, since the work was done as soon as possible. The other approach is **lazy** processing, where the work is done as late as possible.

What if we have values that are used rarely, and are very expensive to compute? What can we do to minimize the up-front computation, and only compute values when they are truly needed?

## Getting ready...

For background, see the NIST Aerosol Particle Size case study: <https://www.itl.nist.gov/div898/handbook/pmc/section6/pmc62.htm>

See the *Designing classes with lots of processing* recipe in this chapter for more details on this dataset. Rather than work with the raw data, it can help to work with summary information contained in a Counter object. The recipe shows a mapping from particle size to number to a count of times the particular size was measured.

We want to compute some statistics on this Counter. We have two overall strategies for doing this:

- **Extend:** We covered this in detail in the *Extending a built-in collection – a list that does statistics* recipe, and we will look at other examples of extending a class in *Chapter 8*.
- **Wrap:** We can wrap the Counter object in another class that provides just the features we need. We'll look at this in *Chapter 8*.

A common variation on wrapping creates a statistical computation object separate from the data collection object. This variation on wrapping often leads to an elegant solution.

No matter which class architecture we choose, we also have two ways to design the processing:

- **Eager:** This means we'll compute the statistics as soon as possible. This was the approach followed in the *Designing classes with lots of processing* recipe.
- **Lazy:** This means we won't compute anything until it's required via a method function or property. In the *Extending a built-in collection – a list that does statistics* recipe, we added methods to a collection class. These additional methods are examples of lazy calculation. The statistical values are computed only when required.

The essential math for both designs is the same. The only question is when the computation is done.

## How to do it...

1. Define the class with a descriptive name:

```
class LazyCounterStatistics:
```

2. Write the initialization method to include the object to which this object will be connected. We've defined a method function that takes a Counter object as an argument value. This Counter object is saved as part of the Counter\_Statistics instance:

```
def __init__(self, raw_counter: Counter[int]) -> None:  
    self.raw_counter = raw_counter
```

3. Define some useful helper methods. Each of these is decorated with @property to make it behave like a simple attribute:

```
@property  
def sum(self) -> float:  
    return sum(  
        f * v  
        for v, f in self.raw_counter.items()  
    )  
  
@property  
def count(self) -> float:  
    return sum(  
        f  
        for v, f in self.raw_counter.items()  
    )
```

4. Define the required methods for the various values. Here's the calculation of the mean. This too is decorated with @property. The other methods can be referenced as if they are attributes, even though they are proper method functions:

```
@property  
def mean(self) -> float:  
    return self.sum / self.count
```



5. Here's how we can calculate the standard deviation. Note that we've been using `math.sqrt()`. Be sure to add the required import `math` statement in the Python module:

```
@property
def sum2(self) -> float:
    return sum(
        f * v ** 2
        for v, f in self.raw_counter.items()
    )
@property
def variance(self) -> float:
    return (
        (self.sum2 - self.sum ** 2 / self.count) /
        (self.count - 1)
    )
@property
def stddev(self) -> float:
    return math.sqrt(self.variance)
```

To show how this works, we'll apply an instance of this class to some summarized data. The repository of code for this book includes a `data/binned.csv` file that has the binned summary data. This data has three columns: `size_code`, `size`, and `frequency`. We're only interested in `size_code` and `frequency`.

Here's how we can build a suitable `Counter` object from this file:

```
>>> from pathlib import Path
>>> import csv
>>> from collections import Counter

>>> data_path = Path.cwd() / "data" / "binned.csv"
>>> with data_path.open() as data_file:
...     reader = csv.DictReader(data_file)
...     extract = {
...         int(row['size_code']): int(row['frequency'])
...         for row in reader
...     }
>>> data = Counter(extract)
```

We've used a dictionary comprehension to create a mapping from `size_code` to the frequency of that code value. This is then provided to the `Counter` class to build a `Counter` object named `data` from this existing summary.

Here's how we can analyze the `Counter` object:

```
>>> stats = LazyCounterStatistics(data)
>>> print(f"Mean: {stats.mean:.1f}")
Mean: 10.4
>>> print(f"Standard Deviation: {stats.stddev:.2f}")
Standard Deviation: 4.17
```

We provided the `data` object to create an instance of the `LazyCounterStatistics` class, the `stats` variable. When we print the value for the `stats.mean` property and the `stats.stddev` property, the methods are invoked to do the appropriate calculations of the various values.

The cost for the computation is not paid until a client object requests the `stats.mean` or `stats.stddev` property values. This will invoke a cascade of computation to compute these values.

When the underlying data is changed, the entire computation is performed again. This can be costly in the rare case of highly dynamic data. In the more common case of analyzing previously summarized data, this is quite efficient.

## How it works...

The idea of lazy calculation works out well when the value is used rarely. In this example, the count is computed twice as part of computing the variance and standard deviation.

A naïve lazy design may not be optimal in some cases when values are recomputed frequently. This is an easy problem to fix in general. We can always create additional local variables to cache intermediate results instead of recomputing them. We'll look at this later in this recipe.

To make this class look like it has performed eager calculations, we used the `@property` decorator. This makes a method appear to be an attribute. This can only work for methods

that have no argument values.

In all cases, an attribute that's computed eagerly can be replaced by a lazy property. The principle reason for creating eager attribute variables is to optimize computation costs. In the case where a computed result may not always be used, a lazy property can avoid an expensive calculation.

## There's more...

There are some situations in which we can further optimize a property to limit the amount of additional computation that's done when a value changes. This requires a careful analysis of the use cases in order to understand the pattern of updates to the underlying data.

In the situation where a collection is loaded with data and an analysis is performed, we can cache results to save computing them a second time. We might do something like this:

```
from typing import cast

class CachingLazyCounterStatistics:
    def __init__(self, raw_counter: Counter[int]) -> None:
        self.raw_counter = raw_counter
        self._sum: float | None = None
        self._count: float | None = None

    @property
    def sum(self) -> float:
        if self._sum is None:
            self._sum = sum(
                f * v
                for v, f in self.raw_counter.items()
            )
        return self._sum
```

This technique uses two attributes to save the results of the sum and count calculations, `self._sum` and `self._count`. These values will be computed once and returned as often as needed with no additional cost for recalculation.

The type hints show these attributes as being optional. Once the values for `self._sum` and

`self._count` have been computed, the values are no longer optional, but will be present. We describe this to tools like **mypy** with the `cast()` type hint. This hint tells type-checking tools to consider `self._sum` as being a float object, not a `float | None` object. There's no cost to this function as it does nothing; its purpose is to annotate the processing to show the design intent.

This caching optimization is helpful if the state of the `raw_counter` object never changes. In an application that updates the underlying `Counter`, this cached value would become out of date. That kind of application would need to reset the internal cache values of `self._sum` and `self._count` when the underlying `Counter` is updated.

## See also...

- In the *Designing classes with lots of processing* recipe, we defined a class that eagerly computed a number of attributes. This represents a different strategy for managing the cost of the computation.

## Creating contexts and context managers

A number of Python objects behave like context managers. Some of the most visible examples are file objects. We generally use `with path.open() as file:` to process a file in a context that can guarantee the resources are released. In *Chapter 2*, the *Managing a context using the with statement* recipe covers the basics of using a file-based context manager.

How can we create our own classes that act as context managers?

## Getting ready

We'll look at a function from *Chapter 3*, in the *Picking an order for parameters based on partial functions* recipe. This recipe introduced a function, `haversine()`, which has a context-like parameter used to adjust the answer from dimensionless radians to a useful unit of measure, such as kilometers, nautical miles, or US statute miles. In many ways, this distance factor is a kind of context, used to define the kinds of computations that are done.

What we want is to be able to use the `with` statement to describe an object that doesn't change very quickly; indeed the change acts as a kind of boundary, defining the scope of computations. We might want to use code like the following:

```
>>> with Distance(r=NM) as nm_dist:
...     print(f"{nm_dist(p1, p2)=: .2f}")
...     print(f"{nm_dist(p2, p3)=: .2f}")
nm_dist(p1, p2)=39.72
nm_dist(p2, p3)=30.74
```

The `Distance(r=NM)` constructor provides the definition of the context, creating a new object, `nm_dist`, that has been configured to perform the required calculation in nautical miles. This can be used only within the body of the `with` statement.

This `Distance` class definition can be seen as creating a partial function, `nm_dist()`. This function provides a fixed unit-of-measure parameter, `r`, for a number of following computations using the `haversine()` function.

There are a number of other ways to create partial functions, including a lambda object, the `functools.partial()` function, and callable objects. We looked at the partial function alternative in *Chapter 3*, in the *Picking an order for parameters based on partial functions* recipe.

## How to do it...

A context manager class has two special methods that we need to define:

1. Start with a meaningful class name:

```
class Distance:
```

2. Define an initializer that creates any unique features of the context. In this case, we want to set the units of distance that are used:

```
def __init__(self, r: float) -> None:
    self.r = r
```

3. Define the `__enter__()` method. This is called when the `with` statement block begins. The statement `with Distance(r=NM) as nm_dist` does two things. First, it creates the instance of the `Distance` class, then it calls the `__enter__()` method of that object to start the context. The return value from the `__enter__()` method is assigned to a local variable via the `as` clause. This isn't always required. For simple cases, the context manager often returns itself. If this method needs to return an instance in the same class, note that the class hasn't been fully defined yet, and the class name type hint must be provided as a string. For this recipe, we'll return a function, with the type hint based on `Callable`:

```
def __enter__(self) -> Callable[[Point, Point], float]:  
    return self.distance
```

4. Define the `__exit__()` method. When the context finishes, this method is invoked. This is where resources are released and cleanup can happen. In this example, nothing more needs to be done. The details of any exception are provided to this method; the method can silence the exception or allow it to propagate. If the return value from the `__exit__()` method is `True`, the exception is silenced. A return value of `False` or `None` will allow the exception to be seen outside the `with` statement:

```
def __exit__(  
    self,  
    exc_type: type[Exception] | None,  
    exc_val: Exception | None,  
    exc_tb: TracebackType | None  
    ) -> bool | None:  
    return None
```

5. Create a class (or define the methods of this class) that works within the context. In this case, the method will make use of a separately defined `haversine()` function from *Chapter 3*:

```

def distance(self, p1: Point, p2: Point) -> float:
    return haversine(
        p1.lat, p1.lon, p2.lat, p2.lon, R=self.r
    )

```

Most context-manager classes require a fairly large number of imports:

```

from collections.abc import Callable
from types import TracebackType
from typing import NamedTuple

```

This class has been defined to work with objects of the class `Point`. This can be a `NamedTuple`, `@dataclass`, or some other class that provides the required two attributes. Here's the `NamedTuple` definition:

```

class Point(NamedTuple):
    lat: float
    lon: float

```

This class definition provides a class, `Point`, with the required attribute names.

## How it works...

The context manager relies on the `with` statement doing a large number of things.

We'll put the following construct under a microscope:

```

>>> p1 = Point(38.9784, -76.4922)
>>> p2 = Point(36.8443, -76.2922)
>>> nm_distance = Distance(r=NM)
>>> with nm_distance as nm_calc:
...     print(f"{nm_calc(p1, p2)=:.2f}")
nm_calc(p1, p2)=128.48

```

The first line creates an instance of the `Distance` class. This has a value for the `r` parameter equal to the constant `NM`, allowing us to do computations in nautical miles. The `Distance` instance is assigned to the `nm_distance` variable.

When the `with` statement starts execution, the context manager object is notified by having the `__enter__()` method executed. In this case, the value returned by the `__enter__()` method is a function, with the type `Callable[[Point, Point], float]`. The function accepts two `Point` objects and returns a floating-point result. The `as` clause assigns this function object to the `nm_calc` name.

The `print()` function does its work using the `nm_calc` object. The object is a function that will compute a distance from two `Point` instances.

When the `with` statement finishes, the `__exit__()` method will be executed. For more complex context managers, this may involve closing files or releasing network connections. There are a great many kinds of context cleanup that might be necessary. In this case, there's nothing that needs to be done to clean up the context.

This has the advantage of defining a fixed boundary in which the partial function is used. In some cases, the computation inside the context manager might involve a database or complex web services, leading to a more complex `__exit__()` method.

## There's more...

The operation of the `__exit__()` method is central to making best use of a context manager. In the previous example, we use the following “do nothing” `__exit__()` method:

```
def __exit__(
    self,
    exc_type: type[Exception] | None,
    exc_val: Exception | None,
    exc_tb: TracebackType | None
) -> bool | None:
    # Cleanup goes here.
    return None
```

The point here is to allow any exception to propagate normally. We often see any cleanup processing replacing the `# Cleanup goes here.` comment. This is where buffers are flushed, files are closed, and error log messages are written.



Sometimes, we'll need to handle specific exception details. Consider the following snippet of an interactive session:

```
>>> p1 = Point(38.9784, -76.4922)
>>> p2 = Point(36.8443, -76.2922)
>>> with Distance(None) as nm_dist:
...     print(f"{nm_dist(p1, p2)=:.2f}")
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for *: 'NoneType' and 'int'
```

The `Distance` object was initialized with the `r` argument value set to `None`. While this code will lead to warnings from tools like **mypy**, it's syntactically valid. The `TypeError` traceback, however, doesn't point to `Distance`; it points to a line of code within the `haversine()` function.

We might want to report a `ValueError` instead of this `TypeError`. Here's a variation on the `Distance` class, which conceals the `TypeError`, replacing it with a `ValueError`:

```
class Distance_2:
    def __init__(self, r: float) -> None:
        self.r = r

    def __enter__(self) -> Callable[[Point, Point], float]:
        return self.distance

    def __exit__(
        self,
        exc_type: type[Exception] | None,
        exc_val: Exception | None,
        exc_tb: TracebackType | None
    ) -> bool | None:
        if exc_type is TypeError:
            raise ValueError(f"Invalid r={self.r!r}")
        return None

    def distance(self, p1: Point, p2: Point) -> float:
        return haversine(p1.lat, p1.lon, p2.lat, p2.lon, R=self.r)
```

This shows how we can examine the details of the exception in the `__exit__()` method.

The information provided parallels the `sys.exc_info()` function, and includes the exception's type, the exception object, and a traceback object with the `types.TracebackType` type.

## See also

- In the *Managing a context using the with statement* recipe in *Chapter 2*, we cover the basics of using a file-based context manager.

## Managing multiple contexts with multiple resources

We often use context managers with open files. Because the context manager can guarantee the OS resources are released, doing so prevents resource leaks. It can be used to prevent files from being closed without having all buffers flushed to persistent storage.

When multiple resources are being processed, it often means multiple context managers will be needed. For example, if we have three open files, we could require three nested `with` statements? How can we optimize or simplify multiple `with` statements?

## Getting ready

We'll look at creating a plan for a journey with multiple legs. Our starting data collection is a list of points that define our route. For example, traveling through Chesapeake Bay may involve starting in Annapolis, Maryland, sailing to Solomon's Island, Deltaville, Virginia, and then Norfolk, Virginia. For planning purposes, we'd like to think of this as three legs, instead of four points. A leg has a distance and takes time to traverse: computing time, speed, and distance is the essence of the planning problem.

We'll start with some foundational definitions before we run the recipe. First is the definition of a single point, with attributes of latitude and longitude:

```
@dataclass(frozen=True)
class Point:
    lat: float
    lon: float
```

A point can be built with a statement like this: `p = Point(38.9784, -76.4922)`. This lets us refer to `p.lat` and `p.lon` in subsequent computations. The use of attribute names makes the code much easier to read.

A leg is a pair of points. We can define it as follows:

```
@dataclass
class Leg:
    start: Point
    end: Point
    distance: float = field(init=False)
```

We've created this as a mutable object. The `distance` attribute has an initial value defined by the `dataclasses.field()` function. The use of `init=False` means the attribute is not provided when the object is initialized; it must be supplied after initialization.

Here's a context manager to create `Leg` objects from `Point` instances. This is similar to the context managers shown in the *Creating contexts and context managers* recipe. There is a tiny but important difference here. The `__init__()` saves a value for `self.r` to set the distance unit context. The default value is nautical miles:

```
from types import TracebackType

class LegMaker:
    def __init__(self, r: float=NM) -> None:
        self.last_point: Point | None = None
        self.last_leg: Leg | None = None
        self.r = r

    def __enter__(self) -> "LegMaker":
        return self
```

```
def __exit__(
    self,
    exc_type: type[Exception] | None,
    exc_val: Exception | None,
    exc_tb: TracebackType | None
) -> bool | None:
    return None
```

The important method, `waypoint()`, accepts a waypoint and creates a `Leg` object. The very first waypoint, the starting point for the voyage, will return `None`. All subsequent points will return a `Leg` object:

```
def waypoint(self, next_point: Point) -> Leg | None:
    leg: Leg | None
    if self.last_point is None:
        # Special case for the first leg
        self.last_point = next_point
        leg = None
    else:
        leg = Leg(self.last_point, next_point)
        d = haversine(
            leg.start.lat, leg.start.lon,
            leg.end.lat, leg.end.lon,
            R=self.r
        )
        leg.distance = round(d)
        self.last_point = next_point
    return leg
```

This method uses a cached `Point` object, `self.last_point`, and the next point, `next_point`, to create a `Leg` instance and then update that instance.

If we want to create an output file in CSV format, we'll need to use two context managers: one to create `Leg` objects, and another to manage the open file. We'll put this complex multi-context processing into a single function.

## How to do it...

1. We'll be working with the `csv` and `pathlib` modules. Additionally, this recipe will also make use of the `Iterable` type hint, and the `asdict` function from the `dataclasses` module:

```
from collections.abc import Iterable
import csv
from dataclasses import asdict
from pathlib import Path
```

2. Since we'll be creating a CSV file, we need to define the headers to be used for the CSV output:

```
HEADERS = ["start_lat", "start_lon", "end_lat", "end_lon", "distance"]
```

3. Define a function to transform complex objects into a dictionary suitable for writing each individual row. The input is a `Leg` object; the output is a dictionary with keys that match the `HEADERS` list of column names:

```
def flat_dict(leg: Leg) -> dict[str, float]:
    struct = asdict(leg)
    return dict(
        start_lat=struct["start"]["lat"],
        start_lon=struct["start"]["lon"],
        end_lat=struct["end"]["lat"],
        end_lon=struct["end"]["lon"],
        distance=struct["distance"],
    )
```

4. Define the function with a meaningful name. We'll provide two parameters: a list of `Point` objects and a `Path` object showing where the CSV file should be created. We've used `Iterable[Point]` as a type hint so this function can accept any iterable collection of `Point` instances:

```
def make_route_file(
    points: Iterable[Point], target: Path
) -> None:
```

5. Start the two contexts with a single `with` statement. This will invoke both `__enter__()` methods to prepare both contexts for work. This line can get long:

```
with (
    LegMaker(r=NM) as legger,
    target.open('w', newline='') as csv_file
):
```

6. Once the contexts are ready for work, we can create a CSV writer and begin writing rows:

```
writer = csv.DictWriter(csv_file, HEADERS)
writer.writeheader()
for point in points:
    leg = legger.waypoint(point)
    if leg is not None:
        writer.writerow(flat_dict(leg))
```

7. At the end of the context, do any final summary processing. This is not indented within the `with` statement's body; it is at the same indentation level as the `with` keyword itself:

```
print(f"Finished creating {target}")
```

By keeping this *outside* the `with` context, this message provides important evidence that the files were properly closed and all of the computations were completed.

## How it works...

The compound `with` statement created a number of context managers for us. All of the managers will have their `__enter__()` methods used to both start processing and, optionally, return an object usable within the context. The `LegMaker` class defined an `__enter__()` method that returned the `LegMaker` instance. The `Path.open()` method returns a `TextIO` object; these are also context managers.

When the context exits at the end of the `with` statement, all of the context manager `__exit__()` methods are called. This allows each context manager to do any finalization.

In the case of `TextIO` objects, this closes the external files, releasing any of the OS resources being used.

In the case of the `LegMaker` object, there is no finalization processing on context exit. A `LegMaker` object was created; the value returned from the `__enter__()` method is a reference to a method of this object. The logger callable will continue to operate correctly even outside the context. This is an odd special case that occurs in instances where there is no cleanup in the `__exit__()` method. If it's important to prevent further use of the logger callable, then the `__exit__()` method needs to make an explicit state change inside the `LegMaker` object so it raises an exception. One approach is for the `__exit__()` method to set the `self.r` value to `None`, which would prevent further use of the `waypoint()` method.

## There's more...

A context manager's job is to isolate details of resource management. The most common examples are files and network connections. We've shown the use of a context manager around an algorithm to help manage a cache with a single `Point` object.

When working with very large datasets, it's often helpful to use compression. This can create a different kind of context around the processing. The built-in `open()` method is generally assigned to the `io.open()` function in the `io` module. This means we can often replace `io.open()` with a function such as `bz2.open()` to work with compressed files.

We can replace an uncompressed file context manager with something like this:

```
import bz2

def make_route_bz2(points: Iterable[Point], target: Path) -> None:
    with (
        LegMaker(r=NM) as logger,
        bz2.open(target, "wt") as archive
    ):
        writer = csv.DictWriter(archive, HEADERS)
        writer.writeheader()
        for point in points:
            leg = logger.waypoint(point)
```

```
if leg is not None:
    writer.writerow(flat_dict(leg))
print(f"Finished creating {target}")
```

We've replaced the original `path.open()` method with `bz2.open(path)`. The rest of the context processing remains identical. This flexibility allows us to work with text files initially and later convert them to compressed files when the volume of data grows.

## See also

- In the *Managing a context using the with statement* recipe in *Chapter 2*, we cover the basics of using a file-based context manager.
- The *Creating contexts and context managers* recipe covers the core of creating a class that is a context manager.

## Join our community Discord space

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

<https://packt.link/dHrHU>







# 8

## More Advanced Class Design

In *Chapter 7*, we looked at some recipes that covered the basics of class design. In this chapter, we'll dive more deeply into Python classes and class design.

In the *Designing classes with lots of processing* and *Using properties for lazy attributes* recipes in *Chapter 7*, we identified a design choice that's central to object-oriented programming, the “wrap versus extend” decision. One way to add features is to create a new subclass via an extension. The other technique for adding features is to wrap an existing class, making it part of a new class.

In addition to direct inheritance, there are some other class extension techniques available in Python. A Python class can inherit features from more than one superclass. We call this design pattern a **mixin**.

In *Chapter 4* and *Chapter 5*, we looked at the core built-in data structures. We can combine and extend these collection definition features to create more complex data structures or

data structures with additional features.

In this chapter, we'll look at the following recipes:

- *Choosing between inheritance and composition – the "is-a" question*
- *Separating concerns via multiple inheritance*
- *Leveraging Python's duck typing*
- *Managing global and singleton objects*
- *Using more complex structures – maps of lists*
- *Creating a class that has orderable objects*
- *Deleting from a list of complicated objects*

There are a great many techniques of object-oriented class design available in Python. We'll start with a foundational design concept: making the design choice between using inheritance from a base class and wrapping a class to extend it.

## Choosing between inheritance and composition – the "is-a" question

In the *Using cmd to create command-line applications* recipe in *Chapter 6*, and the *Extending a built-in collection – a list that does statistics* recipe in *Chapter 7*, we looked at extending a class. In both cases, the class implemented in the recipe was a subclass of one of Python's built-in classes.

The idea of extension via inheritance is sometimes called the generalization-specialization relationship. It can also be called an **is-a relationship**. There's an important semantic issue:

- Do we mean that instances of the subclass are also instances of the superclass? This is an **is-a** relationship, an example of **inheritance**, where we **extend** a class, changing the implementation details of features.

- Or do we mean something else? Perhaps there's a **composition** or **association**, sometimes called a **has-a relationship**. In this case, we may **wrap** another class, adding or removing features.

One of the **SOLID** design principles, the **Liskov Substitution Principle**, requires any subclass to be a proper replacement for the superclass. We'll look at both the inheritance and composition techniques for creating new features for existing classes.

## Getting ready

For this recipe, we'll use models for a deck of playing cards as concrete examples. We'll look at several ways to design a collection.

The core ingredient for both implementations is the underlying Card object. We can define this using NamedTuple:

```
from typing import NamedTuple

class Card(NamedTuple):
    rank: int
    suit: str

Spades, Hearts, Diamonds, Clubs = ('\u2660', '\u2661', '\u2662', '\u2663')
```

We'll use this Card class in the rest of this recipe. What's important is the various kinds of collection representing a deck or hand; all have considerable overlaps in the kinds of features they support.

We have several common pattern for collections:

- **Aggregation:** Some objects are bound into collections, but the objects have a properly independent existence. While Card objects can be aggregated into a Hand collection, when the Hand object is deleted, the Card objects continue to exist.
- **Composition:** Some objects in collections do not have an independent existence. A Hand of cards cannot exist without a Player. When a Player instance leaves a game, the Hand object must also be removed.

- **Inheritance** (also called an **is-a** relationship): This is the idea that a Deck is a Hand with some extra features. We can extend a built-in class like `list` to implement this.

The distinction between aggregation and composition is very important when designing a database, where persistence of objects is the focus. In Python, the distinction is a minor nuance. The ordinary Python memory management will preserve the objects still referenced by collections or variables. We'll consider both to be examples of composition.

Once the relationships are understood, there are two distinct paths: *Composition or aggregation* or *Inheritance and extension*.

## How to do it...

This recipe has two separate mini-recipes: aggregation and inheritance.

### Composition or aggregation

Wrapping a collection object inside another class's instance variables has two common variants, sometimes called composition and aggregation. The nuanced difference doesn't matter in Python. Here's how we design a collection using composition:

1. Define the collection class.

To distinguish similar examples in this book, the name has a `_W` suffix to show it is a wrapper. This is not a generally recommended practice; it's only used here to emphasize the distinctions between class definitions in this recipe.

Here's the definition of the class:

```
class Deck_W:
```

2. Use the `__init__()` method of this class as one way to provide the underlying collection object. This will also initialize any stateful variables. We might create an iterator for dealing:

```
def __init__(self, cards: list[Card]) -> None:
    self.cards = cards
    self.deal_iter = iter(self.cards)
```

This uses a type hint, `list[Card]`, to show the source collection that will be wrapped.

3. Provide the methods appropriate to the aggregate object. The `shuffle()` method randomizes the internal list object. It also creates an iterator used to step through the list by the `deal()` method. We've provided a type hint on `deal()` to clarify that it returns `Card` instances:

```
def shuffle(self) -> None:
    random.shuffle(self.cards)
    self.deal_iter = iter(self.cards)

def deal(self) -> Card:
    return next(self.deal_iter)
```

Here's how we can use the `Deck_W` class. We'll be working with a list of `Card` objects. In this case, the domain variable was created from a list comprehension that generated all 52 combinations of 13 ranks and four suits:

```
>>> domain = list(
...     Card(r+1,s)
...     for r in range(13)
...     for s in (Spades, Hearts, Diamonds, Clubs)
... )
>>> len(domain)
52
```

We can use the items in this collection, `domain`, to create a second aggregate object that shares the same underlying `Card` objects. We'll build the `Deck_W` object from a list of `Card` objects:

```
>>> d = Deck_W(domain)

>>> import random
>>> random.seed(1)
>>> d.shuffle()
>>> [d.deal() for _ in range(5)]
[Card(rank=13, suit='♥'), Card(rank=3, suit='♥'), Card(rank=10, suit='♥'),
Card(rank=6, suit='♦'), Card(rank=1, suit='♦')]
```

## Inheritance and extension

Here's an approach to defining a class that extends one of the built-in collections of objects:

1. Start by defining the extension class as a subclass of a built-in collection. To distinguish similar examples in this book, the name has an `_X` suffix. The subclass relationship is a formal statement—a `Deck_X` instance is also a kind of `list`. Here's the class definition:

```
class Deck_X(list[Card]):
```

2. No additional code is needed to initialize the instance, as we'll use the `__init__()` method inherited from the `list` class.
3. No additional code is needed to update the deck, as we'll use other methods of the `list` class for adding, changing, or removing items from the `Deck_X` instance.
4. Provide the appropriate new features to the extended object. The `shuffle()` method randomizes the object as a whole. The collection here is `self`, because this method is an extension of the `list` class. The `deal()` object relies on an iterator created by the `shuffle()` method to step through the list, returning `Card` instances:

```
def shuffle(self) -> None:
    random.shuffle(self)
    self.deal_iter = iter(self)

def deal(self) -> Card:
    return next(self.deal_iter)
```

Here's how we can use the `Deck_X` class. First, we'll build a deck of cards:

```
>>> dx = Deck_X(  
...     Card(r+1,s)  
...     for r in range(13)  
...     for s in (Spades, Hearts, Diamonds, Clubs)  
... )  
>>> len(dx)  
52
```

Using only the deck-specific features for the `Deck_X` implementation looks exactly like the other implementation, `Deck_W`:

```
>>> import random  
>>> random.seed(1)  
>>> dx.shuffle()  
>>> [dx.deal() for _ in range(5)]  
[Card(rank=13, suit='♥'), Card(rank=3, suit='♥'), Card(rank=10, suit='♥'),  
Card(rank=6, suit='♦'), Card(rank=1, suit='♦')]
```

As we'll see below in *There's more...*, because `Deck_X` is a list, it has all of the methods of a list object. When designing a framework for others to use, this may be a bad idea. When designing an application, it's easy to avoid using the extra features.

## How it works...

Python implements the idea of inheritance via a clever search algorithm for finding methods (and attributes) of an object's class. The search works like this:

1. Examine the object's class for the method or attribute name.
2. If the name is not defined in the immediate class, then search in all of the parent classes for the method or attribute. The **Method Resolution Order (MRO)** defines the order in which these classes are searched.

Searching through the parent classes ensures two things:

- All methods defined in any superclass are available to subclasses.



- Any subclass can override a method to replace the superclass method. The `super()` function searches parent classes for the definition being overridden.

Because of this, a subclass of the `list` class inherits all the features of the parent class. It is a specialized extension of the built-in `list` class.

This also means that all methods have the potential to be overridden by a subclass. Some languages have ways to lock a method against extension. Because Python doesn't have this, a subclass can override any method.

The `super()` function allows a subclass to add features by wrapping the superclass version of a method. One way to use it is like this:

```
class SomeClass(Parent):
    def some_method(self) -> None:
        # do something extra
        super().some_method()
```

In this case, the `some_method()` method of a class will do something extra and then use the superclass version of the method. This allows us a handy way to extend selected methods of a class. We can preserve the superclass features while adding features unique to the subclass.

## There's more...

There are some huge differences between the two definitions, `Deck_W` and `Deck_X`. When wrapping, we get precisely the methods we defined and no others. When using inheritance, we receive a wealth of method definitions from the superclass. This leads to some additional behaviors in the `Deck_X` class that may not be desirable:

- We can use a variety of collections as a source to create `Deck_X` instances. This works because the `list` class has a number of features for converting Python collections to lists. The `Deck_W` class will only work for sequences offering the methods implicitly required by the `shuffle()` method. Further, the type hint of `list[Card]` will cause programs like **mypy** to raise errors for the use of other source collections.

- A `Deck_X` instance can be sliced and indexed outside the core sequential iteration supported by the `deal()` method.
- Because the `Deck_X` class is a list, it also works directly with the `iter()` function; it can be used as an iterable source of `Card` objects without the `deal()` method.

These differences are also important parts of deciding which technique to use. If the additional features are desirable, that suggests inheritance. If the additional features create problems, then composition might be a better choice.

## See also

- We've looked at built-in collections in *Chapter 4*. Also, in *Chapter 7*, we looked at how to define simple collections.
- In the *Designing classes with lots of processing* recipe, we looked at wrapping a class with a separate class that handles the processing details. We can contrast this with the *Using properties for lazy attributes* recipe of *Chapter 7*, where we put the complicated computations into the class as properties; this design relies on extension.

## Separating concerns via multiple inheritance

In the *Choosing between inheritance and composition – the "is-a" question* recipe earlier in the chapter, we looked at the idea of defining a `Deck` class that was a composition of playing card objects. For the purposes of that example, we treated each `Card` object as simply having rank and suit attributes. This created two small problems:

- The display for the card always showed a numeric rank. We didn't see J, Q, or K. Instead we saw 11, 12, and 13. Similarly, an ace was shown as 1 instead of A.
- Many games like *Cribbage* assign a point value to each rank. Generally, the face cards have 10 points. The remaining cards have points that match their rank.

Python's multiple inheritance lets us handle variations in card game rules while keeping a single, essential `Card` class. Using multiple inheritance lets us separate rules for specific games from generic properties of playing cards. We can combine a base class definition

with a mixin class that provides needed features.

Python multiple inheritance relies on a clever algorithm called **C3** to resolve various parent classes into a single list, in a useful order. When we combine multiple classes, they will have common parent classes, which now have multiple references. The C3 algorithm creates a linear list that respects all of the parent-child relationships.

## Getting ready

A practical extension to the Card class needs to be a mixture of two feature sets. Python lets us define a class that includes features from multiple parents. There are two parts to this pattern:

1. **Essential features:** These are the rank and suit. This also includes a method to show the Card object's value nicely as a string using "J", "Q", and "K" for court cards, and "A" for aces.
2. **Mixin features:** These are all of the less essential, game-specific features, such as the number of points allotted to each particular card.

The working application relies on a combination of features built from the essentials and the mixins.

## How to do it...

This recipe will create two hierarchies of classes, one for the essential Card and one for game-specific features including *Cribbage* point values:

1. Define the essential class. This is a generic Card class, suitable for ranks 2 to 10:

```
@dataclass(frozen=True)
class Card:
    """Superclass for cards"""
    rank: int
    suit: str

    def __str__(self) -> str:
        return f"{self.rank:2d} {self.suit}"
```

2. Define the subclasses to implement specializations. We need two subclasses of the Card class—the AceCard class and the FaceCard class, as defined in the following code:

```
class AceCard(Card):
    def __str__(self) -> str:
        return f" A {self.suit}"

class FaceCard(Card):
    def __str__(self) -> str:
        names = {11: "J", 12: "Q", 13: "K"}
        return f" {names[self.rank]} {self.suit}"
```

Each of this overrides the built-in `__str__()` method to provide distinct behaviors.

3. Define the core features required by the mixin classes. Use the `typing.Protocol` superclass to be sure the various implementations all provide the required features. The rank attribute is required by the protocol, and will be defined in the essential class. A `points()` method will be defined in the mixin classes. Here's how it looks:

```
from typing import Protocol

class PointedCard(Protocol):
    rank: int
    def points(self) -> int:
        ...
```

When writing type hint classes, the body can be `...` because this will be ignored by tools like `mypy`.

4. Define a mixin subclasses for additional features that will be added. For the game of *Cribbage*, the points for some cards are equal to the rank of the card, and face cards are 10 points:

```
class CribbagePoints(PointedCard):
    def points(self) -> int:
        return self.rank
```

```
class CribbageFacePoints(PointedCard):
    def points(self) -> int:
        return 10
```

5. Create the final concrete class definitions to combine an essential base class and all of the required mixin classes:

```
class CribbageCard(Card, CribbagePoints):
    pass

class CribbageAce(AceCard, CribbagePoints):
    pass

class CribbageFace(FaceCard, CribbageFacePoints):
    pass
```

Note that the `CribbagePoints` mixin is used for both `Card` and `AceCard` classes, allowing us to reuse code.

6. Define a function (or class) to create the appropriate objects based on the input parameters. This is often called a *factory function* or *factory class*. The objects being created will all be considered as subclasses of the `Card` class because it's first in the list of base classes:

```
def make_cribbage_card(rank: int, suit: str) -> Card:
    if rank == 1:
        return CribbageAce(rank, suit)
    elif 2 <= rank < 11:
        return CribbageCard(rank, suit)
    elif 11 <= rank:
        return CribbageFace(rank, suit)
    else:
        raise ValueError(f"invalid rank {rank}")
```

We can use the `make_cribbage_card()` function to create a shuffled deck of cards, as shown in this example interactive session:

```
>>> import random
>>> random.seed(1)
>>> deck = [make_cribbage_card(rank+1, suit) for rank in range(13) for suit
in SUITS]
>>> random.shuffle(deck)
>>> len(deck)
52

>>> [str(c) for c in deck[:5]]
['K ♠', '3 ♠', '10 ♠', '6 ♦', 'A ♦']
```

We can evaluate the `points()` method of each `Card` object:

```
>>> sum(c.points() for c in deck[:5])
30
```

The hand has two face cards, plus 3, 6, and ace, so the total points are 30.

## How it works...

Python's mechanism for finding a method (or attribute) of an object works like this:

1. Search the instance for the attribute.
2. Search in the class for the method or attribute.
3. If the name is not defined in the immediate class, then search all of the parent classes for the method or attribute. The parent classes are searched in a sequence called, appropriately, the **Method Resolution Order (MRO)**.

We can display the MRO using the `mro()` method of a class. Here's an example:

```
>>> c.__class__.__name__
'CribbageCard'

>>> from pprint import pprint
>>> pprint(c.__class__.mro())
[<class 'recipe_02.CribbageCard'>,
```

```
<class 'recipe_02.Card'>,
<class 'recipe_02.CribbagePoints'>,
<class 'recipe_02.PointedCard'>,
<class 'typing.Protocol'>,
<class 'typing.Generic'>,
<class 'object'>]
```

The `mro()` method of the `CribbageCard` class shows us the order that's used to resolve names. Because the class object uses an internal dict to store method definitions, the search is an extremely fast hash-based lookup of the attribute name.

## There's more...

There are several kinds of design concerns we can separate in the form of mixins:

- **Persistence and representation of state:** A mixin class could add methods to manage conversion to a consistent external representation like CSV or JSON notation.
- **Security:** A mixin class could add methods performs a consistent authorization check that applies to a number of base classes.
- **Logging:** A mixin class could introduce a logger with a definition consistent across a variety of classes.
- **Event signaling and change notification:** A mixin might report object state changes so one or more GUI widgets can refresh the display.

As an example, we'll create a mixin to introduce logging to cards. We'll define this class in a way that must be provided first in the list of superclasses. Since it's early in the MRO list, it uses the `super()` function to use methods defined by subsequent classes in the MRO list.

This class will add the `logger` attribute to each object that has the `PointedCard` protocol defined:

```
import logging

class Logged(Card, PointedCard):
    def __init__(self, rank: int, suit: str) -> None:
        self.logger = logging.getLogger(self.__class__.__name__)
        super().__init__(rank, suit)

    def points(self) -> int:
        p = super().points() # type: ignore [safe-super]
        self.logger.debug("points {0}", p)
        return p
```

Note that we've used `super().__init__()` to perform the `__init__()` method of any other classes defined. The order for these initializations comes from the class MRO. The simplest approach to have one class that defines the essential features of an object, and all other mixins add features in the form of additional methods to the essential object.

We've provided an overriding definition for `points()`. This will search other classes in the MRO list for an implementation of the `points()` method. Then it will log the results computed by the method from another mixin class.

The `# type: ignore [safe-super]` comment is a note to tools like **mypy** that do strict type-checking. When we look at the definitions of the `PointedCard` protocol, there's no definition for this method. From the tool's examination of the class hierarchy, it's possible that calling `super().points()` is unsafe. We're sure this won't happen in practice, because a mixin will always be present to define the `points()` method. We've flagged the unsafe use of `super()` as an error to be ignored.

Here are some classes that include the `Logged` mixin features:

```
class LoggedCribbageAce(Logged, AceCard, CribbagePoints):
    pass

class LoggedCribbageCard(Logged, Card, CribbagePoints):
    pass
```



```
class LoggedCribbageFace(Logged, FaceCard, CribbageFacePoints):  
    pass
```

Each of these classes are built from three separate class definitions. Since the `Logged` class is provided first, we're assured that all classes have consistent logging. We're also assured that any method in `Logged` can use `super()` to locate an implementation in the class list that follows it in the sequence of classes in the definition.

To make use of these classes, we'd need to define a `make_logged_card()` function to use these new classes.

## See also

- The method resolution order is computed when the class is created. The algorithm used is called C3. The process was originally developed for the Dylan language and is now also used by Python. The C3 algorithm ensures that each parent class is searched exactly once. It also ensures the relative ordering of superclasses is preserved; subclasses will be searched before any of their parent classes are examined. More information is available at <https://dl.acm.org/doi/10.1145/236337.236343>.
- When considering multiple inheritance, it's always essential to also consider whether or not a wrapper is a better design than a subclass. See the *Choosing between inheritance and composition – the "is-a" question* recipe.

## Leveraging Python's duck typing

When a design involves inheritance, there is often a clear relationship from a superclass to one or more subclasses. In the *Choosing between inheritance and composition – the "is-a" question* recipe of this chapter, as well as the *Extending a built-in collection – a list that does statistics* recipe in *Chapter 7*, we've looked at extensions that involve a proper subclass-superclass relationship.

In order to have classes that can be used in place of one another ("polymorphic" classes),

some languages require a common superclass. In many cases, the common class doesn't have concrete implementations for all of the methods; it's called an abstract superclass.

Python doesn't *require* common superclasses. The standard library offers the `abc` module to support creating abstract classes in cases where it can help to clarify the relationships among classes.

Instead of defining polymorphic classes with common superclasses, Python relies on **duck typing** to establish equivalence. This name comes from the quote:

*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.* (James Whitcomb Riley)

In the case of Python class relationships, if two objects have the same methods and the same attributes, these similarities have the same effect as having a common superclass. No formal definition of a common class is required.

This recipe will show how to exploit the concept of duck typing to create polymorphic classes. Instances of these classes can be used in place of each other, giving us more flexible designs.

## Getting ready

In some cases, it can be awkward to define a superclass for a number of loosely related implementation choices. For example, if an application is spread across several modules, it might be challenging to factor out a common superclass and put this by itself in a separate module where it can be imported into other modules. Instead of factoring out a common abstraction, it's sometimes easier to create classes that will pass the "duck test": the various classes have the same methods and attributes; therefore, they are effectively interchangeable, polymorphic classes.

## How to do it...

We'll define a pair of classes to show how this works. These classes will both simulate rolling a pair of dice. We'll create two distinct implementations that have enough common

features that they are interchangeable:

1. Start with a class, `Dice1`, with the required methods and attributes. In this example, we'll have one attribute, `dice`, that retains the result of the last roll, and one method, `roll()`, that changes the state of the dice:

```
import random

class Dice1:
    def __init__(self, seed: int | None = None) -> None:
        self._rng = random.Random(seed)
        self.roll()
    def roll(self) -> tuple[int, ...]:
        self.dice = (
            self._rng.randint(1, 6),
            self._rng.randint(1, 6))
        return self.dice
```

2. Define another class, `Dice2`, with the same methods and attributes. Here's a somewhat more complex definition that creates a class that has the same signature as the `Dice1` class:

```
import random

class Die:
    def __init__(self, rng: random.Random) -> None:
        self._rng = rng
    def roll(self) -> int:
        return self._rng.randint(1, 6)

class Dice2:
    def __init__(self, seed: int | None = None) -> None:
        self._rng = random.Random(seed)
        self._dice = [Die(self._rng) for _ in range(2)]
        self.roll()
    def roll(self) -> tuple[int, ...]:
        self.dice = tuple(d.roll() for d in self._dice)
        return self.dice
```

At this point, the two classes, `Dice1` and `Dice2`, can be interchanged freely. Here's a

function that accepts either class as an argument, creates an instance, and yields several rolls of the dice:

```
from collections.abc import Iterator

def roller(
    dice_class: type[Dice1 | Dice2],
    seed: int | None = None,
    *,
    samples: int = 10
) -> Iterator[tuple[int, ...]]:
    dice = dice_class(seed)
    for _ in range(samples):
        yield dice.roll()
```

We can use this function providing either the `Dice1` class or `Dice2` class for the dice argument value. The `type[Dice1 | Dice2]` type hint specifies a union of multiple equivalent classes. This function creates an instance of the given class in the `dice` parameter, and can even provide the seed value. Using a known seed creates reproducible results, often required for unit testing, and also used for reproducing statistical studies that involve randomized selection.

The following interactive session shows the `roller()` function being applied to both classes:

```
>>> list(roller(Dice1, 1, samples=5))
[(1, 3), (1, 4), (4, 4), (6, 4), (2, 1)]

>>> list(roller(Dice2, 1, samples=5))
[(1, 3), (1, 4), (4, 4), (6, 4), (2, 1)]
```

The objects built from `Dice1` and `Dice2` have enough similarities that they're indistinguishable.

## How it works...

We've created two classes with identical collections of attributes and methods. This is the essence of duck typing. Because of the way Python searches through a sequence of dictionaries for matching names, classes do not need to have a common superclass to be interchangeable.

It can be helpful to define a union of related classes. An alternative is to define a common Protocol that the classes adhere to. It's not necessary for each class to explicitly inherit from the Protocol definition, but it can make it more clear to your readers to do this. Tools like **mypy** can discern whether or not a class fits a Protocol, which is how duck typing works.

## There's more...

In the definition of the `roller()` function, we used the following type hint:

```
dice: type[Dice1 | Dice2].
```

It's often helpful to make this explicit with code like this:

```
Dice = Dice1 | Dice2
```

This can be easily extended as new alternative definitions are added. Client classes can then use `type[Dice]` to refer to the union of alternatives.

An alternative is to define a protocol. A protocol defines a generic type with only the common features the various implementations will share:

```
class DiceP(Protocol):  
    def roll(self) -> tuple[int, ...]:  
        ...
```

It helps to create type hints later in the development process. After creating alternative implementations, it's easy to define a type that's a union of the various choices. If more implementations arise, they can be added to the union.

A protocol permits easier expansion of the alternatives. The protocol defines only the relevant features of the implementations. This is often done by refactoring the signatures of client methods and attributes to refer to the protocol class.

## See also

- The duck type question is implicit in the *Choosing between inheritance and composition – the "is-a" question* recipe; if we leverage duck typing, we're also making a claim that two classes are not the same thing. When we bypass inheritance, we are implicitly claiming that the is-a relationship doesn't hold.
- When looking at the *Separating concerns via multiple inheritance* recipe, we're also able to leverage duck typing to create composite classes that may not have a simple inheritance hierarchy.

## Managing global and singleton objects

The Python environment contains a number of implicit global objects. These objects provide a convenient way to work with a collection of other objects. Because the collection is implicit, we're saved from the annoyance of explicit initialization code.

One example of this is an implicit random number generating object in the `random` module. When we evaluate `random.random()`, we're actually making use of an instance of the `random.Random` class.

Because a module is only imported once, a module implements the **Singleton** design pattern. We can rely on this technique to implement these global singletons.

Other examples of this include the following:

- The collection of data encoders and decoders (*codecs*) available. The `codecs` module has a registry for encoders and decoders. We can add encodings and decodings to this registry.
- The `webbrowser` module has a registry of known browsers.
- The `numbers` module has a registry of numeric data types. This allows a module to

define a new implementation of a numeric type and add it to the mix of known types.

- The logging module maintains a collection of named loggers. The `getLogger()` function tries to find an existing logger; it creates a new logger if needed.
- The `re` module has a cache of compiled regular expressions. This saves the time to recompile a regular expression that's defined inside a method or function.

This recipe will show how to work with an implicit global object like the registries used for codecs, browsers, and number classes.

## Getting ready

A collection of functions can all work with an implicit global object, created by a module. The benefit is to allow other modules to share a common object without having to write any code that explicitly coordinates sharing.



This is potentially confusing to people reading your code.

The idea of shared global state can become a design nightmare. The further step of making a shared object implicit may compound the problem.

Looking at the examples from the Python standard library, there are two important patterns. First, there's a **narrow focus**. Second, the updates to the registry are limited to adding new instances.

As an example, we'll define a module with a global singleton object. We'll look more at modules in *Chapter 13*.

Our global object will be a counter that we can use to accumulate centralized data from several independent modules or objects. We'll use this global to count events in the application. The counts provide a summary of the work completed, and a check to confirm that *all* the work was completed.

The goal is to be able to write something like this:

```
for row in source:
    count('input')
    some_processing()
print(counts())
```

The `some_processing()` function might use something like `count('reject')` to count rejected input rows. This function may call other functions that also use the `count()` function to record evidence of the processing.

These two independent functions both refer to a shared global counter:

- `count(key)` increments a global Counter and returns the current value for the given key.
- `counts()` provides all of the Counter values.

## How to do it...

There are two common ways to handle global state information:

- Use a module global variable because modules are singleton objects.
- Use a class-level variable (called `static` in some programming languages). In Python, a class definition is also a singleton object that can be shared.

We'll cover these as separate mini-recipes, starting with module globals.

### Module global variables

We can do the following to create a variable that is global to a module:

1. Create a module file. This will be a `.py` file with the definitions in it. We'll call it `counter.py`.
2. If necessary, define a class for the global singleton. In our case, we can use this definition to create a `collections.Counter` object:

```
from collections import Counter
```



3. Define the one and only instance of the global singleton object. We've used a leading `_` in the name to make it slightly less visible. It's not — technically — private. It is, however, gracefully ignored by many Python tools and utilities:

```
_global_counter: Counter[str] = Counter()
```

A common idiom for marking global variables is to use an ALL\_CAPS name. This seems more important for global variables that are to be considered as constants. In this case, this variable will be updated, and an ALL\_CAPS name seems misleading.

4. Define the two functions to use the global object, `_global_counter`. These functions encapsulate the detail of how the counter is implemented:

```
def count(key: str, increment: int = 1) -> None:
    _global_counter[key] += increment

def counts() -> list[tuple[str, int]]:
    return _global_counter.most_common()
```

Now we can write applications that use the `count()` function in a variety of places. The counted events, however, are fully centralized in a single object, defined as part of the module.

We might have code that looks like this:

```
>>> from counter import *
>>> from recipe_03 import Dice1

>>> d = Dice1(1)
>>> for _ in range(1000):
...     if sum(d.roll()) == 7:
...         count('seven')
...     else:
...         count('other')
>>> print(counts())
[('other', 833), ('seven', 167)]
```

We've imported the `count()` and `counts()` functions from the `counter` module. We've also

imported the `Dice1` class as a handy object that creates a sequence of events. When we create an instance of `Dice1`, we provide an initialization to force a particular random seed. This gives repeatable results.

The benefit of this technique is that several modules can all share the global object within the counter module. All that's required is an `import` statement. No further coordination or overheads are necessary.

## Class-level “static” variables

We can do the following to create a variable that is global to all instances of a class definition:

1. Define a class with a variable outside the `__init__()` method. This variable is part of the class, not part of an instance. To make it clear that the attribute is shared by all instances of the class, the `ClassVar` type hint is helpful. In this example, we've decided to use a leading `_` so the class-level variable is not seen as part of the public interface:

```
from collections import Counter
from typing import ClassVar

class EventCounter:

    _class_counter: ClassVar[Counter[str]] = Counter()
```

2. Add methods to update and extract data from the class-level `_class_counter` attribute. These will use the `@classmethod` decorators to show they are used directly by the class, not by an instance. The `self` variable is not used; instead, a `cls` variable is used as a reminder that the method applies to the class:

```
@classmethod
def count(cls, key: str, increment: int = 1) -> None:
    cls._class_counter[key] += increment

@classmethod
def counts(cls) -> list[tuple[str, int]]:
    return cls._class_counter.most_common()
```

It's very important to note that the `_class_counter` attribute is part of the class, and is referred to as `cls._class_counter`. We don't use a `self` instance variable because we aren't referring to an instance of the class; we're referring to a variable that's part of the overall class definition, provided as the first parameter to a method decorated with `@classmethod`.

Here's how we can use this class.

```
>>> from counter import *
>>> EventCounter.count('input')

>>> EventCounter.count('input')
>>> EventCounter.count('filter')

>>> EventCounter.counts()
[('input', 2), ('filter', 1)]
```

Since all these operations update the `EventCounter` class, each increments the shared variable.



Shared global state must be used carefully.

They may be tangential to the real work of the class. The focus must be narrow and limited to very state changes.

When in doubt, explicitly shared objects will be a better design strategy, but will involve a bit more code.

## How it works...

The Python import mechanism uses `sys.modules` to track which modules are loaded. Once a module is in this mapping, it is not loaded again. This means that any variable defined within a module will be a singleton: there will only be one instance.

Within a module, a class definition can only be created once. This means the internal state changes of a class also follow the Singleton design pattern.

How can we choose between these two mechanisms? The choice is based on the degree of confusion created by having multiple classes sharing a global state. As shown in the previous example in *Class-level "static" variables*, we could have multiple variables sharing a common Counter object. If the presence of an implicitly shared global state seems confusing, then a module-level global is a better choice. In the cases where a module-level global is confusing, share state explicitly with ordinary visible variables.

## There's more...

A shared global state can be called the opposite of object-oriented programming. One ideal of object-oriented programming is to encapsulate state changes in individual objects. Used too broadly, global variables break the idea of encapsulation of state within a single object.

In a sense, a module is a kind of class-like structure. A module is a namespace with module-level variables to define state and module functions that are like methods.

One example of a need for a common global state often arises when trying to define configuration parameters for an application. It can help to have a single, uniform configuration that's shared widely throughout multiple modules. When these objects are used for pervasive features such as configuration, audits, logging, and security, globals can be helpful for segregating a single, focused cross-cutting concern into a generic class separate from application-specific classes.

An alternative is to create a configuration object explicitly. This configuration object can then be provided as a parameter to other objects throughout an application.

## See also

- *Chapter 14* covers additional topics in module and application design.

## Using more complex structures – maps of lists

In *Chapter 4*, we looked at the basic data structures available in Python. Those recipes generally looked at the various structures in isolation.

We'll look at a common combination structure—the mapping from a single key to a list of

related values. This can be used to accumulate detailed information about database or log records identified by a given key. This recipe will partition a flat list of details into lists organized by shared key values.

This blurs into class design because we can often leverage Python's built-in classes for this kind of work. This can reduce the volume of unique, new code we have to write.

## Getting ready

We'll look at a mapping from a string to a list of instances of a class we'll design. We're going to start with some raw log entries from an application, decompose each line into individual fields, and then create individual Event objects from the fields of data. Once we have these objects, we can then reorganize and regroup them into lists associated with common attribute values like module name, or message severity.

In *Chapter 5*, we looked at log data in the *Creating dictionaries – inserting and updating* recipe.

The first step will be to transform the log lines into a more useful **comma-separated value (CSV)** format. A regular expression can pick out the various syntactic groups. See the *String parsing with regular expressions* recipe in *Chapter 1* for information on how the parsing works.

The raw data looks like the following:

```
[2016-04-24 11:05:01,462] INFO in module1: Sample Message One
[2016-04-24 11:06:02,624] DEBUG in module2: Debugging
[2016-04-24 11:07:03,246] WARNING in module1: Something might have gone
wrong
```

Each row can be parsed into the component fields with a regular expression. We'll define a `NamedTuple` subclass that has a static method, `from_line()`, to create instances of the class using these four fields. Making sure the attribute names match the regular expression group names, we can build instances of the class using the following definition:

```
import re
from typing import NamedTuple

class Event(NamedTuple):
    timestamp: str
    level: str
    module: str
    message: str

    @staticmethod
    def from_line(line: str) -> 'Event | None':
        pattern = re.compile(
            r"\[(?P<timestamp>.*?)\]\s+"
            r"(?P<level>\w+)\s+"
            r"in\s+(?P<module>\w+)"
            r":\s+(?P<message>.*)"
        )
        if log_line := pattern.match(line):
            return Event(**log_line.groupdict())
        else:
            return None
```

Our objective is to group the log messages by the module name attribute. We want to see something like this:

```
>>> pprint(summary)
{'module1': [
    Event('2016-04-24 11:05:01,462', 'INFO', 'module1', 'Sample Message
    One'),
    Event('2016-04-24 11:07:03,246', 'WARNING', 'module1', 'Something might
    have gone wrong')],
'module2': [
    Event('2016-04-24 11:06:02,624', 'DEBUG', 'module2', 'Debugging')]
}
```

## How to do it...

We can write a `summarize()` function to restructure the log data as follows:

1. Import the required modules and some type hints for various kinds of collections:

```
from collections import defaultdict
from collections.abc import Iterable
```

The `defaultdict` type is a concrete class that extends the `MutableMapping` abstract base class. This is in a separate module from the `Iterable` type hint, which is an abstract base class definition.

2. The source data type, the `Event` class, was shown in the *Getting ready* section.
3. Define an overall type hint for the summary dictionary we'll be working with:

```
from typing import TypeAlias
Summary: TypeAlias = defaultdict[str, list[Event]]
```

4. Start the definition of a function to summarize an iterable source of the `Event` instances, and produce a `Summary` object:

```
def summarize(data: Iterable[Event]) -> Summary:
```

5. Use the `list` function as the default value for `defaultdict`. It's also helpful to create a type hint for this collection:

```
module_details: Summary = defaultdict(list)
```

The `list` function is provided as a name only. A common mistake using `list()` will evaluate the function and create a list object that is not a function. An error message like `TypeError: first argument must be callable or None` is a reminder that the argument must be the name of a function only.

6. Iterate through the data, appending to the list associated with each key. The `defaultdict` object will use the supplied `list()` function to build an empty list as the value corresponding to each new key the first time each key is encountered:

```
for event in data:
    module_details[event.module].append(event)
return module_details
```

The result of the `summarize()` function is a dictionary that maps from a module name string to a list of all log rows for that module name. The data will look like the following:

```
>>> pprint(summary)
defaultdict(<class 'list'>,
           {'module1': [Event(timestamp='2016-04-24 11:05:01,462',
                               level='INFO', module='module1', message='Sample Message One'),
                       Event(timestamp='2016-04-24 11:07:03,246',
                               level='WARNING', module='module1',
                               message='Something might have gone wrong')],
           'module2': [Event(timestamp='2016-04-24 11:06:02,624',
                               level='DEBUG', module='module2', message='Debugging')]}])
```

The key for this mapping is the module name and the value in the mapping is the list of rows for that module name. We can now focus the analysis on a specific module. This seems to be a close match with the initial expectations for the summarized results.

## How it works...

There are two choices for how a mapping behaves when a key is not found:

- The built-in `dict` class raises an exception when a key is missing. This makes it difficult to accumulate values associated with keys that aren't known in advance.
- The `defaultdict` class evaluates a function that creates a default value when a key is missing. In many cases, the function is `int` or `float` to create a default numeric value of 0 or 0.0. In this case, the function is `list` to create an empty list.

We can imagine using the `set` function instead of `list` to create an empty set object for a missing key. This would be suitable for a mapping from a key to a set of immutable objects that share that key.



## There's more...

We can also build a version of this as an extension to the built-in dict class:

```
class ModuleEvents(dict[str, list[Event]]):
    def __missing__(self, key: str) -> list[Event]:
        self[key] = list()
        return self[key]
```

We've provided an implementation for the special `__missing__()` method. The default behavior is to raise a `KeyError` exception. This will create a new empty list in the mapping.

This allows us to use code such as the following:

```
>>> event_iter = (Event.from_line(l) for l in log_data.splitlines())
>>> module_details = ModuleEvents()
>>> for event in filter(None, event_iter):
```

The use case is identical to the `defaultdict`, but the definition of the collection class is slightly more complicated. This permits further extension to add features to the `ModuleEvents` class.

## See also

- In the *Creating dictionaries – inserting and updating* recipe in *Chapter 4*, we looked at the basics of using a mapping.
- In the *Avoiding mutable default values for function parameters* recipe in *Chapter 4*, we looked at other places where default values are used.
- In the *Using more sophisticated collections* recipe in *Chapter 7*, we looked at other examples of using the `defaultdict` class.

## Creating a class that has orderable objects

We often need objects that can be sorted into order. Log records, to give one example, are often ordered by date and time. Most of our class definitions have not included the

features necessary for sorting objects into order. Many of the recipes have kept objects in mappings or sets based on the internal hash value computed by the `__hash__()` method, and an equality test defined by the `__eq__()` method.

In order to keep items in a sorted collection, we'll need the comparison methods that implement `<`, `>`, `<=`, and `>=`. These comparisons are all based on the attribute values of each object.

When we extend the `NamedTuple` class, the comparison methods that apply to the `tuple` class are available. If we defined class using the `@dataclass` decorator, the comparison methods are not provided by default. We can use `@dataclass(order=True)` to have the ordering methods included. For this recipe, we'll look at a class that is not based on either of these helpers.

## Getting ready

In the *Separating concerns via multiple inheritance* recipe, we defined playing cards using two class definitions. The `Card` class hierarchy defined essential features of each card. A second set of mixin classes provided game-specific features for each card.

The core definition, `Card`, was a frozen `dataclass`. It did not have the `order=True` parameter, and does not properly put cards into order. We'll need to add features to this `Card` definition to create objects that can be ordered properly.

We'll assume one more class definition, `PinochlePoints` to follow the rules for assigning points to cards for the game of Pinochle. The details don't matter; all that matters is that the class implements the `points()` method.

In order to create a sortable collection of cards, we need to add yet another feature to the family of `Card` class definitions. We'll need to define four special methods used for the comparison operators.

## How to do it...

To create an orderable class definition, we'll create a comparison protocol, and then define a class that implements the protocol, as follows:

1. We're defining a new protocol that tools like **mypy** can use when comparing objects. This will describe what kinds of objects the mixin will apply to. We've called it `CardLike` because it applies to any class with at least the two attributes of `rank` and `suit`:

```
from typing import Protocol, Any

class CardLike(Protocol):
    rank: int
    suit: str
```

This use of *something*-Like as a protocol name is part of the overall Pythonic approach of duck typing. Rather than insist on a type hierarchy, we define the fewest features required as a new Protocol.

2. Extending the protocol, we can create the `SortableCard` subclass for the comparison features. This subclass can be mixed into any class that fits the protocol definition:

```
class SortableCard(CardLike):
```

3. Add the four order comparison methods to the `SortableCard` subclass. In this case, we're using the relevant attributes of any class that fits the `CardLike` protocol into a tuple, then using Python's built-in tuple comparison to handle the details of comparing the items in the tuple. Here are the methods:

```
def __lt__(self: CardLike, other: Any) -> bool:
    return (self.rank, self.suit) < (other.rank, other.suit)
def __le__(self: CardLike, other: Any) -> bool:
    return (self.rank, self.suit) <= (other.rank, other.suit)
def __gt__(self: CardLike, other: Any) -> bool:
    return (self.rank, self.suit) > (other.rank, other.suit)
def __ge__(self: CardLike, other: Any) -> bool:
```

```
return (self.rank, self.suit) >= (other.rank, other.suit)
```

4. Write the composite class definitions, built from an essential `Card` class and two mixin classes to provide the *Pinochle* and comparison features:

```
class PinochleAce(AceCard, SortableCard, PinochlePoints):
    pass

class PinochleFace(FaceCard, SortableCard, PinochlePoints):
    pass

class PinochleNumber(Card, SortableCard, PinochlePoints):
    pass
```

5. There's no simple superclass for this collection of classes. We'll add a type hint to create a common definition:

```
from typing import TypeAlias
PinochleCard: TypeAlias = PinochleAce | PinochleFace | PinochleNumber
```

6. Now we can create a function that will create individual `PinochleCard` objects from the classes defined previously:

```
def make_pinochle_card(rank: int, suit: str) -> PinochleCard:
    if rank in (9, 10):
        return PinochleNumber(rank, suit)
    elif rank in (11, 12, 13):
        return PinochleFace(rank, suit)
    else:
        return PinochleAce(rank, suit)
```

The dauntingly complex point rules for Pinochle are encapsulated in the `PinochlePoints` class. We've omitted them because the points don't parallel the six card ranks at all. Building composite classes as a base subclass of `Card` plus `PinochlePoints` leads to an accurate model of the cards without too much overt complexity.

We can now make cards that respond to comparison operators, using the following sequence

of interactive commands:

```
>>> c1 = make_pinochle_card(9, '♥')
>>> c2 = make_pinochle_card(10, '♥')
>>> c1 < c2
True

>>> c1 == c1 # Cards match themselves
True

>>> c1 == c2
False

>>> c1 > c2
False
```

The equality comparisons to implement `==` and `!=` are defined in the base class, `Card`. This is a frozen data class. By default, data classes contain equality test methods.

Here's a function that builds the special 48-card deck. It creates two copies of each of the 24 different rank and suit combinations:

```
def make_pinochle_deck() -> list[PinochleCard]:
    return [
        make_pinochle_card(r, s)
        for _ in range(2)
        for r in range(9, 15)
        for s in SUITS
    ]
```

The value of the `SUITS` variable is the four Unicode characters for the suits. The generator expression inside the `make_deck()` function builds two copies of each card using the 6 ranks that are part of the Pinochle game.

## How it works...

Python uses special methods for a vast number of things. Almost every operator in the language is implemented by a special method. (The few exceptions are the `is` operator and `and`, `or`, and `not`.) In this recipe, we've leveraged the four ordering operators.

The expression `c1 <= c2` is evaluated as if we'd written `c1.__le__(c2)`. This kind of transformation happens for almost all Python operators.

The *Python Language Reference* organizes the special methods into several distinct groups. In this recipe, we've looked at methods used for basic customization of a class.

Here's how it looks when we work with instances of this class hierarchy. The first example will create a 48-card *Pinochle* deck:

```
>>> deck = make_pinochle_deck()
>>> len(deck)
48
```

```
>>> import random
>>> random.seed(4)
>>> random.shuffle(deck)
>>> [str(c) for c in sorted(deck[:12])]
['9 ♣', '10 ♣', 'J ♠', 'J ♠', 'J ♠', 'Q ♠', 'Q ♣', 'K ♠', 'K ♠',
 'K ♣', 'A ♥', 'A ♣']
```

The important part of the above example is the use of the `sorted()` function. Because we've defined proper comparison operators, we can sort the `PinochleCard` instances, and they are presented in the expected order from low rank to high rank.

## There's more...

A little formal logic suggests that we really only need to implement two of the comparisons in detail. From an equality method and one ordering method, all the remaining methods can be built. For example, if we build the operations for less than (`__lt__()`) and equal to (`__eq__()`), we could compute the other three comparisons following these equivalence rules:

- $a \leq b \equiv a < b \vee a = b$
- $a \geq b \equiv b < a \vee a = b$
- $a \neq b \equiv \neg(a = b)$

Python emphatically does not do any of this kind of advanced algebra for us. We need to do the algebra carefully and implement the necessary comparison methods.

The `functools` library includes a decorator, `@total_ordering`, that can generate these missing comparison methods.

## See also

- See the *Separating concerns via multiple inheritance* recipe for the essential definitions of cards and card game rules.
- See *Chapter 7* for more information on dataclasses and named tuple classes.

## Deleting from a list of complicated objects

Removing items from a list has an interesting consequence. Specifically, when an item is removed, all the subsequent items move forward. The rule is this:

On deleting item `y`, items `list[y+1:]` take the place of items `list[y:]`.

This is a side-effect that happens in addition to removing the selected item. Because things can move around in a list, it makes deleting more than one item at a time potentially challenging.

When the list contains items that have a definition for the `__eq__()` special method, then the list `remove()` method can remove each item. When the list items don't have a simple `__eq__()` test, then the `remove()` method doesn't work, making it more challenging to remove multiple items from the list.

## Getting ready

For this example, we'll work with a list of dictionaries, where a naïve approach to removing items doesn't work out. It's helpful to see what can go wrong with trying repeatedly to search a list for an item to delete.

In this case, we've got some data that includes a song name, the writers, and a duration. The dictionary objects are rather long. The data looks like this:

```
>>> song_list = [
... {'title': 'Eruption', 'writer': ['Emerson'], 'time': '2:43'},
... {'title': 'Stones of Years', 'writer': ['Emerson', 'Lake'], 'time':
'3:43'},
... {'title': 'Iconoclast', 'writer': ['Emerson'], 'time': '1:16'},
... {'title': 'Mass', 'writer': ['Emerson', 'Lake'], 'time': '3:09'},
... {'title': 'Manticore', 'writer': ['Emerson'], 'time': '1:49'},
... {'title': 'Battlefield', 'writer': ['Lake'], 'time': '3:57'},
... {'title': 'Aquatarkus', 'writer': ['Emerson'], 'time': '3:54'}
... ]
```

The type hint for each row of this complex structure can be defined with as follows:

```
from typing import TypedDict

class SongType(TypedDict):
    title: str
    writer: list[str]
    time: str
```



A better design would use a `datetime.timedelta` for the song's time. We've omitted this complication from the recipe.

The list of songs as a whole can be described as `list[SongType]`.

Here's a naïve approach that emphatically does not work:

```
def naive_delete(data: list[SongType], writer: str) -> None:
    for index in range(len(data)):
        if 'Lake' in data[index]['writer']:
            del data[index]
```

Because items are moved, the computed index values will skip over the item just after one that is deleted. Further, because the list gets shorter, the range is wrong after a deletion. This fails with an index error, as shown in the following output:



```
>>> naive_delete(song_list, 'Lake')
Traceback (most recent call last):
...
IndexError: list index out of range
```

Another failing approach looks like this:

```
>>> remove = list(filter(lambda x: 'Lake' in x['writer'], song_list))
>>> for x in remove:
...     song_list.remove(x)
```

This suffers from the problem that each `remove()` operation must search the list from the beginning. This approach will be slow for a very large list.

We need to combine search and remove operations in a way that avoids multiple passes through the list.

## How to do it...

To efficiently delete multiple items from a list, we'll need to implement our own list index processing function as follows:

1. Define a function to update a list object by removing selected items:

```
def incremental_delete(
    data: list[SongType],
    writer: str
) -> None:
```

2. Initialize an index value, `i`, to zero to begin with the first item in the list:

```
i = 0
```

3. While the value for the `i` variable is not equal to the length of the list, we want to make a state change to either increment the `i` value or shrink the list:

```
while i != len(data):
```

4. If the `data[i]` value is the searched-for target, we can remove it, shrinking the list.

Otherwise, increment the index value, `i`, one step closer to the length of the list:

```
if 'Lake' in data[i]['writer']:
    del data[i]
else:
    i += 1
```

This leads to the expected behavior of removing the items from the list without suffering from index errors, making multiple passes through the list items, or failing to delete matching items.

## How it works...

The goal is to examine each item exactly once and either remove it or step over it, leaving it in place. The `while` statement design stems from looking at statements that advance toward the goal: increment the index, and delete an item. Each of these works in a limited set of conditions:

- Incrementing the index only works if the item should not be removed.
- Deleting an item only works if the item is a match.

What's important is that the conditions are exclusive. When we use a `for` statement, the increment processing always happens, an undesirable feature. The `while` statement permits us to increment *only* when the item should be left in place.

## There's more...

An alternative to the overhead of removal is to create a new list with some items rejected. Making a shallow copy of items is much faster than removing items from a list, but uses more storage. This is a common example of the time versus memory trade-off.

We can use a list comprehension like the following one to create a new list of only the

desired items:

```
>>> [item
...     for item in song_list
...         if 'Lake' not in item['writer']
... ]
```

This will create a shallow copy of selected items from the list. The items we don't want to keep will be ignored. For more information on the idea of a shallow copy, see the *Making shallow and deep copies of objects* recipe in *Chapter 4*.

We can also use a higher-order function, `filter()`, as part of the copy operation. Consider this example:

```
>>> list(
...     filter(
...         lambda item: 'Lake' not in item['writer'],
...         song_list
...     )
... )
```

The `filter()` function has two arguments: a lambda object and the original set of data. In this case, the lambda expression is used to decide which items to pass. Items for which the lambda returns `False` are rejected.

The `filter()` function is a generator. This means that we need to collect all of the items to create a final list object. The `list()` function is one way to consume all items from a generator, stashing them in the collection object they create and return.

## See also

- We've leveraged two other recipes: *Making shallow and deep copies of objects* and *Slicing and dicing a list* in *Chapter 4*.
- We'll look closely at filters and generator expressions in *Chapter 9*.

## Join our community Discord space

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

<https://packt.link/dHrHU>





# 9

## Functional Programming Features

The idea of **functional programming** is to focus on writing small, expressive functions that perform the required data transformations. Combinations of functions can often create code that is more succinct and expressive than long strings of procedural statements or the methods of complex, stateful objects. This chapter focuses on functional programming features of Python more than procedural or object-oriented programming.

This provides an avenue for software design distinct from the strictly object-oriented approach used elsewhere in this book. The combination of objects with functions permits flexibility in assembling an optimal collection of components.

Conventional mathematics defines many things as functions. Multiple functions can be combined to build up a complex result from previous transformations. When we think of mathematical operators as functions, an expression like  $p = f(n, g(n))$  can also be written as two separate functions. We might think of this as  $p = f(n, b)$ , where  $b = g(n)$ .

Ideally, we can also create a composite function from these two functions:

$$p = f(n, g(n)) = g \circ f(n)$$

Defining a new composite function,  $g \circ f$ , instead of nested functions can help to clarify the intent behind a design. This re-framing of the components can allow us to take a number of small details and combine them into a larger chunk of knowledge that embodies the concept behind the design.

Since programming often works with collections of data, we'll often be applying a function to all the items of a collection. This happens when doing database extraction and transformation to align data from diverse source applications. It also happens when summarizing data. Something as commonplace as transforming a CSV file into a statistical summary is a composition of transformation functions from rows of text to rows of data, and from rows of data to a mean and standard deviation. This fits nicely with the mathematical idea of a **set builder** or **set comprehension**.

There are three common patterns for applying one function to a set of data:

- **Mapping:** This applies a function to all the elements of a collection,  $\{m(x)|x \in S\}$ . We apply some function,  $m(x)$ , to each item,  $x$ , of a larger collection,  $S$ .
- **Filtering:** This uses a function to select elements from a collection,  $\{x|x \in S \text{ if } f(x)\}$ . We use a function,  $f(x)$ , to determine whether to pass or reject each item,  $x$ , from the larger collection,  $S$ .
- **Reducing:** This summarizes the items of a collection. One of the most common reductions is creating a sum of all items in a collection,  $S$ , written as  $\sum_{x \in S} x$ . Other common reductions include finding the smallest item, the largest one, and the product of all items.

We'll often combine these patterns to create more complex composite applications. What's important here is that small functions, such as  $m(x)$  and  $f(x)$ , can be combined via the built-in higher-order functions such as `map()`, `filter()`, and `reduce()`. The `itertools` module

contains many additional higher-order functions that we can use to build an application. And, of course, we can define our own higher-order functions to combine smaller functions. Some of these recipes will show computations that could also be defined as properties of a class definition created using the `@property` decorator. This is yet another design alternative that can limit the complexity of stateful objects. In this chapter, however, we'll try to stick to a functional approach, that is, transformation to create new objects rather than using properties.

In this chapter, we'll look at the following recipes:

- *Writing generator functions with the yield statement*
- *Applying transformations to a collection*
- *Using stacked generator expressions*
- *Picking a subset – three ways to filter*
- *Summarizing a collection – how to reduce*
- *Combining the map and reduce transformations*
- *Implementing “there exists” processing*
- *Creating a partial function*
- *Writing recursive generator functions with the yield from statement*

We'll start with a recipe where we will create functions that yield an iterable sequence of values. Rather than creating an entire list (or set, or some other collection), a generator function yields the individual items of a collection as demanded by a client operation. This saves memory and may save time.

## Writing generator functions with the yield statement

A generator function is often designed to apply some kind of transformation to each item of a collection. Generators can create data, too. A generator is called *lazy* because the values



it yields must be consumed by a client; values are not computed until a client attempts to consume them. Client operations like the `list()` function or a `for` statement are common examples of consumers. Each time a function like `list()` demands a value, the generator function must yield a value using the `yield` statement.

In contrast, an ordinary function can be called *eager*. Without the `yield` statement, a function will compute the entire result and return it via the `return` statement.

A lazy approach is very helpful in cases where we can't fit an entire collection in memory. For example, analyzing gigantic web log files can be done in small doses rather than by creating a vast in-memory collection.

In the language of Python's type hints, we'll often use the `Iterator` generic to describe generators. We'll need to clarify this generic with a type, like `Iterator[str]`, to show that the function yields string objects.

The items that are being consumed by a generator will often be from a collection described by the `Iterable` generic type. All of Python's built-in collections are `Iterable`, as are files. A list of string values, for example, can be viewed as an `Iterable[str]`.

Both the `Iterable` and `Iterator` types are available from the `collections.abc` module. They can also be imported from the `typing` module.



The `yield` statement is what changes an ordinary function into a generator. It will compute and yield results iteratively.

## Getting ready

We'll apply a generator to some web log data. We'll design a generator that will transform raw text into more useful structured objects. The generator function serves to isolate transformation processing. This permits flexibility in applying filter or summary operations after the initial transformation.

The entries start out as lines of text that look like this:

```
[2016-06-15 17:57:54,715] INFO in ch10_r10: Sample Message One
[2016-06-15 17:57:54,716] DEBUG in ch10_r10: Debugging
[2016-06-15 17:57:54,720] WARNING in ch10_r10: Something might have gone
wrong
```

We've seen other examples of working with this kind of log in the *Using more complex structures – maps of lists* recipe in *Chapter 8*. Using REs from the *String parsing with regular expressions* recipe in *Chapter 1*, we can decompose each line into a more useful structure.

It's often helpful to capture the details of each line of the log in an object of a distinct type. This helps make the code more focused, and it helps us use the **mypy** tool to confirm that types are used properly. Here's a `NamedTuple` class definition:

```
from typing import NamedTuple

class RawLog(NamedTuple):
    date: str
    level: str
    module: str
    message: str
```

We'll start with the transformation of an iterable source of strings string into an iterator over tuple of fields. After that, we'll apply the recipe again to transform the date attribute from a string into a useful datetime object.

## How to do it...

A generator function is a function, so the recipe is similar to those shown in *Chapter 3*. We'll start by defining the function, as follows:

1. Import the needed type hints from the `collections.abc` module. Import the re module to parse the line of the log file:

```
import re
from collections.abc import Iterable, Iterator
```

2. Define a function that iterates over RawLog objects. It seems helpful to include `_iter` in the function name to emphasize that the result is an iterator, not a single value. The parameter is an iterable source of log lines:

```
def parse_line_iter(
    source: Iterable[str]
) -> Iterator[RawLog]:
```

3. The `parse_line_iter()` transformation function relies on a regular expression to decompose each line. We can define this inside the function to keep it tightly bound with the rest of the processing:

```
pattern = re.compile(
    r"\[(?P<date>.*?)\]\s+"
    r"(?P<level>\w+)\s+"
    r"in\s+(?P<module>.+)"
    r":\s+(?P<message>.+)",
    re.X
)
```

4. A `for` statement will consume each line of the iterable source, allowing us to create and then yield each RawLog object in isolation:

```
for line in source:
```

5. The body of the `for` statement can map each string instance that matches the pattern to a new RawLog object using the match groups:

```
if match := pattern.match(line):
    yield RawLog(*match.groups())
```

Non-matching lines will be silently dropped. For the most part, this seems sensible because a log can be filled with messages from a variety of sources.



Without a `yield` statement, a function is “ordinary” and computes a single result.

Here’s how we use this function to emit a sequence of `RawLog` instances from the sample data shown above:

```
>>> from pprint import pprint

>>> for item in parse_line_iter(log_lines):
...     pprint(item)
RawLog(date='2016-04-24 11:05:01,462', level='INFO', module='module1',
message='Sample Message One')
RawLog(date='2016-04-24 11:06:02,624', level='DEBUG', module='module2',
message='Debugging')
RawLog(date='2016-04-24 11:07:03,246', level='WARNING', module='module1',
message='Something might have gone wrong')
```

We could also collect items into a list object using something like this:

```
>>> details = list(parse_line_iter(log_lines))
```

In this example, the `list()` function consumes all of the items produced by the `parse_line_iter()` function. A generator is a relatively passive construct: until data is demanded, it doesn’t do any work.

## How it works...

Each of Python’s built-in collection types implements a special method, `__iter__()`, to produce an iterator object. An iterator object implements the `__next__()` special method to both return an item and advance the state of the iterator to return the next item. This is the Iterator protocol. The built-in `next()` function evaluates this method of an iterator object.

While the Python built-in collections can create Iterator objects, a generator function also implements this protocol. A generator will return itself in response to the `iter()`

function. In response to the `next()` function, a generator suspends execution at a `yield` statement, and provides a value that becomes the result of the `next()` function. Since the function is suspended, it can be resumed when another `next()` function is evaluated.

To see how the `yield` statement works, look at this small function, which yields two objects:

```
test_example_4_3 = """
>>> def gen_func():
...     print("pre-yield")
...     yield 1
...     print("post-yield")
```

Here's what happens when we evaluate the `next()` function on this generator:

```
>>> y = gen_func()
>>> next(y)
pre-yield
1

>>> next(y)
post-yield
```

The first time we evaluated the `next()` function, the first `print()` function was evaluated, and then the `yield` statement produced a value.

The use of the `next()` function resumed processing, and the statements between the two `yield` statements were evaluated.

What happens next? Since there are no more `yield` statements in the function's body, so we observe the following:

```
>>> next(y)
Traceback (most recent call last):
...

```

The `StopIteration` exception is raised at the end of a generator function. This is expected

by the processing of a for statement. It is quietly absorbed to break from processing.

If we don't use a function like `list()` or a for statement to consume the data, we'll see something like this:

```
>>> parse_line_iter(data)
<generator object parse_line_iter at ...>
```

The value returned by evaluating the `parse_line_iter()` function is a generator. It's not a collection of items, but an object that will produce items, one at a time, on demand from a consumer.

## There's more...

We can apply this recipe to convert the date attributes in each `RawLog` object. The more refined kind of data from each line will follow this class definition:

```
import datetime
from typing import NamedTuple

class DatedLog(NamedTuple):
    date: datetime.datetime
    level: str
    module: str
    message: str
```

This has a more useful `datetime.datetime` object for the timestamp. The other fields remain as strings.

Here's a generator function – using a for statement and `yield` so that it's an iterator – that's used to refine each `RawLog` object into a `DatedLog` object:

```
def parse_date_iter(
    source: Iterable[RawLog]
) -> Iterator[DatedLog]:
    for item in source:
        date = datetime.datetime.strptime(
```

```

        item.date, "%Y-%m-%d %H:%M:%S,%f"
    )
    yield DatedLog(
        date, item.level, item.module, item.message
    )

```

Breaking overall processing into small generator functions confers several significant advantages. First, the decomposition makes each function more succinct because it is focused on a specific task. This makes these functions easier to design, test, and maintain. Second, it makes the overall composition somewhat more expressive of the work being done.

We can combine these two generators in the following kind of composition:

```

>>> for item in parse_date_iter(parse_line_iter(log_lines)):
...     print(item)
DatedLog(date=datetime.datetime(2016, 4, 24, 11, 5, 1, 462000),
level='INFO', module='module1', message='Sample Message One')
DatedLog(date=datetime.datetime(2016, 4, 24, 11, 6, 2, 624000),
level='DEBUG', module='module2', message='Debugging')
DatedLog(date=datetime.datetime(2016, 4, 24, 11, 7, 3, 246000),
level='WARNING', module='module1', message='Something might have gone
wrong')

```

The `parse_line_iter()` function will consume lines from the source data, creating `RawLog` objects when they are demanded by a consumer. The `parse_date_iter()` function is a consumer of `RawLog` objects; from these, it creates `DatedLog` objects when demanded by a consumer. The outer `for` statement is the ultimate consumer, demanding `DatedLog` objects.

At no time will there be a large collection of intermediate objects in memory. Each of these functions works with a single object, limiting the amount of memory used.

## See also

- In the *Using stacked generator expressions* recipe, we'll combine generator functions to build complex processing stacks from simple components.

- In the *Applying transformations to a collection* recipe, we'll see how the built-in `map()` function can be used to create complex processing from a simple function and an iterable source of data.
- In the *Picking a subset – three ways to filter* recipe, we'll see how the built-in `filter()` function can also be used to build complex processing from a simple function and an iterable source of data.

## Applying transformations to a collection

We often define generator functions with the intention apply the function to a collection of data items. There are a number of ways that generators can be used with collections.

In the *Writing generator functions with the yield statement* recipe in this chapter, we created a generator function to transform data from a string into a more complex object.

Generator functions have a common structure, and generally look like this:

```
def new_item_iter(source: Iterable[X]) -> Iterator[Y]:  
    for item in source:  
        new_item: Y = some_transformation(item)  
        yield new_item
```

The `yield` statement means the results will be generated iteratively. The function's type hints emphasize that it consumes items from the source collection. This template for writing a generator function exposes a common design pattern.

Mathematically, we can summarize this as follows:

$$N = \{m(x) \mid x \in S\}$$

The new collection,  $N$ , is a transformation,  $m(x)$ , applied to each item,  $x$ , of the source,  $S$ . This emphasizes the transformation function,  $m(x)$ , separating it from the details of consuming the source and producing the result. In the Python example previously shown, this function was called `some_transformation()`.



This mathematical summary suggests that the `for` statement can be understood as a kind of scaffold around the transformation function. There are two additional forms this scaffolding can take. We can write a generator expression or we can use the built-in `map()` function. This recipe will examine all three techniques.

## Getting ready

We'll look at the web log data from the *Writing generator functions with the yield statement* recipe. This had dates as strings that we would like to transform into a proper datetime object to be used for further computations. We'll make use of the `DatedLog` class definition from that earlier recipe.

The *Writing generator functions with the yield statement* recipe used a generator function like the following example to transform a sequence of `RawLog` objects into an iterator of more useful `DatedLog` instances:

```
import datetime
from recipe_01 import RawLog, DatedLog

def parse_date_iter(
    source: Iterable[RawLog]
) -> Iterator[DatedLog]:
    for item in source:
        date = datetime.datetime.strptime(
            item.date, "%Y-%m-%d %H:%M:%S,%f"
        )
        yield DatedLog(
            date, item.level, item.module, item.message
        )
```

This `parse_date_iter()` function has a significant amount of scaffolding code around an interesting function. The `for` and `yield` statements are examples of scaffolding. The date parsing, on the other hand, is the distinctive, interesting part of the function. We need to extract this distinct processing to permit use of more flexible scaffolding.

## How to do it...

To make use of different approaches to applying a generator function, we'll need to start by refactoring the original `parse_date_iter()` function. This will extract a `parse_date()` function that can be used in a variety of ways. After this initial step, we'll show three separate mini-recipes for using the refactored code.

Refactor the iterator to define a function that can be applied to a single row of the data. It should produce an item of the result type from an item of the source type:

```
def parse_date(item: RawLog) -> DatedLog:
    date = datetime.datetime.strptime(
        item.date, "%Y-%m-%d %H:%M:%S,%f")
    return DatedLog(
        date, item.level, item.module, item.message)
```

This transformation can be applied to a collection of data in three ways: a generator function, a generator expression, and via the `map()` function. We'll start by rebuilding the original generator.

## Using the for and yield statements

We can apply a single-row `parse_date()` transformation function to each item of a collection using the `for` and `yield` statements. This was shown in the *Writing generator functions with the yield statement* recipe earlier in this chapter. Here's what it looks like:

```
def parse_date_iter_y(
    source: Iterable[RawLog]
) -> Iterator[DatedLog]:
    for item in source:
        yield parse_date(item)
```

## Using a generator expression

We can apply the `parse_date()` function to each item of a collection using a generator expression. A generator expression includes two parts – the mapping function, and a `for` clause – enclosed by `()`. This follows the pattern of the *Building lists – literals, appending,*

and comprehensions recipe in Chapter 4:

1. Write the `()` brackets that surround the generator.
2. Write a `for` clause for the source of the data, assigning each item to a variable, in this case, `item`:

```
(... for item in source)
```

3. Prefix the `for` clause with the mapping function, applied to the variable:

```
(parse_date(item) for item in source)
```

4. The expression can be the return value from a function that provides suitable type hints for the source and the resulting expression. Here's the entire function, since it's so small:

```
def parse_date_iter_g(
    source: Iterable[RawLog]
) -> Iterator[DatedLog]:
    return (parse_date(item) for item in source)
```

The function returns the generator expression that applies the `parse_date()` function to each item in the source iterable.

Yes, this function is so small, it doesn't seem to require the overhead of a `def` statement and a name. The type hints can be helpful in some contexts, making this a sensible choice.

## Using the `map()` function

We can apply the `parse_date()` function to each item of a collection using the `map()` built-in function:

1. Use the `map()` function to apply the transformation to the source data:

```
map(parse_date, source)
```

2. The expression can be the return value from a function that provides suitable type

hints for the source and the resulting expression. Here's the entire function, since it's so small:

```
def parse_date_iter_m(
    source: Iterable[RawLog]
) -> Iterator[DatedLog]:
    return map(parse_date, source)
```

The `map()` function is an iterator that applies the `parse_date()` function to each item from the source iterable. It yields the objects created by the `parse_date()` function.



It's important to note that the `parse_date` name without `()` is a reference to a function object.

It's a common error to think the function must be evaluated, and include extra, unnecessary uses of `()`.

All three techniques are equivalent.

## How it works...

The `map()` function replaces some common code that acts as a scaffold around the processing. It does the work of a `for` statement. It applies the given function to each item in the source iterable.

We can define our own version of `map()` as follows:

```
def my_map2(f: Callable[[P], Q], source: Iterable[P]) -> Iterator[Q]:
    return (f(item) for item in source)
```

As we've seen, these are identical in their behavior. Different audiences for the code may have distinct preferences. The guidance we offer is to choose the style that makes the meaning and intention the most clear to the audience reading the code.

## There's more...

In this example, we've used the `map()` function to apply a function that takes a single parameter to each item of a single iterable collection. It turns out that the `map()` function can do a bit more than this. The `map()` function can process several sequences.

Consider this function and these two sources of data:

```
>>> def mul(a, b):  
...     return a * b  
  
>>> list_1 = [2, 3, 5, 7]  
>>> list_2 = [11, 13, 17, 23]
```

We can apply the `mul()` function to the sequence of pairs drawn from each source of data:

```
>>> list(map(mul, list_1, list_2))  
[22, 39, 85, 161]
```

This allows us to merge several sequences of values using different kinds of operations on argument values pulled from the sequences.

## See also

- In the *Using stacked generator expressions* recipe later in this chapter, we will look at stacked generators. We will build a composite function from a number of individual mapping operations, written as various kinds of generator functions.

## Using stacked generator expressions

In the *Writing generator functions with the yield statement* recipe earlier in this chapter, we created a simple generator function that performed a single transformation on a piece of data. As a practical matter, we often have several functions that we'd like to apply to incoming data.

How can we stack or combine multiple generator functions to create a composite function?

## Getting ready

This recipe will apply several different kinds of transformations to source data. There will be restructuring of the rows to combine three rows into a single row, data conversions to convert the source strings into useful numbers or datetime stamps, and filtering to reject rows that aren't useful.

We have a spreadsheet that is used to record fuel consumption on a large sailboat.

For details of this data, see the *Slicing and dicing a list* recipe in *Chapter 4*. We'll look at parsing this in more detail in the *Reading delimited files with the CSV module* recipe in *Chapter 11*.

We'd like to apply a number of transformations to each row-level list of this list-of-lists-of-strings object:

- Exclude the three lines of headers (and any blank lines) that are present in the data.
- Merge three physical lines of text strings into one logical row of data.
- Convert the separated date and time strings into datetime objects.
- Convert the fuel height from a string to a float, ideally in gallons (or liters) instead of inches.

Our goal is to create a group of generator functions. Assuming we have assigned the results of a generator function to a variable, `datetime_gen`, the transformations allow us to have software that looks like this:

```
>>> total_time = datetime.timedelta(0)
>>> total_fuel = 0
>>> for row in datetime_gen:
...     total_time += row.engine_off - row.engine_on
...     total_fuel += (
...         float(row.engine_on_fuel_height) -
...         float(row.engine_off_fuel_height)
...     )
>>> print(
```

```
... f"{total_time.total_seconds()/60/60 = :.2f}, "  
... f"{total_fuel = :.2f}")
```

We need to design a composite function that can create this `datetime_gen` generator.

## How to do it...

We'll decompose this into three separate mini-recipes:

- Restructuring the rows.
- Excluding the header row.
- Creating more useful row objects.

We'll start with restructuring three physical lines into a logical row.

### Restructuring the rows

We'll start by creating a `row_merge()` function to restructure the data:

1. We'll use a named tuple to define a type for the combined logical rows:

```
from typing import NamedTuple  
  
class CombinedRow(NamedTuple):  
    # Line 1  
    date: str  
    engine_on_time: str  
    engine_on_fuel_height: str  
    # Line 2  
    filler_1: str  
    engine_off_time: str  
    engine_off_fuel_height: str  
    # Line 3  
    filler_2: str  
    other_notes: str  
    filler_3: str
```

The raw data has empty cells; we've called them `filler_1`, `filler_2`, and `filler_3`. Preserving these junk columns can make it easier to debug problems.

2. The source rows created by a CSV reader will have a `list[str]` type; we'll provide an alias for this type, `RawRow`. The function's definition will accept an iterable of `RawRow` instances. It is an iterator over `CombinedRow` objects:

```
from typing import TypeAlias
from collections.abc import Iterable, Iterator

RawRow: TypeAlias = list[str]

def row_merge(
    source: Iterable[RawRow]
) -> Iterator[CombinedRow]:
```

3. The body of the function will consume rows from the source iterator, skipping empty lines, building a cluster that defines a `CombinedRow` object. When the first column is non-empty, any previous cluster is complete, it is yielded, and a new cluster is started. The very last cluster also needs to be yielded:

```
cluster: RawRow = []
for row in source:
    if all(len(col) == 0 for col in row):
        continue
    elif len(row[0]) != 0:
        # Non-empty column 1: line 1
        if len(cluster) == 9:
            yield CombinedRow(*cluster)
            cluster = row.copy()
        else:
            # Empty column 1: line 2 or line 3
            cluster.extend(row)
if len(cluster) == 9:
    yield CombinedRow(*cluster)
```

This initial transformation can be used to convert a sequence of lines of CSV cell values into `CombinedRow` objects where each of the field values from three separate rows have their own unique attributes.

The first row output from this transformation will be a header row. The next part is a



function to drop this row.

## Excluding the header row

The first three lines of text from the source CSV file will create a `CombinedRow` object that's not very useful. We'll exclude a row with labels instead of data:

1. Define a function to work with an iterable collection of `CombinedRow` objects, creating an iterator of `CombinedRow` objects:

```
def skip_header_date(
    source: Iterable[CombinedRow]
) -> Iterator[CombinedRow]:
```

2. The function's body consumes each row of the source and yields the good rows. It uses a `continue` statement to reject the undesirable rows:

```
for row in source:
    if row.date == "date":
        continue
    yield row
```

This can be combined with the `row_merge()` function shown in the previous recipe to provide an iterator over good data.

There are several transformation steps required to make the merged data truly useful. Next, we'll look at one of these, creating proper `datetime.datetime` objects.

## Creating more useful row objects

The dates and times in each row aren't very useful as separate strings. The function we'll write can have a slightly different form than the previous two steps in this recipe because it applies to each row in isolation. The single-row transformation looks like this:

1. Define a new `NamedTuple` class that specifies a more useful type for the time values:

```
import datetime
from typing import NamedTuple

class DatetimeRow(NamedTuple):
    date: datetime.date
    engine_on: datetime.datetime
    engine_on_fuel_height: str
    engine_off: datetime.datetime
    engine_off_fuel_height: str
    other_notes: str
```

2. Define a mapping function to convert one CombinedRow instance into a single DatetimeRow instance:

```
def convert_datetime(row: CombinedRow) -> DatetimeRow:
```

3. The body of this function will perform a number of date-time computations and create a new DatetimeRow instance:

```
    travel_date = datetime.datetime.strptime(
        row.date, "%m/%d/%y").date()
    start_time = datetime.datetime.strptime(
        row.engine_on_time, "%I:%M:%S %p").time()
    start_datetime = datetime.datetime.combine(
        travel_date, start_time)
    end_time = datetime.datetime.strptime(
        row.engine_off_time, "%I:%M:%S %p").time()
    end_datetime = datetime.datetime.combine(
        travel_date, end_time)

    return DatetimeRow(
        date=travel_date,
        engine_on=start_datetime,
        engine_off=end_datetime,
        engine_on_fuel_height=row.engine_on_fuel_height,
        engine_off_fuel_height=row.engine_off_fuel_height,
        other_notes=row.other_notes
    )
```

We can now stack the transformation functions to merge rows, exclude the header, and

perform date time conversions. The processing looks like this:

```
>>> row_gen = row_merge(log_rows)
>>> tail_gen = skip_header_date(row_gen)
>>> datetime_gen = (convert_datetime(row) for row in tail_gen)
```

We've decomposed the reformatting, filtering, and transformation problems into three separate functions. Each of these three steps does a small part of the overall job. We can test each of the three functions separately. More important than being able to test is being able to fix or revise one step without completely breaking the entire stack of transformations.

## How it works...

When we write a generator function, the argument value can be a collection of items, or it can be any other kind of iterable source of items. Since generator functions are iterators, it becomes possible to create a *pipeline* of generator functions by stacking them. The results of one generator are the input to the next one in the stack.

The `datetime_gen` object created by this recipe is a composition of three separate generators. A `for` statement can gather values from the `datetime_gen` generator expression. The body of that statement can print details and compute summaries of the objects being generated.

This design emphasizes small, incremental operations at each stage. Some stages of the pipeline will consume multiple source rows for a single result row, restructuring the data as it is processed. Other stages consume and transform a single row, making them amenable to being described by generator expressions.

The entire pipeline is driven by demand from the client. Note that there's no concurrency in this processing. Each function is "suspended" at the `yield` statement until a client demands more data from it via the built-in `next()` function.

Most importantly, the individual transformation steps can be debugged and tested separately. This decomposition can help to create more robust and reliable software.

## There's more...

There are a number of other conversions required to make this data useful. We'll want to transform the start and end timestamps into a duration. We also need to transform the fuel height values into floating-point numbers instead of strings.

We have a number of ways to handle these derived data computations:

1. We can create additional transformation steps in our stack of generator functions. This reflects an eager computation approach.
2. We can also add `@property` methods to the class definition. This is lazy computation; it's only performed when the property value is required.

To compute additional fuel height and volume values eagerly, we can apply the design pattern again. First, define additional named tuple classes with the required fields. Then, define a transformation function to convert height from a string to a float. Also, define a transformation to convert height from inches to gallons. These additional functions will be small and easy to test.

We now have a sophisticated computation that's defined in a number of small and (almost) completely independent chunks. Each function does the work required to create only one row, keeping the overheads to a minimum. We can modify one piece without having to think deeply about how the other pieces work.

## See also

- See the *Writing generator functions with the yield statement* recipe for an introduction to generator functions.
- See the *Slicing and dicing a list* recipe in *Chapter 4*, for more information on the fuel consumption dataset.
- See the *Combining the map and reduce transformations* recipe for another way to combine operations.
- The *Picking a subset – three ways to filter* recipe covers the filter function in more

detail.

## Picking a subset – three ways to filter

Choosing a subset of relevant rows can be termed filtering a collection of data. We can view a filter as rejecting bad rows or including the desirable rows. There are several ways to apply a filtering function to a collection of data items.

In the *Using stacked generator expressions* recipe, we wrote the `skip_header_date()` generator function to exclude some rows from a set of data. The `skip_header_date()` function combined two elements: a rule to pass or reject items, and a source of data. This generator function had a general pattern that looks like this:

```
from collections.abc import Iterable, Iterator
from typing import TypeVar

T = TypeVar("T")
```

```
def data_filter_iter(
    source: Iterable[T]
) -> Iterator[T]:
    for item in source:
        if should_be_passed(item):
            yield item
```

This `data_filter_iter()` function's type hints emphasize that it is an iterable that consumes items of a type, `T`, from an iterable source collection. Some expression is applied to each item to determine if it's valid. This expression can be defined as a separate function. We can define filters of considerable sophistication.

The design pattern can be summarized as follows:

$$N = \{x \mid x \in S \text{ if } f(x)\}$$

The new collection,  $N$ , is each item,  $x$ , in the source,  $S$ , where a filter function,  $f(x)$ , is

true. This summary emphasizes the filter function,  $f(x)$ , separating it from minor technical details of consuming the source and producing the result.

This mathematical summary suggests the `for` statement is little more than scaffolding code. Because it's less important than the filter rule, it can help to refactor a generator function and extract the filter from the other processing.

Considering the `for` statement as scaffolding, how else can we apply a filter to each item of a collection? There are two additional techniques we can use:

- We can write a generator expression.
- We can use the built-in `filter()` function.

Both of these require refactoring the generator function — `skip_header_date()`, shown earlier in the *Using stacked generator expressions* recipe — to extract the decision-making expression, separate from the `for` and `if` scaffolding around it. From this function, we can then move to creating a generator expression, and using the `filter()` function.

## Getting ready

In this recipe, we'll look at the fuel consumption data from the *Using stacked generator expressions* recipe in this chapter. For details of this data, see the *Slicing and dicing a list* recipe in *Chapter 4*.

We used two generator functions. The first, `row_merge()`, restructured the physical lines into logical rows. A named tuple, `CombinedRow`, was used to provide a more useful structure to the row data. The second generator function, `skip_header_date()`, rejected the heading rows of the data, passing the useful data rows.

We'll rewrite the `skip_header_date()` function to demonstrate three distinct approaches to extracting useful data.

## How to do it...

The first part of this recipe will refactor the “good data” rule out of the generator function to make it more widely useful.

1. Start with a draft version of a generator function with the following outline:

```
def skip_header_date(
    source: Iterable[CombinedRow]
) -> Iterator[CombinedRow]:
    for row in source:
        if row.date == "date":
            continue
        yield row
```

2. The expression in the `if` statement can be refactored into a function that can be applied to a single row of the data, producing a `bool` value:

```
def pass_non_date(row: CombinedRow) -> bool:
    return row.date != "date"
```

3. The original generator function can now be simplified:

```
def skip_header_date_iter(
    source: Iterable[CombinedRow]
) -> Iterator[CombinedRow]:
    for item in source:
        if pass_non_date(item):
            yield item
```

The `pass_non_date()` function can be used in three ways. As shown here, it can be used by a generator function. It can also be used in a generator expression, and with the `filter()` function. Next, we'll look at writing an expression.

## Using a filter in a generator expression

A generator expression includes three parts – the item, a `for` clause, and an `if` clause – all enclosed by `()`:

1. Start with a `for` clause that assigns objects to a variable. This source comes from some iterable collection, which is called `source` in this example:

```
(... for item in source)
```

2. Because this is a filter, the result expression should be the variable from the for clause:

```
(item for item in source)
```

3. Write an if clause using the filter rule function, `pass_non_date()`:

```
(item for item in source if pass_non_date(source))
```

4. This generator expression can be the return value from a function that provides suitable type hints for the source and the resulting expression. Here's the entire function, since it's so small:

```
def skip_header_gen(
    source: Iterable[CombinedRow]
) -> Iterator[CombinedRow]:
    return (
        item
        for item in source
        if pass_non_date(item)
    )
```

This function returns the result of the generator expression. The function doesn't do very much, but it does applying a name and a set of type hints to the expression.

The `skip_header_gen()` function uses a generator expression that applies the `pass_non_date()` function to each item in the source collection to determine whether it passes and is kept, or whether it is rejected.

The results are identical to the original `skip_header_date()` function, shown above.

## Using the `filter()` function

Using the `filter()` function includes two parts – the decision function and the source of data – as arguments:

1. Use the `filter()` function to apply the function to the source data:



```
filter(pass_non_date, source)
```

The `filter()` function is an iterator that applies the given function, `pass_non_date()` as a rule to pass or reject each item from the given iterable, `data`. It yields the rows for which the `pass_non_date()` function returns `True`.



It's important to note that the `pass_non_date` name without `()` is a reference to a function object.

It's a common error to think the function must be evaluated, and include extra, unnecessary uses of `()`.

## How it works...

A generator expression must include a `for` clause to provide a source of data items. The optional `if` clause can apply a condition that preserves some items while rejecting others. Placing a filter condition in an `if` clause can make the expression clear and expressive of the algorithm.

Generator expressions have an important limitation. As expressions, they cannot use statement-oriented features of Python. The `try-except` statement, used to handle exceptional data conditions, is often helpful.

## There's more...

Sometimes, it's difficult to write a simple rule to define the valid data or reject the invalid data. In many cases, it may be impossible to use a simple string comparison to identify rows to reject. This happens when a file is filled with extraneous information; manually prepared spreadsheets suffer from this. In some cases, there's no trivial regular expression that helps to characterize valid data.

We can often encounter data where the easiest way to establish validity is to attempt a conversion, and transform the presence or absence of an exception into a Boolean condition.

Consider the following function to ascertain if a row of data has a valid date:

```
import datetime

def row_has_date(row: CombinedRow) -> bool:
    try:
        datetime.datetime.strptime(row.date, "%m/%d/%y")
        return True
    except ValueError as ex:
        return False
```

This will attempt a conversion of a date. It will reject invalid strings of characters that fail to follow the essential format rule. It will also reject 2/31/24; while the string of digits is valid, this is not a real date.

## See also

- In the *Using stacked generator expressions* recipe earlier in this chapter, we placed a function like this in a stack of generators. We built a composite function from a number of individual mapping and filtering operations written as generator functions.

## Summarizing a collection – how to reduce

A reduction is the generalized concept behind computing a summmary like the total or the maximum of a collection of numbers. Computing statistical measures like mean or variance are also reductions. In this recipe, we'll look at several summarization or reduction techniques.

In the introduction to this chapter, we noted that there are three processing patterns that Python supports elegantly: map, filter, and reduce. We saw examples of mapping in the *Applying transformations to a collection* recipe and examples of filtering in the *Picking a subset – three ways to filter* recipe.

The third common pattern is reduction. In the *Designing classes with lots of processing* and the *Extending a built-in collection – a list that does statistics* recipes, we looked at class definitions that computed a number of statistical values. These definitions relied – almost

exclusively — on the built-in `sum()` function. This is one of the more common reduce operations.

In this recipe, we'll look at a way to generalize summation, leading to ways to write a number of different kinds of reductions that are similar. Generalizing the concept of reduction will let us build on a reliable foundation to create more sophisticated algorithms.

## Getting ready

Some of the most common reduce operations are the sum, minimum, maximum. These reductions are so common, they're built in. The average and variance, on the other hand, are reductions defined in the `statistics` module. The `math` module has a variant on `sum`, `fsum()`, that works particularly well for collections of floating-point values.

Summations are the backbone of financial reporting. It is the essence of what a spreadsheet has been used for since the days of doing financial reports using pen and paper.

The mathematics of summation help us to see how an operator is used to convert a collection of values into a single value. Here's a way to think of the mathematical definition of the sum function using an operator, `+`, applied to the values in a collection,  $C = \{c_0, c_1, c_2, \dots, c_n\}$ :

$$\sum_{c_i \in C} c_i = c_0 + c_1 + c_2 + \dots + c_n + 0$$

We've expanded the definition of sum by *folding* the `+` operator into the sequence of values in  $C$ .

Folding involves two items: a binary operator and a base value. For sum, the operator was `+` and the base value was zero. For product, the operator is `×` and the base value is one. The base value needs to be the identity element for the given operator.

We can apply this concept to many algorithms, potentially simplifying the definition. In this recipe, we'll define a product function. This is the  $\prod$  operator, similar to the  $\sum$  operator.

## How to do it...

Here's how we define a reduction that implements product of a collection of numbers:

1. Import the `reduce()` function from the `functools` module:

```
from functools import reduce
```

2. Pick the operator. For sum, it's `+`. For product, it will be `*`. These can be defined in a variety of ways. Here's the long version. Other ways to define the necessary binary operators will be shown later:

```
def mul(a: int, b: int) -> int:  
    return a * b
```

3. Pick the base value required. The additive identity value for sum is zero. The multiplicative identity for a product is one:

```
def prod(values: Iterable[float]) -> float:  
    return reduce(mul, values, 1)
```

```
def prod(values: Iterable[int]) -> int:  
    return reduce(mul, values, 1)
```

We can use this `prod()` function to define other functions. One example is the factorial function. It looks like this:

```
def factorial(n: int) -> int:  
    return prod(range(1, n+1))
```

How many six-card cribbage hands are possible? The binomial calculation uses the factorial function to compute the number of ways 6 cards can be extracted from a 52 card deck:

$$\binom{52}{6} = \frac{52!}{6!(52-6)!}$$

Here's a Python implementation:

```
>>> factorial(52) // (factorial(6) * factorial(52 - 6))
20358520
```

For any given shuffle, there are about 20 million different possible cribbage hands we might see.

## How it works...

The `reduce()` function behaves as though it has this definition:

```
T = TypeVar("T")

def my_reduce(
    fn: Callable[[T, T], T],
    source: Iterable[T],
    initial: T | None = None
) -> T:
```

The type hints shows how there has to be a unifying type, `T`, that applies to the operator being folded, and the initial value for the folding. The given function, `fn()`, must combine two values of type `T` and return another value of the same type `T`. The result of the `reduce()` function will be a value of this type also.

Furthermore, in Python, the `reduce` operation will iterate through the values from left to right. It will apply the given binary function, `fn()`, between the previous result and the next item from the source collection. This additional detail is important when thinking about non-commutative operators like subtraction or division.

## There's more...

We'll look at three additional topics. First, ways to define the operation. After that, we'll look at applying `reduce` to Boolean values in *Logical reductions: any and all*. Finally, in *Identity elements*, we'll look at the identity elements used by various operators.

## Operation definition

When designing a new application for the `reduce()` function, we need to provide a binary operator. There are three ways to define the necessary binary operator. First, we can use a complete function definition, as shown above in the recipe. There are two other choices. We can use a lambda object instead of a complete function:

```
from collections.abc import Callable

lmul: Callable[[int, int], int] = lambda a, b: a * b
```

A lambda object is an anonymous function boiled down to just two essential elements: the parameters and the return expression. There are no statements inside a lambda, only a single expression.

We assigned the lambda object to a variable, `lmul`, so that we can use the expression `lmul(2, 3)` to apply the lambda object to argument values.

When the operation is one of Python's built-in operators, we have another choice – import the definition from the operator module:

```
from itertools import takewhile
```

This works nicely for all the built-in arithmetic operators.

It's essential to consider the complexity of the operator being used to reduce. Performing a reduce operation increases an operation's complexity by a factor of  $n$ . An operation that's  $O(1)$  becomes  $O(n)$  when applied to  $n$  items in a collection. For the operators we've shown, like `add` and `mul`, this fits our expectations. An operator with more complexity than  $O(1)$  can turn into a performance nightmare.

In the next section, we'll look at the logical reduction functions.

## Logical reductions: any and all

Conceptually, it seems like we should be able to do `reduce()` operations using the Boolean operators `and` and `or`. It turns out this involves some additional consideration.

Python's Boolean operators have a *short-circuit* feature: when we evaluate the expression `False and 3 / 0`, the result is only `False`. The expression on the right-hand side of the `and` operator, `3 / 0`, is never evaluated. The `or` operator is similar: when the left side is `True`, the right-hand side is never evaluated.

If we want to be sure that a sequence of `bool` values is all true, building our own `reduce()` will do far too much work. Once the initial `False` is seen, there's no reason to process the remaining items. The *short-circuit* feature of `and` and `or` does not *not* fit with the `reduce()` function.

The built-in functions `any()` and `all()`, on the other hand, are reductions using logic operators. The `any()` function is, effectively, a kind of `reduce()` using the `or` operator. Similarly, the `all()` function behaves as if it's a `reduce()` with the `and` operator.

## Identity elements

Generally, an operator used for a reduction must have an identity element. This is provided to the `reduce()` function as the initial value. The identity element will also be the result when they're applied against an empty sequence. Here are some common examples:

- `sum([])` is zero.
- `math.prod([])` is one.
- `any([])` is `False`.
- `all([])` is `True`.

The identity value for the given operation is a matter of definition.

In the case of `any()` and `all()` specifically, it can help to think of the fundamental *fold* operation. The identity element can always be folded in without changing the result. Here's how `all()` would look with explicitly folded `and` operators:

$b_0$  **and**  $b_1$  **and**  $b_2$  **and** ... **and**  $b_n$  **and** True

If all of the values  $b_0, b_1, b_2, \dots, b_n$  are True, then the additional **and** True doesn't change the value. If any of the values  $b_0, b_1, b_2, \dots, b_n$  are False, similarly, the additional **and** True has no impact.

When there are no values in a collection, the value for `all()` is the identity element, True.

## See also

- See the *Using stacked generator expressions* recipe in this chapter for a context in which `sum` can be applied to compute total hours and total fuel.

## Combining the map and reduce transformations

In the other recipes in this chapter, we've been looking at `map`, `filter`, and `reduce` operations. We've looked at each of these functions in isolation:

- The *Applying transformations to a collection* recipe shows the `map()` function.
- The *Picking a subset – three ways to filter* recipe shows the `filter()` function.
- The *Summarizing a collection – how to reduce* recipe shows the `reduce()` function.

Many algorithms will involve creating composite functions that combine these more basic operations. Additionally, we'll need to look at a profound limitation of working with iterators and generator functions.

Here's an example of this limitation:

```
>>> typical_iterator = iter([0, 1, 2, 3, 4])
>>> sum(typical_iterator)
10

>>> sum(typical_iterator)
0
```



We created an iterator over a sequence of values by manually applying the `iter()` function to a literal list object. The first time that the `sum()` function consumed the values from `typical_iterator`, it consumed all five values. The next time we try to apply any function to `typical_iterator`, there will be no more values to be consumed; the iterator will appear empty. By definition, the identity value — 0 for summation — is the result.



An iterator can only produce values once.

After the values have been consumed, an iterator appears to be an empty collection.

This one-time-only constraint will force us to cache intermediate results when we need to perform multiple reductions on the data. Creating intermediate collection objects will consume memory, leading to the need for a careful design when working with very large collections of data. (Processing large collections of data is difficult. Python offers some ways to create workable solutions; it does not magically make the problem evaporate.)

To apply a complex transformation of a collection, we often find instances of `map`, `filter`, and `reduce` operations that can be implemented separately. These can then be combined into sophisticated composite operations.

## Getting ready

In the *Using stacked generator expressions* recipe earlier in this chapter, we looked at some sailboat data. The spreadsheet was badly organized, and a number of steps were required to impose a more useful structure on the data.

In that recipe, we looked at a spreadsheet that is used to record fuel consumption on a large sailboat. For details of this data, see the *Slicing and dicing a list* recipe in *Chapter 4*. We'll look at parsing this in more detail in the *Reading delimited files with the CSV module* recipe in *Chapter 11*.

The initial processing in the *Using stacked generator expressions* recipe created a sequence of operations to change the organization of the data, filter out the headings, and compute

some useful values. We'd need to supplement this with two more reductions to get some average and variance information. These statistics will help us understand the data more fully. We'll build on that earlier processing with some additional steps.

## How to do it...

We'll start with a target line of code as the design goal. In this case, we'd like a function to sum the fuel use per hour. This follows a common three-step processing pattern. First, we normalize the data with `row_merge()`. Second, we use mapping and filtering to create more useful objects with `clean_data_iter()`.

The third step should look like the following:

```
>>> round(
...     total_fuel(clean_data_iter(row_merge(log_rows))),
...     3
... )
7.0
```

Our target function, `total_fuel()`, is designed to work with a few other functions that clean and organize the raw data. We'll start with the normalization and proceed to defining the final summary function as follows:

1. Import the functions from earlier recipes to reuse the initial preparation:

```
from recipe_03 import row_merge, CombinedRow
```

2. Define the target data structure created by the cleaning and enrichment step. We'll use a mutable dataclass in this example. The fields coming from the normalized `CombinedRow` object can be initialized directly. The other five fields will be computed eagerly by several separate functions. Fields not computed in the `__init__()` method must be given an initial value of `field(init=False)`:

```
import datetime
from dataclasses import dataclass, field
```

```

@dataclass
class Leg:
    date: str
    start_time: str
    start_fuel_height: str
    end_time: str
    end_fuel_height: str
    other_notes: str
    start_timestamp: datetime.datetime = field(init=False)
    end_timestamp: datetime.datetime = field(init=False)
    travel_hours: float = field(init=False)
    fuel_change: float = field(init=False)
    fuel_per_hour: float = field(init=False)

```

3. Define the overall data cleansing and enrichment data function. This will build the enriched Leg objects from the source CombinedRow objects. We'll build this from seven simpler functions. The implementation is a stack of map() and filter() operations that will derive data from the source fields:

```

from collections.abc import Iterable, Iterator

def clean_data_iter(
    source: Iterable[CombinedRow]
) -> Iterator[Leg]:
    leg_iter = map(make_Leg, source)
    fitered_source = filter(reject_date_header, leg_iter)
    start_iter = map(start_datetime, fitered_source)
    end_iter = map(end_datetime, start_iter)
    delta_iter = map(duration, end_iter)
    fuel_iter = map(fuel_use, delta_iter)
    per_hour_iter = map(fuel_per_hour, fuel_iter)
    return per_hour_iter

```

Each statement makes use of the iterator produced by the preceding statement.

4. Write the make\_Leg() function to create Leg instances from CombinedRow instances:

```

def make_Leg(row: CombinedRow) -> Leg:
    return Leg(
        date=row.date,

```

```
        start_time=row.engine_on_time,  
        start_fuel_height=row.engine_on_fuel_height,  
        end_time=row.engine_off_time,  
        end_fuel_height=row.engine_off_fuel_height,  
        other_notes=row.other_notes,  
    )
```

5. Write the `reject_date_header()` function used by `filter()` to remove the heading rows:

```
def reject_date_header(row: Leg) -> bool:  
    return not (row.date == "date")
```

6. Write the data conversion functions. We'll start with the two date and time strings, which need to become a single `datetime` object:

```
def timestamp(  
    date_text: str, time_text: str  
) -> datetime.datetime:  
    date = datetime.datetime.strptime(  
        date_text, "%m/%d/%y").date()  
    time = datetime.datetime.strptime(  
        time_text, "%I:%M:%S %p").time()  
    timestamp = datetime.datetime.combine(  
        date, time)  
    return timestamp
```

7. Mutate the `Leg` instances with additional values:

```
def start_datetime(row: Leg) -> Leg:  
    row.start_timestamp = timestamp(  
        row.date, row.start_time)  
    return row  
  
def end_datetime(row: Leg) -> Leg:  
    row.end_timestamp = timestamp(  
        row.date, row.end_time)  
    return row
```

This update-in-place approach is an optimization to avoid creating intermediate objects.

8. Compute the derived duration from the timestamps:

```
def duration(row: Leg) -> Leg:
    travel_time = row.end_timestamp - row.start_timestamp
    row.travel_hours = round(
        travel_time.total_seconds() / 60 / 60,
        1
    )
    return row
```

9. Compute any other metrics that are needed for the analysis:

```
def fuel_use(row: Leg) -> Leg:
    end_height = float(row.end_fuel_height)
    start_height = float(row.start_fuel_height)
    row.fuel_change = start_height - end_height
    return row

def fuel_per_hour(row: Leg) -> Leg:
    row.fuel_per_hour = row.fuel_change / row.travel_hours
    return row
```

The final `fuel_per_hour()` function's calculation depends on the entire preceding stack of calculations. Each of these computations is done separately to clarify and isolate the computation details. This approach permits changes to the isolated computations. Most importantly, it permits testing each computation as a separate unit.

## How it works...

The core concept is to build a composite transformation from a sequence of small steps. Since each step is conceptually distinct, it makes it somewhat easier to understand the composition.

In this recipe, we've used three kinds of transformations:

- **Structural changes.** An initial generator function grouped physical lines into logical

rows.

- **Filters.** A generator function rejected rows that were invalid.
- **Enrichment.** As we've seen, there are two design approaches to enriching data: lazy and eager. The lazy approach may involve methods or properties, computed only as needed. This design shows eager computation, where a number of fields had values built by the processing pipeline.

The various enrichment methods worked by updating stateful Leg objects, setting computed column values. Using stateful objects like this requires the various enrichment transformations be performed in a strict order because some (like `duration()`) depend on others having been performed first.

We can now design the target computation functions:

```
from statistics import *

def avg_fuel_per_hour(source: Iterable[Leg]) -> float:
    return mean(row.fuel_per_hour for row in source)

def stdev_fuel_per_hour(source: Iterable[Leg]) -> float:
    return stdev(row.fuel_per_hour for row in source)
```

This meets our design goal of being able to perform meaningful computations on the raw data.

## There's more...

As we noted, we can only perform one iteration of consuming the items from an iterable source of data. If we want to compute several averages, or the average as well as the variance, we'll need to use a slightly different design pattern.

In order to compute multiple summaries of the data, it's often best to create a concrete object of some kind that can be summarized repeatedly:

```
def summary(raw_data: Iterable[list[str]]) -> None:
    data = tuple(clean_data_iter(row_merge(raw_data)))
    m = avg_fuel_per_hour(data)
    s = 2 * stdev_fuel_per_hour(data)
    print(f"Fuel use {m:.2f} ±{s:.2f}")
```

Here, we've created a very large tuple from the cleaned and enriched data. From this tuple, we can produce any number of iterators. This lets us compute any number of distinct summaries.

We can also use the `tee()` function in the `itertools` module for this kind of processing. This can devolve to inefficient processing because of the way the cloned instances of the iterator maintain their internal state. It's often better to create an intermediate structure like a list or tuple than to use `itertools.tee()`.

The design pattern applies a number of transformations to the source data. We've built it using a stack of separate `map`, `filter`, and `reduce` operations.

## See also

- See the *Using stacked generator expressions* recipe in this chapter for a context in which `sum` can be applied to compute total hours and total fuel.
- See the *Summarizing a collection – how to reduce* recipe in this chapter for some background on the `reduce()` function.
- See Python High Performance for more on distributed map-reduce processing.
- We look at lazy properties in the *Using properties for lazy attributes* recipe in *Chapter 7*. Also, this recipe looks at some important variations of map-reduce processing.

## Implementing “there exists” processing

The processing patterns we've been looking at can all be summarized with the universal quantifier,  $\forall$ , meaning *for all*. It's been an implicit part of all of the processing definitions:

- **Map:** For **all** items in the source,  $S$ , apply the map function,  $m(x)$ . We can use the

universal quantifier:  $\forall_{x \in S} m(x)$ .

- **Filter:** This, also, means for **all** items in the source,  $S$ , pass those for which the filter function,  $f(x)$ , is true. Here, also, we can use the universal quantifier:  $\forall_{x \in S} x$  **if**  $f(x)$ .
- **Reduce:** For all items in the source, use the given operator and base value to compute a summary. The universal quantification is implicit in the definition of operators

$$\sum_{x \in S} x \text{ and } \prod_{x \in S} x.$$

Contrast these universal functions with the cases where we are only interested in locating a single item. We often describe these cases as a **search** to show there exists at least one item where a condition is true. This can be described with the existential quantifier,  $\exists$ , meaning *there exists*.

We'll need to use some additional features of Python to create generator functions that stop when the first value matches some predicate. We'd like to emulate the short-circuit capabilities of the built-in `any()` and `all()` functions.

## Getting ready

For an example of an existence test, consider a function to determine if a number is prime or composite. A prime number has no factors other than 1 and itself. Numbers with multiple factors are called composite. The number 42 is composite because it has the numbers 2, 3, and 7 as prime factors.

Finding if a number is prime number is the same as showing it is not composite. For any composite (or non-prime) number,  $n$ , the rule is this:

$$\neg P(n) = \exists_{2 \leq i < n} (n \equiv 0 \pmod{i})$$

A number,  $n$ , is not prime if there exists a value,  $i$ , between 2 and the number itself, that divides the number evenly. For a test to see if a number is prime, we don't need to know **all** the factors. The existence of a single factor shows the number is composite.

The overall idea is to iterate over the range of candidate numbers, breaking from the



iteration when a factor is found. In Python, this early exit from a `for` statement is done with the `break` statement, shifting the semantics from “for all” to “there exists.” Because `break` is a statement, we can’t easily use a generator expression; we’re constrained to writing a generator function.

(The Fermat test is generally more efficient than what we’re using for these examples, but it doesn’t involve a simple search for the existence of a factor. We’re using this as an illustration of search, not as an illustration of good primality tests.)

## How to do it...

In order to build this kind of search function, we’ll need to create a generator function that will complete processing when it finds the first match. One way to do this is with the `break` statement, as follows:

1. Define a generator function to skip items until a test is passed. The generator can yield the first value that passes the predicate test. The generator works by applying a predicate function, `fn()`, to the items in a sequence of items of some type, `T`:

```
from collections.abc import Callable, Iterable, Iterator
from typing import TypeVar

T = TypeVar("T")

def find_first(
    fn: Callable[[T], bool], source: Iterable[T]
) -> Iterator[T]:
    for item in source:
        if fn(item):
            yield item
            break
```

2. Define the specific predicate function for this application. Since we’re testing for being prime, we’re looking for any value that divides the target number, `n`, evenly. Here’s the kind of expression that’s needed:

```
lambda i: n % i == 0
```

3. Apply the `find_first()` search function with the given range of values and predicate. If the factors iterable has an item, then  $n$  is composite. Otherwise, there are no values in the factors iterable, which means  $n$  is a prime number:

```
import math

def prime(n: int) -> bool:
    factors = find_first(
        lambda i: n % i == 0,
        range(2, int(math.sqrt(n) + 1)) )
    return len(list(factors)) == 0
```

As a practical matter, we don't need to test every number between two and  $n$  to see whether  $n$  is prime. It's only necessary to test values,  $i$ , such that  $2 \leq i < \lfloor \sqrt{n} \rfloor$ .

## How it works...

In the `find_first()` function, we introduce a `break` statement to stop processing the source iterable. When the `for` statement stops, the generator will reach the end of the function and return normally.

A client function consuming values from this generator will be given the `StopIteration` exception. The `find_first()` function can raise an exception, but it's not an error; it's the signal that an iterable has finished processing the input values.

In this case, the `StopIteration` exception means one of two things:

- If a value had been yielded previously, the value is a factor of  $n$ .
- If no value was yielded, then  $n$  is prime.

This small change of breaking early from the `for` statement makes a dramatic difference in the meaning of the generator function. Instead of processing **all** values from the source, the `find_first()` generator will stop processing as soon as the predicate is true.

## There's more...

In the `itertools` module, there is an alternative to the `find_first()` function. The `takewhile()` function uses a predicate function to take values from the input while the predicate function is true. When the predicate becomes false, then the function stops consuming and producing values.

To use the `takewhile()` function, we need to invert our factor test. We need to consume values that are non-factors until we find the first factor. This leads to a change in the lambda from `lambda i: n % i == 0` to `lambda i: n % i != 0`.

Let's look at a test to see if 47 is prime. We need to check numbers in the range 2 to  $\sqrt{49} = 7$ :

```
>>> from itertools import takewhile

>>> n = 47
>>> list(takewhile(lambda i: n % i != 0, range(2, 8)))
[2, 3, 4, 5, 6, 7]
```

For a prime number, like 47, none of the test values are factors. All these non-factor test values pass the `takewhile()` predicate because it's always true. The resulting list is the same as the original set of test values.

For a composite number, the non-factor test values will be a subset of the test values. Some values will have been excluded because a factor was found.

There are a number of additional functions in the `itertools` module that can be used to simplify complex map-reduce applications. We encourage you to look closely at this module.

## See also

- In the *Using stacked generator expressions* recipe earlier in this chapter, we made extensive use of immutable class definitions.
- See <https://projecteuler.net/problem=10> for a challenging problem related to

prime numbers less than 2 million. Parts of the problem seem obvious. It can be difficult, however, to test all those numbers for being prime.

- The `itertools` module provides numerous functions that can simplify functional design.
- Outside the standard library, packages like `Pysistent` offer functional programming components.

## Creating a partial function

When we look at functions such as `reduce()`, `sorted()`, `min()`, and `max()`, we see that we'll often have some argument values that change very rarely, if at all. In a particular context, they're essentially fixed. For example, we might find a need to write something like this in several places:

```
reduce(operator.mul, ..., 1)
```

Of the three argument values for `reduce()`, only one – the iterable to process – actually changes. The operator and the initial value argument values are essentially fixed at `operator.mul` and `1`.

Clearly, we can define a whole new function for this:

```
from collections.abc import Iterable
from functools import reduce
import operator

def prod(iterable: Iterable[float]) -> float:
    return reduce(operator.mul, iterable, 1)
```

Python has a few ways to simplify this pattern so we don't have to repeat the boilerplate `def` and `return` statements.

The goal of this recipe is different from providing general default values. A partial function doesn't provide a way for us to override the defaults. A partial function has specific values

bound when it is defined. The idea is to be able to create many partial functions, each with specific argument values bound in advance. This is also sometimes called a *closure*, but applied to some of the parameters. See *Picking an order for parameters based on partial functions* in *Chapter 3* for more examples of partial function definition.

## Getting ready

Some statistical modeling is done with standardized values, sometimes called **z-scores**. The idea is to standardize a raw measurement onto a value that can be easily compared to a normal distribution, and easily compared to related numbers that may be measured in different units.

The calculation is this:

$$z = \frac{x - \mu}{\sigma}$$

Here,  $x$  is a raw value,  $\mu$  is the population mean, and  $\sigma$  is the population standard deviation. The value  $z$  will have a mean of 0 and a standard deviation of 1, providing a standardized value. We can use this value to spot **outliers** – values that are suspiciously far from the mean. We expect that (approximately) 99.7% of our  $z$  values will be between -3 and +3.

We could define a function to compute standard scores, like this:

```
def standardize(mean: float, stdev: float, x: float) -> float:
    return (x - mean) / stdev
```

This `standardize()` function will compute a z-score from a raw score,  $x$ . When we use this function in a practical context, we'll see that there are two kinds of argument values for the parameters:

- The argument values for the `mean` and `stdev` parameters are essentially fixed. Once we've computed the population values, we'll have to provide the same two values to the `standardize()` function over and over again.

- The value for the `x` parameter will vary each time we evaluate the `standardize()` function.

Let's work with a collection of data samples with two variables, `x` and `y`. These pairs are defined by the `DataPair` class:

```
from dataclasses import dataclass

@dataclass
class DataPair:
    x: float
    y: float
```

As an example, we'll compute a standardized value for the `x` attribute. This means computing the mean and standard deviation for the `x` values. Then, we'll need to apply the mean and standard deviation values to standardize the data in our collection. The computation looks like this:

```
>>> import statistics
>>> mean_x = statistics.mean(item.x for item in data_1)
>>> stdev_x = statistics.stdev(item.x for item in data_1)
>>> for DataPair in data_1:
...     z_x = standardize(mean_x, stdev_x, DataPair.x)
...     print(DataPair, z_x)
```

Providing the `mean_x`, and `stdev_x` values each time we evaluate the `standardize()` function can clutter an algorithm with details that aren't deeply important.

We can use a partial function to simplify this use of `standardize()` with two fixed argument values and one that is left to vary.

## How to do it...

To simplify using a function with a number of fixed argument values, we can create a partial function. This recipe will show two ways to create a partial functions as separate mini-recipes:

- Using the `partial()` function from the `functools` module to build a new function from the full `standardize()` function
- Creating a lambda object to supply argument values that don't change

## Using `functools.partial()`

1. Import the `partial()` function from the `functools` module:

```
from functools import partial
```

2. Create a new function using `partial()`. We provide the base function, plus the positional arguments that need to be included. Any parameters that are not supplied when the partial is defined must be supplied when the partial is evaluated:

```
z = partial(standardize, mean_x, stdev_x)
```

We've provided fixed values for the first two parameters, `mean` and `stdev`, of the `standardize()` function. We can now use the `z()` function with a single value, `z(a)`, and it will evaluate the expression `standardize(mean_x, stdev_x, a)`.

## Creating a lambda object

1. Define a lambda object that binds the fixed parameters:

```
lambda x: standardize(mean_x, stdev_x, x)
```

2. Assign this lambda to a variable to create a callable object, `z()`:

```
z = lambda x: standardize(mean_x, stdev_x, x)
```

This provides fixed values for the first two parameters, `mean` and `stdev`, of the `standardize()` function. We can now use the `z()` lambda object with a single value, `z(a)`, and it will evaluate the expression `standardize(mean_x, stdev_x, a)`.

## How it works...

Both techniques create a callable object – a function – named `z()` that has the values for `mean_x` and `stdev_x` already bound to the first two positional parameters. With either approach, we can now have processing that can look like this:

```
for DataPair in data_1:
    print(DataPair, z(DataPair.x))
```

We've applied the `z()` function to each set of data. Because `z()` is a partial function and has some parameters already applied, its use is simplified.

There's one significant difference between the two techniques for creating the `z()` function:

- The `partial()` function binds the actual values of the parameters. Any subsequent change to the variables that were used *will not* change the definition of the partial function that's created.
- The lambda object binds the variable name, not the value. Any subsequent change to the variable's value *will* change the way the lambda behaves.

We can modify the lambda slightly to bind specific values instead of names:

```
z = lambda x, m=mean_x, s=stdev_x: standardize(m, s, x)
```

This extracts the current values of `mean_x` and `stdev_x` to create default values for the lambda object's parameters. The values of `mean_x` and `stdev_x` are now irrelevant to the proper operation of the lambda object, `z()`.

## There's more...

We can provide keyword argument values as well as positional argument values when creating a partial function. While this works nicely in general, there are a few cases where it doesn't work.

We started this recipe looking at the `reduce()` function. Interestingly, this function is one



example of functions that can't be trivially turned into a partial function. The parameters aren't in the ideal order for creating a partial and it doesn't permit providing argument values by name.

It appears as though the `reduce()` function is defined like this:

```
def reduce(function, iterable, initializer=None)
```

If this were the actual definition, we could do this:

```
prod = partial(reduce(mul, initializer=1))
```

Practically, the preceding example raises a `TypeError`. It doesn't work because the definition of `reduce()` does not take keyword argument values. Consequently, we can't easily create partial functions that use it.

This means that we're forced to use the following lambda technique:

```
>>> from operator import mul
>>> from functools import reduce
>>> prod = lambda x: reduce(mul, x, 1)
```

In Python, a function is an object. We've seen numerous ways that functions can be arguments to other functions. A function that accepts or returns another function as an argument is sometimes called a **higher-order function**.

Similarly, functions can also return a function object as a result. This means that we can create a function like this:

```
from collections.abc import Sequence, Callable
import statistics

def prepare_z(data: Sequence[DataPair]) -> Callable[[float], float]:
    mean_x = statistics.mean(item.x for item in data_1)
    stdev_x = statistics.stdev(item.x for item in data_1)
```

```
return partial(standardize, mean_x, stdev_x)
```

Here, we've defined a function over a sequence of type `DataPair`, which are  $(x, y)$  samples. We've computed the mean and standard deviation of the  $x$  attribute of each sample. We then created a partial function that can standardize scores based on the computed statistics. The result of this function is a function we can use for data analysis.

The following example shows how this newly created function is used:

```
>>> z = prepare_z(data_1)
>>> for DataPair in data_1:
...     print(DataPair, z(DataPair.x))
```

The result of the `prepare_z()` function is a callable object that will standardize a score based on the computed mean and standard deviation.

## See also

- See *Picking an order for parameters based on partial functions* in *Chapter 3* for more examples of partial function definition.

## Writing recursive generator functions with the `yield` from statement

Many algorithms can be expressed neatly as recursions. In the *Designing recursive functions around Python's stack limits* recipe, we looked at some recursive functions that could be optimized to reduce the number of function calls.

When we look at some data structures, we see that they involve recursion. In particular, JSON documents (as well as XML and HTML documents) can have a recursive structure. A JSON document is a complex object that can contain other complex objects within it.

In many cases, there are advantages to using generators for processing these kinds of structures. In this recipe, we'll look at ways to handle recursive data structures with

generator functions.

## Getting ready

In this recipe, we'll look at a way to search for all matching values in a complex, recursive data structure. When working with complex JSON documents, they often contain dict-of-dict, dict-of-list, list-of-dict, and list-of-list structures. Of course, a JSON document is not limited to two levels; dict-of-dict can really mean dict-of-dict-of.... Similarly, dict-of-list could mean dict-of-list-of.... The search algorithm must descend through the entire structure looking for a particular key or value.

A document with a complex structure might look like this:

```
document = {
  "field": "value1",
  "field2": "value",
  "array": [
    {"array_item_key1": "value"},
    {"array_item_key2": "array_item_value2"}
  ],
  "object": {
    "attribute1": "value",
    "attribute2": "value2"
  },
}
```

The value "value" can be found in three places:

- ["array", 0, "array\_item\_key1"]: This path starts with the top-level field named array, then visits item zero of a list, then a field named array\_item\_key1.
- ["field2"]: This path has just a single field name where the value is found.
- ["object", "attribute1"]: This path starts with the top-level field named object, then the child, attribute1, of that field.

A `find_value()` function should yield all these paths when it searches the overall document for the target value. The core algorithm for this is a depth-first search. The output from this

function must be a list of paths that identify the target value. Each path will be a sequence of field names or field names mixed with index positions.

## How to do it...

We'll start with an overview of the depth-first algorithm to visit all of the nodes in a JSON document:

1. Start with a sketch of the function to process each of the alternative structures in the overall data structure. Here are the imports and some type hints:

```
from collections.abc import Iterator
from typing import Any, TypeAlias

JSON_DOC: TypeAlias = (
    None | str | int | float | bool | dict[str, Any] | list[Any]
)
Node_Id: TypeAlias = Any
```

Here is the sketch of the function:

```
def find_value_sketch(
    value: Any,
    node: JSON_DOC,
    path: list[Node_Id] | None = None
) -> Iterator[list[Node_Id]]:
    if path is None:
        path = []
    match node:
        case dict() as dnode:
            pass # apply find_value to each key in dnode
        case list() as lnode:
            pass # apply find_value to each item in lnode
        case _ as pnode: # str, int, float, bool, None
            if pnode == value:
                yield path
```

2. Here's a starting version to look at each key of a dictionary. This replaces the `# apply find_value to each key in dnode` line in the preceding code. Test this to be sure the recursion works properly:

```

for key in sorted(dnode.keys()):
    for match in find_value_y(value, dnode[key], path + [key]):
        yield match

```

3. Replace the inner for with a yield from statement:

```

for key in sorted(dnode.keys()):
    yield from find_value(
        value, node[key], path + [key])

```

4. This has to be done for the list case as well. Start an examination of each item in the list:

```

for index in range(len(lnode)):
    for match in find_value_y(value, lnode[index], path +
        [index]):
        yield match

```

5. Replace the inner for with a yield from statement:

```

for index, item in enumerate(lnode):
    yield from find_value(
        value, item, path + [index])

```

The complete depth-first find\_value() search function, when complete, will look like this:

```

def find_value(
    value: Any,
    node: JSON_DOC,
    path: list[Node_Id] | None = None
) -> Iterator[list[Node_Id]]:
    if path is None:
        path = []
    match node:
        case dict() as dnode:
            for key in sorted(dnode.keys()):
                yield from find_value(
                    value, node[key], path + [key])

```

```
    case list() as lnode:
        for index, item in enumerate(lnode):
            yield from find_value(
                value, item, path + [index])
    case _ as pnode:
        # str, int, float, bool, None
        if pnode == value:
            yield path
```

When we use the `find_value()` function, it looks like this:

```
>>> places = list(find_value('value', document))
>>> places
[['array', 0, 'array_item_key1'], ['field2'], ['object', 'attribute1']]
```

The resulting list has three items. Each of these is a list of keys that form a path to an item with the target value of "value".

## How it works...

For background, see the *Writing generator functions with the yield statement* recipe in this chapter.

The `yield from` statement is shorthand for:

```
for item in x:
    yield item
```

The `yield from` statement lets us write a succinct recursive algorithm that will behave as an iterator and properly yield multiple values. It saves the overhead of a boilerplate `for` statement.

This can also be used in contexts that don't involve a recursive function. It's entirely sensible to use a `yield from` statement anywhere that an iterable result is involved. It's a handy simplification for recursive functions because it preserves a clearly recursive structure.

## There's more...

Another common style of definition assembles a list of items using append operations. We can rewrite this into an iterator and avoid the overhead of building and mutating a list object.

When factoring a number, we can define the set of prime factors of a number,  $x$ , like this:

$$F(x) = \begin{cases} x & \text{if } x \text{ is prime} \\ n \cup F(\frac{x}{n}) & \text{if } x \equiv 0 \pmod n \text{ and } 2 \leq n \leq \sqrt{x} \end{cases}$$

If the value,  $x$ , is prime, it has only itself in the set of prime factors. Otherwise, there must be some prime number,  $n$ , which is the least factor of  $x$ . We can assemble a set of factors starting with this number  $n$  and then append all factors of  $\frac{x}{n}$ . To be sure that only prime factors are found,  $n$  must be prime. If we search ascending values of  $n$ , starting from 2, we'll find prime factors before finding composite factors.

An eager approach builds a complete list of factors. A lazy approach be be a generator of factors for a consumer. Here's an eager list-building function:

```
import math

def factor_list(x: int) -> list[int]:
    limit = int(math.sqrt(x) + 1)
    for n in range(2, limit):
        q, r = divmod(x, n)
        if r == 0:
            return [n] + factor_list(q)
    return [x]
```

This `factor_list()` function will build a list object. If a factor,  $n$ , is found, it will start a list with that factor. It will then extend the list with the factors built from the value of  $x // n$ . If there are no factors of  $x$ , then the value is prime, and this returns a list with only the value of  $x$ .

(This has an inefficiency stemming from the way this searches for composite numbers as well as prime numbers. For example, after testing 2 and 3, this will also test 4 and 6, even though they're composite and all of their factors have already been tested. The example is centered on list-building, not efficient factoring of numbers.)

We can rewrite this as an iterator by replacing the recursive calls with `yield` from statements. The function will look like this:

```
def factor_iter(x: int) -> Iterator[int]:
    limit = int(math.sqrt(x) + 1)
    for n in range(2, limit):
        q, r = divmod(x, n)
        if r == 0:
            yield n
            yield from factor_iter(q)
    return
yield x
```

When a factor is found, the function will yield the factor, `n`, followed by all other factors found via a recursive call to `factor_iter()`. If no factors are found, the function will yield the prime number, `x`, and nothing more.

Using an iterator allows the client of this function to build any kind of collection from the factors. Instead of being limited to always creating a `list` object, we can create a multiset using the `collections.Counter` class. It would look like this:

```
>>> from collections import Counter
>>> Counter(factor_iter(384))
Counter({2: 7, 3: 1})
```

This shows us that:

$$384 = 2^7 \times 3$$

In some cases, this kind of multiset can be easier to work with than a simple list of factors.



What's important is that the multiset was created directly from the `factor_iter()` iterator without creating any intermediate `list` objects. This kind of optimization lets us build complex algorithms that aren't forced to consume large volumes of memory.

## See also

- In the *Designing recursive functions around Python's stack limits* recipe, earlier in this chapter, we covered the core design patterns for recursive functions. This recipe provides an alternative way to create the results.
- For background, see the *Writing generator functions with the yield statement* recipe in this chapter.

## Join our community Discord space

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

<https://packt.link/dHrHU>



# 10

## Working with Type Matching and Annotations

This chapter will look at how we can work with data structures that have a variety of data types. This often means inspecting the type of an attribute, an element of a tuple, or a value in a dictionary.

In previous chapters, we've avoided spending too much time on data validation considerations. In this chapter, we'll look closely at validating input values to be sure they conform to expected data types and value ranges.

This data validation is a kind of type-checking. It validates a narrower domain of values than the very broad classes of integer or string. The application must check the values of objects to be sure they're valid for the intended purpose.

Some data structures like JSON or XML documents can contain objects of a variety of data types. A common situation is summarized as **First Normal Form (1NF)**, where each item in a collection is of the same type. This isn't universal, however. When parsing complex

files like programming language statements, we'll see a sequence of distinct data types. The presence of diverse types means that the application software can't simply assume a single, consistent type, but must process the available data.

In this chapter, we'll look at a number of recipes related to types and type matching:

- *Designing with type hints*
- *Using the built-in type matching functions*
- *Using the match statement*
- *Handling type conversions*
- *Implementing more strict type checks with Pydantic*
- *Including run-time valid value checks*

## Designing with type hints

Annotations in function definitions were introduced to the language syntax back in 2006, without any formal semantics. The annotations idea came with a list of potential use cases, one of which was type checking. In 2014, the idea of type hints were solidified and formalized into a typing module and some associated tools including the **mypy** tool.

For a few years, *annotations* were a general kind of syntax and *type hints* were a specific use case for annotations. By 2017, other uses for annotations were deprecated and the annotation syntax was expressly focused on type hints. While there was once a subtle difference between annotations and type hints, the distinction has since evaporated, leaving us with two synonyms.

There are three important aspects of using type hints:

- Type hints are optional. We can write Python without type hints.
- Type hints can be applied *gradually*. Part of an application can have hints, where another part lacks them. Tools like **mypy** can tolerate mixtures of code with and without hints.

- Type hints are not used at run time and have no performance overhead.

Throughout this book, we've treated hints as essential to good software design. They're as essential as unit tests and coherent documentation, both of which are also technically optional, but essential for trustworthy software. We've found them to help prevent problems by enforcing a level of rigor and formality.

Python's processing relies on *duck typing* rules. For more background, see *Chapter 8*, specifically, the *Leveraging Python's duck typing* recipe. There are two broad design patterns available to us:

- A strict hierarchy with a common superclass.
- Leveraging duck typing, a collection of classes can have common features, often defined as a protocol that specifies the relevant features.

In this recipe, we'll look at two approaches to designing code that includes type hints and can be checked by tools like **mypy**.

## Getting ready

We'll look at a problem that involves working with two distinct types of data that are mingled together in a source file. In this case, we're going to classify the contents of the data directory with a large number of data files. Additionally, we have an `src` directory with a large number of sub-directories that contain application programs and scripts. We want to create a collection of data structures to represent two distinct classes of data files:

- Data files not named by any application program or script
- Data files that are referenced by one or more application programs

## How to do it...

There are two broad strategies for designing this kind of program:

- Sketch out the data types and transformations first, then write code to fit the types.
- Write code first and then add type hints to the working code.

Neither can be described as best. In many cases, the two evolve side by side.

We'll look at each of these in separate variations in this recipe.

## Type hints first design

We're going to work with diverse classes of objects. In this variation, we'll define the type hints first, and then fill in the needed processing. Here's how we can define a classifier and the related classes starting from the class definitions:

1. Define the two subclasses. In this example, we'll call them `Unreferenced` and `Referenced` files. For each class, write a sentence describing the unique purpose for the instances of each class. These serve as the starting point for the class definitions.
2. Chose an appropriate variety of available classes. This might be an ordinary class with mutable attributes, a `NamedTuple`, or a `@dataclass`. Starting with a `@dataclass` often gives the most flexibility. Switching between named tuples, dataclasses, and frozen dataclasses involves minimal syntax changes:

```
from pathlib import Path
from dataclasses import dataclass

@dataclass
class Referenced:
    """Defines a data file and applications that reference it."""
```

The `Unreferenced` class definition would be similar, with an appropriate docstring.

3. Add the attributes with values that will define the state of each instance. For the `Referenced` class, this is the `Path` and a collection of `Path` objects for each source file that has a reference. These two attribute definitions look like this:

```
datafile: Path
recipes: list[Path]
```

For the `Unreferenced` class, however, there really aren't very many other attributes beyond the path. This raises an interesting question: does this deserve a separate

class declaration, or can it simply be a Path object?

Because Python permits type aliases and type unions, there's no real need for an Unreferenced class; the existing Path will do. It is helpful to provide a type alias for this:

```
from typing import TypeAlias

Unreferenced: TypeAlias = Path
```

4. Formalize the union of these distinct classes.

```
ContentType: TypeAlias = Unreferenced | Referenced
```

Now that we have the type definitions, we can write a function that is an iterator over the ContentType union of classes. This function will yield a sequence of Unreferenced and Referenced objects, one for each data file.

The function might look like this:

```
def datafile_iter(base: Path) -> Iterator[ContentType]:
    data = (base / "data")
    code = (base / "src")
    for path in sorted(data.glob("*.py")):
        if not path.is_file():
            continue

        used_by = [
            chap_recipe.relative_to(code)
            for chap_recipe in code.glob("**/*.py")
            if (
                chap_recipe.is_file()
                and "__pycache__" not in chap_recipe.parts
                and ".venv" not in chap_recipe.parts
                and "ch10" not in chap_recipe.parts
                and path.name in chap_recipe.read_text()
            )
        ]

        if used_by:
```

```
        yield Referenced(path.relative_to(data), used_by)
    else:
        yield path.relative_to(data)
```

The `datafile_iter()` function skips past any non-file names in the data directory. It also skips some source code directories, `__pycache__`, and `.venv`. Additionally, we have to ignore some of the files in *Chapter 10* because these files will have test cases that contain names of data files, creating confusing results.

If a data file name appears in a source file, the reference is saved in the `used_by` collection. Files with a non-empty `used_by` collection will create a `Referenced` instance. The remaining files are `Path` objects; because of the `TypeAlias` these are recognized as `Unreferenced` instances, also. We don't need to formally cast or convert a `Path` object to the `Unreferenced` type. Tools like **mypy** will use the `TypeAlias` to see the equivalence without any additional code.

The resulting iterator provides a mix of objects of distinct types. In the *Using the match statement* recipe, we'll look at convenient ways to process objects of diverse types.

## Code first design

We're going to work with diverse classes of objects. In this variation, we'll define the processing first, and then fold in type hints to clarify our intent. Here's how we can define a classifier and the related classes starting from a function definition:

1. Start with a function definition that provides the needed parameters:

```
def datafile_iter(base):
    data = (base / "data")
    code = (base / "src")
```

2. Write the processing to accumulate the required data values. In this case, we need to iterate through the names of the data files. For each data file, we need to look for references in all of the source files.

```

for path in sorted(data.glob("*. *")):
    if not path.is_file():
        continue

    used_by = [
        chap_recipe.relative_to(code)
        for chap_recipe in code.glob("**/*.py")
        if (
            chap_recipe.is_file()
            and "__pycache__" not in chap_recipe.parts
            and ".venv" not in chap_recipe.parts
            and "ch10" not in chap_recipe.parts
            and path.name in chap_recipe.read_text()
        )
    ]

```

3. Decide what the various outputs from the function need to be. In some cases, we can yield tuple objects with the various kinds of values that are available.

```

if used_by:
    yield (path.relative_to(data), used_by)
else:
    yield path.relative_to(data)

```

For paths without any references in the source, we yield the Path object. For paths that have references in the source, we can yield the data Path and a list of source Path instances.

4. For objects with more complicated internal state, consider introducing a class definition to properly encapsulate the state. For this example, it makes sense to introduce a type for data files that have references. This would lead to replacing a simple, anonymous tuple with a NamedTuple like the following:

```

from typing import NamedTuple
class Referenced(NamedTuple):
    datafile: Path
    recipes: list[Path]

```



This, in turn, leads to revising the `yield` statement for the Referenced instances.

```
yield Referenced(path.relative_to(data), used_by)
```

5. Revisit the function definition to add type hints.

```
def datafile_iter_2(base: Path) -> Iterator[Path | Referenced]:
```

The processing in both variants of the recipe is nearly identical. The differences are minor choices on how best to present the results. In the previous example, an explicit union named `Content_Type` was created. For this version, the union is implicit.

## How it works...

Python duck typing permits a great deal of latitude in design. We can start with type definitions or we can start from code and add type hints. The final code will tend to be similar because it performs the same processing on the same data.

The choice between a *code first* or *type first* approach may lead to an insight about performance or optimization. Each choice emphasizes distinct attributes of the final code. The code-first approach can emphasize simple processing, where type first might emphasize uniformity of the objects being processed. The choice of approach can also stem from the author's degree of comfort with Python's types.

In some cases, the process of writing the type hints may suggest algorithms or optimizations. This can lead to beneficial refactoring of code already written.

It's important to note that the presence or absence of type hints has no performance impact. Any performance gains (or losses) are ordinary design issues that might be made more visible through the use of type hints.

## There's more...

When decomposing a large problem into smaller pieces, the interfaces among the smaller pieces are essential design decisions that must be made early in the design process. An early

decision on data structures often leads to a *type first* design process overall. The externally facing components must have well-defined interfaces. The functions or methods that support these external components may be designed more freely with fewer constraints.

This leads to *type first* for the overall architecture of complicated software, reserving a choice of *type first* or *code first* design when working on the more detailed layers. When we consider distributed applications – like web services – where servers and clients are on separate machines, we find that *type first* is essential.

As the volume of code grows, the importance of type hints also grows. It’s challenging to keep a lot of details in the space between one’s ears. Having a type hint to summarize a more complicated data structure can reduce the clutter of details around the code.

In distributed computing environments, we’ll often need to consider that some components may not be Python programs. In these cases, we can’t share Python type hints. This means we’re forced to use a schema definition that exists outside Python, but provides needed mappings to Python types.

Examples of these kinds of formal definitions that transcend languages include JSON Schema, Protocol Buffers, AVRO, and many others. The JSON Schema approach is typical, and is supported by a number of Python tools. Later in this chapter, we’ll look at using **Pydantic**, which has support for defining data using JSON Schema.

## See also

- In the *Reading JSON and YAML documents* recipe in Chapter 11, we’ll return to using JSON documents for complicated data.
- In the *Using the match statement* recipe, later in this chapter, we’ll look at how to use the match statement to process data of a variety of types. This makes it relatively easy to work with unions of types.
- In the *Implementing more strict type checks with Pydantic* recipe, later in this chapter, we’ll look at stronger type definitions using the pydantic package.

## Using the built-in type matching functions

When we have a collection of objects with mixed types, we often need to distinguish among the types. When working with classes that we've defined, it's possible to define classes that are properly *polymorphic*. This is not generally the case when working with Python's internal objects, or working with collections of data that involve a mixture of classes we've defined, and built-in classes that are part of Python.

When working entirely with our own classes, we can design them to have common methods and attributes, but offer distinct behavior depending on which of the subclasses is involved. This kind of design fits the “L” design principle in the S.O.L.I.D design principles: the *Liskov Substitution Principle*. Any of the subclasses can be used in place of the superclass, because they all have a common set of method definitions. For more information on this, see *Chapter 8*.

This kind of abstraction-driven design is not always needed with Python. Because of Python's duck-typing, designs don't require a common superclass. In some cases, it isn't even practical: we may have diverse types without a unifying abstraction. It's very common to work with mixtures of objects from built-in classes, as well as objects from our own class definitions. We can't impose polymorphism on the built-in classes.

How can we leverage the built-in functions to write functions and methods that are flexible with respect to type? For this recipe, we'll reuse the processing from the *Designing with type hints* recipe earlier in this chapter.

### Getting ready

In the *Designing with type hints* recipe, we defined a function `datafile_iter()` that emitted two distinct types of objects: Path objects and Referenced objects. A Referenced object was a bundle of Path instances, showing a data file that was used by one or more application programs. A stand-alone Path object was a data file not used by any application program. These unreferenced paths are candidates for removal to reduce clutter.

We need to process these two classes of objects in distinct ways. They're created by a single

generator function, `datafile_iter()`. This function emits a sequence of `Unreferenced` and `Referenced` instances. This mixture means an application must filter objects by their type.

The application will work with a sequence of objects. These will be created by a function with the following definition:

```
from collections.abc import Iterator
DataFileIter: TypeAlias = Iterator[Unreferenced | Referenced]

def datafile_iter(base: Path) -> DataFileIter:
```

The `datafile_iter()` function will produce a sequence of `Unreferenced` and `Referenced` objects. This will reflect the current state of files in a given directory. Some will have references in source code; others will lack any references. See the *Designing with type hints* recipe for this function.

## How to do it...

The application function to do analysis will consume objects of a variety of types. The function is designed as follows:

1. Start with a definition like the following that shows the types consumed:

```
from collections.abc import Iterable

def analysis(source: Iterable[Unreferenced | Referenced]) -> None:
```

2. Create an empty list that will hold the data files that have references. Write the `for` statement to consume objects from the source iterable, and populate that list:

```
good_files: list[Referenced] = []
for file in source:
```

3. To distinguish objects by type, we can use the `isinstance()` function to see if an object is a class of a given type.

To distinguish the class, use the `isinstance()` function:

```
if isinstance(file, Unreferenced):
    print(f"delete {file}")
elif isinstance(file, Referenced):
    good_files.append(file)
```

4. While it's technically unnecessary, it always seems prudent to include an `else` condition to raise an exception in the unlikely event that the `datafile_iter` function was changed in some astonishing way:

```
else:
    raise ValueError(f"unexpected type {type(file)}")
```

For more about this design pattern, see the *Designing complex if...elif chains* recipe in *Chapter 2*.

5. Write the final summary:

```
print(f"Keep {len(good_files)} files")
```

## How it works...

The `isinstance()` function examines an object to see what classes it belongs to. The second argument can be single class or a tuple of alternative classes.

It's important to note that an object often has a number of parent classes, forming a lattice, stemming from the class object. If multiple inheritance is being used, there can be a large number of paths through the super class definitions. The `isinstance()` function examines all the alternative parent classes.

The `isinstance()` function is aware of `TypeAlias` names in addition to the classes imported and defined within the application. This gives us a great deal of flexibility to use meaning names in type hints.



In Python 3.12, the `TypeAlias` construct can be replaced with the new type statement:

```
type Unreferenced = Path
```

See [Mypy Issue #15238](#) for more information on support for the type statement by the **mypy** tool.

Until this is resolved, we've elected to use `TypeAlias` in this book.

## There's more...

The `isinstance()` function is the kind of Boolean function that works well with the `filter()` higher-order function. For more information, see the *Picking a subset – three ways to filter* recipe in [Chapter 9](#).

In addition to the built-in `isinstance()` to interrogate objects, there is also a `issubclass()` function that lets an application examine type definitions. It's important to distinguish between instances of a class and a class object; the `issubclass()` function is used to examine type definitions. The `issubclass()` function is often used for *metaprogramming*: software that's concerned with software rather than the application data. When designing functions that work with types of objects, rather than objects, the `issubclass()` function is necessary.

When examining the type of objects, the `match` statement is often a better choice than the `isinstance()` function. The reason is that a `match` statement's case clause has very sophisticated type pattern matching, where the `isinstance()` function is limited to ensuring the object has a given class (or a class in a tuple of classes) in its parents.

## See also

- See the *Using the match statement* recipe for an alternative to this, using the `match` statement.

- See *Chapter 7* and *Chapter 8* for several recipes related to playing cards and the interesting class hierarchies they involve.

## Using the match statement

One important reason for defining a collection of closely-related types is to distinguish the processing that applies to the objects. One technique for providing distinct behavior is by using a *polymorphic* design: a number of subclasses provide distinct implementations of a common function. When working entirely with our own classes, we can design them to have common methods and attributes, but offer distinct behavior depending on which of the subclasses is involved. This is covered in *Chapter 8*.

This is not generally possible when working with Python's internal objects, or when working with collections of data that involve a mixture of classes we've defined, and built-in classes that are part of Python. In these cases, it's simpler to rely on type matching to implement distinct behaviors. One approach was shown in the *Using the built-in type matching functions* recipe in this chapter.

We can also use the match statement to write functions and methods that are flexible and work with argument values of a variety of types. For the recipe, we'll reuse the processing from the *Designing with type hints* and *Using the built-in type matching functions* recipes earlier in this chapter.

## Getting ready

In the *Designing with type hints* recipe, we defined a `datafile_iter()` function that emitted two distinct types of objects: Path objects and Referenced objects.

We need to process these two classes of objects in distinct ways. This mixture means an application must filter them by their type.

## How to do it...

The application will work with a sequence of objects of distinct types. The function is designed as follows:

1. Start with a definition like the following that shows the types consumed:

```
from collections.abc import Iterable

def analysis(source: Iterable[Unreferenced | Referenced]) -> None:
```

This function will consume an iterable sequence of objects. This function will count the ones that have references. It will suggest deleting the files with no references.

2. Create an empty list that will hold the data files that have references. Write the for statement to consume objects from the source iterable:

```
good_files: list[Referenced] = []
for file in source:
```

3. Write the start of the match statement with the file variable:

```
match file:
```

4. To process a file of the various classes, create case statements that show the kinds of objects that must be matched. These cases are indented within the match statement:

```
case Unreferenced() as unref:
    print(f"delete {unref}")
case Referenced() as ref:
    good_files.append(file)
```

5. While it's technically unnecessary, it always seems prudent to include a case `_:` condition. The `_` will match anything. The body of this clause can raise an exception in the unlikely event that the `datafile_iter` function was changed in some astonishing way:

```
case _:
    raise ValueError(f"unexpected type {type(file)}")
```

For more about this design pattern, see the *Designing complex if...elif chains* recipe in



## Chapter 2.

6. Write the final summary:

```
print(f"Keep {len(good_files)} files")
```

## How it works...

The `match` statement uses a sequence of case clauses to establish a class that matches the given object. While there are a wide variety of case clauses, one common case is the case `class() as name: variant`, called the *class pattern*. Within `()`, we can provide sub-patterns to match objects with specific kinds of parameters.

For this example, we didn't need the more sophisticated matching patterns. We can provide what looks like an instance – made from the class name and `()` – to show that the case clause will match an instance of the class. No additional detail regarding the structure of the instance is necessary.

The use of case `Unreferenced()` almost looks as if the expression `Unreferenced()` will create an instance of the `Unreferenced` class. The intent here is not to create an object, but to write an expression that looks very much like object creation. This syntax helps to clarify the intent of using the case to match any object of the named class.

Other patterns are available that allow matching simple literal values, sequences, and mappings, as well as classes. Further, there are ways to provide groups of alternatives, and even apply additional filtering via a guard condition that's used in conjunction with the pattern matching.

The case `_` clause is a wildcard clause. It will match anything provided in the `match` statement. The `_` variable name has special significance here, and only this variable can be used.

Central to this design is the clarity of the case definitions. These are much more readable than `isinstance()` function evaluation in a series of `elif` clauses.

## There's more...

We'll extend this recipe to show some of the sophisticated type matching available in these case clauses. Consider the case where we want to separate a referenced file that has only a single item in the list of applications that refer to it.

We're looking for objects that look like this specific example:

```
single use: Referenced(datafile=PosixPath('race_result.json'),
                      recipes=[PosixPath('ch11/recipe_06.py')])
```

This case can be summarized as `Referenced(_, [Path()])`. We want to match an instance of the `Referenced` class where the second parameter is a list with a single `Path` instance.

This turns into a new case clause. Here's the new, more specific case, followed by the more general case:

```
case Referenced(_, [Path()]) as single:
    print(f"single use: {single}")
    good_files.append(single)
case Referenced() as multiple:
    good_files.append(multiple)
```

The `match` statement works through the cases in order. The more-specific cases *must* precede the less-specific cases. If we flip the order of these two cases, case `Referenced()` would match before case `Referenced(_, [Path()])` would even be examined. The most general case, case `_:` must be last.

## See also

- See the *Using the built-in type matching functions* recipe for an alternative approach using the built-in `isinstance()` function.
- See *Chapter 8* for several recipes related to polymorphic class design. Sometimes, this can reduce the need for type matching.

## Handling type conversions

One of the useful features of Python is the “numeric tower” idea. See **The Numeric Tower** in the Python standard library documentation. The idea is that numeric values can move “up” the tower from integral to rational to real to complex.

The numeric conversions are based on the idea that there are several overlapping domains of numbers. These include  $\mathbb{Z}$  integers,  $\mathbb{Q}$  rational numbers,  $\mathbb{P}$  irrational numbers,  $\mathbb{R}$  real numbers, and  $\mathbb{C}$  complex numbers. The idea is that these form a nested series of sets:  $\mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$ . Also,  $\mathbb{Q} \cup \mathbb{P} = \mathbb{R}$ : the real numbers include rational and irrational numbers.

These built-in numeric types follow the abstract concepts:

- $\mathbb{C}$  is implemented by the `complex` type. Any of the types below this type can be converted to a complex value.
- $\mathbb{R}$  is supported by the `float` type. It’s important to note that `float` involves approximations, and doesn’t fully match the mathematical ideal of real numbers. When an operator in this class encounters `int` or `fraction` values, it will create the equivalent `float` value.
- $\mathbb{Q}$  uses the `Fraction` class in the `fractions` module. When an arithmetic operator in the `Fraction` class encounters an `int` it will quietly create a `Fraction` with the same value as the integer.  $\frac{z}{1} = z$ .
- $\mathbb{Z}$  is the `int` class.

Generally, the Python language avoids too many conversions to other types. Strings, for example, are not automatically parsed to create numeric values. An explicit built-in function like `int()` or `float()` needs to be used to process strings with numbers.

We’ll often want our own types to share this kind of behavior. We’d like our functions to be flexible, and convert objects to other types when needed. We may, for example, want to permit a number of representations for a latitude-longitude point. These alternatives might include:

- A tuple of two floating-point numeric values
- A pair of strings, with each string representing a floating-point value
- A single string with two numeric values separated by a " , " character

As with the numeric tower, our own class definitions need to convert other types into the needed target type.

## Getting ready

We'll consider a function to compute distances between points on the surface of the Earth. This involves some clever spherical trigonometry. For more information, see *Chapter 3*, specifically, the *Picking an order for parameters based on partial functions* recipe. Also see the *Creating contexts and context managers* recipe in *Chapter 7*.

The function is defined as:

```
def haversine(
    lat_1: float, lon_1: float,
    lat_2: float, lon_2: float, *, R: float) -> float:
    ... # etc.
```

This definition requires converting source data into individual float values. In applications that integrate data from a number of sources, these conversions are so common that it seems better to centralize them into a function that wraps the essential `haversine()` computation.

We want a function like this:

```
def distance(
    *args: str | float | tuple[float, float],
    R: float = NM
) -> float:
```

This function will compute distances among points defined as a variety of data types. The `*args` parameter means all of the positional argument values will be combined into a single

tuple. A number of validation rules must be applied to make sense of this tuple. Here are the rules we'll start with:

- Four float values: use these directly.  
Example: `distance(36.12, -86.67, 33.94, -118.40, R=6372.8)`.
- Four strings: convert these to float.  
Example: `distance("36.12", "-86.67", "33.94", "-118.40", R=6372.8)`.
- two strings: parse each string, breaking on a “,”. Each string should have two float values.  
Example: `distance("36.12,-86.67", "33.94,-118.40", R=6372.8)`.
- Two tuples: unpack each tuple to make sure it has two float values.  
Example: `distance((36.12, -86.67), (33.94, -118.40), R=6372.8)`.

Ideally, it might be nice to support combinations of these, also. We'll design a function that performs the needed type conversions.

## How to do it...

A function that includes type conversions is often built separately from the underlying processing. It can help testing and debugging if these two aspects of processing – conversions and computations – are separated:

1. Import the needed `literal_eval()` function to do the conversions of strings that are expected to be Python literals:

```
from ast import literal_eval
```

With this function, we can evaluate `literal_eval("2,3")` to get a result of a proper tuple, `(2, 3)`. We don't need to use a regular expression to decompose the string to see the pattern of the text.

2. Define the distance function that performs conversions:

```
def distance(
    *args: str | float | tuple[float, float],
    R: float = NM
) -> float:
```

3. Start the match statement for the various kinds of argument patterns.

```
match args:
```

4. Write the individual cases, moving from more specific to less specific. Start with four distinct float values, since no conversion needs to be done. The tuple of float values has a more complex type structure, but doesn't require any conversion.

```
    case [float(lat_1), float(lon_1), float(lat_2), float(lon_2)]:
        pass
    case (
        [[float(lat_1), float(lon_1)],
         [float(lat_2), float(lon_2)]]
    ):
        pass
```

We've provided the `lat_1`, `lon_1`, `lat_2`, and `lon_2` variables to bind the values from the `args` structure to variable names. This saves us from having to write assignment statements to unpack an argument tuple. The `pass` statement placeholder is used because no further processing is required beyond unpacking the data structure.

5. Write the cases that involve conversions of the supplied values:

```
    case [str(s1), str(s2), str(s3), str(s4)]:
        lat_1, lon_1, lat_2, lon_2 = (
            float(s1), float(s2), float(s3), float(s4)
        )
    case [str(l11), str(l12)]:
        lat_1, lon_1 = literal_eval(l11)
        lat_2, lon_2 = literal_eval(l12)
```

When the argument values are four strings, we provided four variables to unpack

the four strings.

When the argument pattern is two strings, we provided two variables, `l11` and `l12`, that each needed to be converted into two tuples of numbers and then unpacked.

6. Write the default case that will match anything else and raise an exception:

```
case _:
    raise ValueError(f"unexpected types in {args!r}")
```

7. Now that the arguments have been properly unpacked and any conversions applied, use the `haversine()` function to compute the required result:

```
return haversine(lat_1, lon_1, lat_2, lon_2, R=R)
```

## How it works...

The essential feature for type conversions is using a `match` statement to provide appropriate conversions for the supported types. In this example, we tolerated a mixture of strings and tuples that could be converted and unpacked to locate the required four argument values. The `match` statement has many clever type-matching rules. For example, an expression like `((float(f1), float(f2)), (float(f3), float(f4)))` will match two tuples, each with two float values. Further, it unpacks the values from the tuples and assigns them to four variables.

The mechanics of converting the values are also based on a built-in feature. The `float()` function converts numeric strings to float values or raises a `ValueError` exception.

The `ast.literal_eval()` function is very handy for evaluating strings that are Python literals. The function is safe from evaluating dangerous expressions because it is limited to literal values, and a few simple data structures – tuples, lists, dicts, and sets – built from literal values. It permits us to parse a string like `"36.12, -86.67"` into `(36.12, -86.67)` directly.

## There's more...

The use of independent case clauses makes it relatively easy to add additional type conversions. We might, for example, want to handle a tuple of two dictionary structures that look like `{"lat": 36.12, "lon": -86.67}`. This can be matched with the following case:

```
case (
    {"lat": float(lat_1), "lon": float(lon_1)},
    {"lat": float(lat_2), "lon": float(lon_2)}
):
    pass
```

The argument tuple pattern has `()` around it, making it easy to break it into multiple lines. The four values extracted from the dictionaries will be bound to four target variables.

If we want to permit more flexibility, we can consider the case where we have two argument values of a mixture of type patterns. For example, `distance("36.12, -86.67", (33.94, -118.40), R=6372.8)`. This has two distinct formats: a string and a tuple with a pair of float values.

Rather than enumerate all of the possible combinations, we can decompose the parsing of a pair of values into a separate function, `parse()`, that applies the same conversion to both argument values:

```
case [p_1, p_2]:
    lat_1, lon_1 = parse(p_1)
    lat_2, lon_2 = parse(p_2)
```

This new `parse()` function must handle all the cases where a latitude and longitude are provided together. This includes strings, tuples, and mappings. It looks like this:

```
def parse(item: Point | float) -> tuple[float, float]:
    match item:
        case [float(lat), float(lon)]:
            pass
        case {"lat": float(lat), "lon": float(lon)}:
            pass
```



```
    case str(s11):
        lat, lon = literal_eval(s11)
    case _:
        raise ValueError(f"unexpected types in {item!r}")
    return lat, lon
```

This will slightly simplify the match statement in the distance function. The refactored statement only handles four cases:

```
match args:
    case [float(lat_1), float(lon_1), float(lat_2), float(lon_2)]:
        pass
    case [str(s1), str(s2), str(s3), str(s4)]:
        lat_1, lon_1, lat_2, lon_2 = float(s1), float(s2), float(s3),
        float(s4)
    case [p_1, p_2]:
        lat_1, lon_1 = parse(p_1)
        lat_2, lon_2 = parse(p_2)
    case _:
        raise ValueError(f"unexpected types in {args!r}")
```

The first two cases handle the situation where four argument values are provided. The third case looks at a pair of values, which can have any of the pair formats.

We expressly avoid the case where three argument values are provided. This requires a bit more care to interpret, since one of the three argument values must be a latitude and longitude pair. The other two values must be separated latitude and longitude values. The logic is not overwhelmingly complicated, but the details stray from the central idea of this recipe.

While the recipe focuses on built-in types including `str` and `float`, any type can be used. A customized `Leg` type, for example, with start and stop locations could easily be added in a case clause.

## See also

- For more information on numbers and conversions, see the *Choosing between float, decimal, and fraction* recipe in *Chapter 1*. This provides some more information about the limitations of the `float` approximation.
- For more information on the `haversine()` function, see the *Picking an order for parameters based on partial functions* recipe in *Chapter 3*. Also see the *Creating contexts and context managers* recipe in *Chapter 7*.

## Implementing more strict type checks with Pydantic

For the most part, Python's internal processing will handle a great many simple validity checks properly. If we've written a function to convert a string to float, the function will work with float values and string values. It will raise a `ValueError` exception if we try to apply the `float()` function to a `Path` object.

In order for type hints to be optional, run-time type-checking is the minimum level of checking required to make sure some processing can proceed. This is emphatically distinct from the strict checks that tools like **mypy** make.



Type hints do no run-time processing.

Python (without any add-on packages) does no data type checks or value range checks at run-time. Exceptions are raised when an operator is confronted with a type it can't process, without regard to the type hints.

This means that Python may be able to process a type that was excluded by a hint. It's possible to write a narrow hint like `list[str]`. An object of `set[str]` may also work with the given body of the function.

There are applications where we'd like stronger checks at run-time. These are often helpful in applications where extensions or plug-ins are used, and we'd like to be sure the additional

plug-in code behaves properly.

One way to provide for run-time type checking is to use the **Pydantic** package. This module allows us to define complex objects that are accompanied by run-time type checking, as well as management of schema definitions that can be shared widely.

In *Chapter 5*, in the *Creating dictionaries – inserting and updating* recipe, we looked at a log file that we needed to parse into a more useful structure. In *Chapter 9*, in the *Writing generator functions with the yield statement* recipe, we looked at writing a generator function that would parse and yield the parsed objects. We called the resulting objects `RawLog`, with no type checks or type conversions. We applied a simple transformation to create a `DatedLog` instance with the date-time stamp converted from text to a `datetime.datetime` object.

The `pydantic` package can handle some of this conversion to a `DatedLog` instance, saving us some programming. Further, because the schema can be generated automatically, we can build a JSON Schema definition and do JSON serialization without a lot of complicated work.

The **Pydantic** package must be downloaded and installed. Generally, this is done with the following terminal command:

```
(cookbook3) % python -m pip install pydantic
```

Using the `python -m pip` command ensures that we will use the `pip` command that goes with the currently active virtual environment, shown as `cookbook3` in the example.

## Getting ready

The log data has date-time stamps represented as string values. We need to parse these to create proper `datetime` objects. To keep things focused in this recipe, we'll use a simplified log produced by a web server written with Flask.

The entries start out as lines of text that look like this:

```
[2016-06-15 17:57:54,715] INFO in ch10_r10: Sample Message One
[2016-06-15 17:57:54,716] DEBUG in ch10_r10: Debugging
[2016-06-15 17:57:54,720] WARNING in ch10_r10: Something might have gone
wrong
```

We've seen other examples of working with this kind of log in the *Using more complex structures – maps of lists* recipe in *Chapter 8*. Using REs from the *String parsing with regular expressions* recipe in *Chapter 1*, we can decompose each line into a more useful structure.

Looking at the other recipes, the regular expression used for parsing had an important feature. The names used in the (`?P<name>...`) groups were specifically designed to be ordinary Python attribute names. This will fit nicely with the class definition we'll build later.

We'll need to define a class that captures the essential content of each log line in a useful form. We'll use the **Pydantic** package to define and populate this class.

## How to do it...

1. We'll need the following imports to create this class definition:

```
import datetime
from enum import StrEnum
from typing import Annotated
from pydantic import BaseModel, Field
```

2. In order to properly validate a string that has a number of values, an Enum class is required. We'll define a subclass of `StrEnum` to list the valid string values. Each class-level variable provides a name and the string literal that is the serialization for the name:

```
class LevelClass(StrEnum):
    DEBUG = "DEBUG"
    INFO = "INFO"
    WARNING = "WARNING"
    ERROR = "ERROR"
```

In this class, the Python attribute names and the string literals match. This isn't a requirement. It happens to be convenient for this collection of enumerated string values.

3. The class will be a subclass of the `BaseModel` class from the `pydantic` package:

```
class LogData(BaseModel):
```

The `BaseModel` class must be the superclass for any model that makes use of `pydantic` features.

4. We'll define each field with a field name that matches the group name in the regular expression used to parse the fields. This is not a requirement, but it makes it very easy to build instances of the `LogData` class from the group dictionary that's part of a regular expression `Match` object:

```
    date: datetime.datetime
    level: LevelClass
    module: Annotated[str, Field(pattern=r'^\w+$')]
    message: str
```

The `date` is defined to be a `datetime.datetime` instance. The inherited methods from the `BaseModel` class will handle this conversion. The `level` is an instance of the `LevelClass`. Again, features from `BaseModel` will handle this conversion for us. We've used the `Annotated` type to provide a type, `str`, and an annotation argument, `Field(...)`. This will be used by the methods of `BaseModel` to validate the field's content.

Here's a generator function to read and parse log records:

```
from typing import Iterable, Iterator

def logdata_iter(source: Iterable[str]) -> Iterator[LogData]:
    for row in source:
        if match := pattern.match(row):
            l = LogData.model_validate(match.groupdict())
```

```
yield 1
```

This will use the regular expression pattern, `pattern` to parse each record. The group dictionary, `match.groupdict()` will have the group names and the parsed text. The `model_validate()` method of the `BaseModel` will build an instance of the `LogData` class from the dictionary created by the compiled regular expression.

It looks like the following example when we use this `logdata_iter` function to create instances of the `LogData` class:

```
>>> from pprint import pprint
>>> pprint(list(logdata_iter(data.splitlines())))
[LogData(date=datetime.datetime(2016, 6, 15, 17, 57, 54, 715000),
level=<LevelClass.INFO: 'INFO'>, module='ch10_r10', message='Sample Message
One'),
LogData(date=datetime.datetime(2016, 6, 15, 17, 57, 54, 716000),
level=<LevelClass.DEBUG: 'DEBUG'>, module='ch10_r10', message='Debugging'),
LogData(date=datetime.datetime(2016, 6, 15, 17, 57, 54, 720000),
level=<LevelClass.WARNING: 'WARNING'>, module='ch10_r10',
message='Something might have gone wrong')]
```

This function has transformed lines of text into `LogData` objects populated with proper Python objects: `datetime.datetime` instances and enumerated values from the `LevelClass`. Further, it's validated the module names to be sure they match a specific regular expression pattern.

## How it works...

The **Pydantic** package includes numerous tools for data validation and class definition. Python's use of types, and more detailed Annotated types, provides syntax that helps us define the members of a class, including data conversions and data validations. In this example, the conversions were implied; the class provided the target type, and the methods inherited from the `BaseModel` class made sure that source data was properly converted to the desired target type.

This small class definition had three distinct kinds of types hints:

- The `date` and `level` field involved conversions to a target type.
- The `module` field used an annotated type to provide a **Pydantic** `Field` definition for the attribute. The regular expression pattern will check each string value to be sure it matches the required pattern.
- The `message` field provided a simple type that will match the source data type. No additional validation will be performed for this field.

There are some parallels between the way the `@dataclass` and a `BaseModel` subclass work. The **Pydantic** package provides considerably more sophisticated definitions than a `dataclass` definition. A `@dataclass`, for example, does not do type checking or any automatic data conversion. The type information provided when defining a `dataclass` is primarily of interest to tools like **mypy**. In contrast, the subclasses of `BaseModel` do considerably more automated conversion and run-time type checking.

The subclasses of `DataModel` come with a large number of methods.

The `model_dump_json()` and `model_validate_json()` methods are particularly helpful for web services where the application often works with RESTful transfers of object state in JSON notation. These can be serialized into newline-delimited files to collect a number of complicated objects into files in a standardized physical format.

The **Pydantic** package tends to be extremely fast. The current version involves Python extensions that are compiled to provide very high performance. Clearly, a `dataclass` – which lacks a number of **Pydantic** features – will be faster, but do less. However, the additional data validation is often worth the overhead.

## There's more...

One of the benefits of working with **Pydantic** is the automatic support for JSON Schema definitions and JSON serialization.

This shows how we can get the JSON Schema for a model:

```
>>> import json
>>> print(json.dumps(LogData.model_json_schema(), indent=2))
```

The details of the JSON Schema are long and match the Python definition of the class. We've omitted the output.

We can serialize these `LogData` instances in JSON notation. Here's how this looks:

```
>>> for record in logdata_iter(data.splitlines()):
...     print(record.model_dump_json())
{"date": "2016-06-15T17:57:54.715000", "level": "INFO", "module": "ch10_r10",
 "message": "Sample Message One"}
{"date": "2016-06-15T17:57:54.716000", "level": "DEBUG", "module": "ch10_r10",
 "message": "Debugging"}
{"date": "2016-06-15T17:57:54.720000", "level": "WARNING", "module": "ch10_r10",
 "message": "Something might have gone wrong"}
```

We've used the `model_dump_json()` method to serialize the object as a JSON document. This lets us convert documents from a variety of sources to a common format. This makes it easy to create analytic processing around the common format, separating parsing, merging, and validating from the analysis and the interesting results of the analytic processing.

## See also

- See the *Including run-time valid value checks* recipe for some additional validation rules that are possible.
- See the *Using dataclasses for mutable objects* recipe in *Chapter 7* for more on dataclasses. The **Pydantic** variant on dataclasses is often more useful than the `dataclasses` module.
- See the *Reading JSON and YAML documents* recipe in *Chapter 11* for information related to reading data in JSON format.



## Including run-time valid value checks

Data analytics often involves a great deal of “data wrangling”: dealing with invalid data, or unusual data. It’s common for source application software to change, leading to new formats for data files, causing problems in downstream analytic applications when parsing those files. A change in enterprise processes or policies may lead to new data types or new coded values that can disrupt analytic processing.

Similarly, when working with machines and robots, sometimes called the *Internet of Things*, it’s common for a device to provide invalid data when it’s starting up, or when it’s failing to operate normally. In some cases, it may be necessary to raise alarms when bad data arrives. In other cases, the out-of-range data needs to be quietly ignored.

The **Pydantic** package offers very sophisticated validation functions that allow us two choices:

- Convert data from unusual formats into Python objects.
- Raise an exception for data that cannot be converted or fails to pass more specific domain checks.

In some cases, we also need to validate the resulting object is internally consistent. This often means that several fields must be checked for consistency with each other. This is called *model validation*, which is distinct from isolated *field validation*.

The idea of validation can be extended. It can embrace both rejecting invalid data and filtering out data that’s valid, but uninteresting, for a given application.

## Getting ready

We’re looking at the US **National Oceanographic and Atmospheric Administration (NOAA)** data on coastal tides. Moving a large sailboat means making sure there’s enough water for it to float. This constraint requires checking predictions for the height of the tides at places that are known to be shallow and difficult to pass.

In particular, a place called El Jobean, on the Myakka river, has a shallow spot that requires

some care when transiting it. We can get the tidal prediction from the NOAA Tides and Currents web site. This web page allows putting in a range of dates and downloading a text file with tide predictions for the given range of dates.

The resulting text file looks as follows:

```
NOAA/NOS/CO-OPS
```

```
Disclaimer: These data are based upon the latest information available as of the  
           date of your request, and may differ from the published tide tables.
```

```
Daily Tide Predictions
```

```
StationName: EL JOBEAN, MYAKKA RIVER
```

```
State: FL
```

```
Stationid: 8725769
```

```
...
```

```
Date           Day      Time      Pred      High/Low  
2024/04/01     Mon      04:30    -0.19     L  
2024/04/01     Mon      20:07     1.91     H
```

```
...
```

This data is *almost* in CSV format, but a few quirks make it difficult to process. Here are some complicating factors:

- The file has 19 lines of data before the useful column heading line.
- The columns use the tab character `\t` as a delimiter instead of a comma.
- The heading row for the relevant data has some extraneous whitespace hidden in it.

The following function will provide clean CSV rows for further processing:

```
import csv  
from collections.abc import Iterator  
from typing import TextIO  
  
def tide_table_reader(source: TextIO) -> Iterator[dict[str, str]]:  
    line_iter = iter(source)
```

```

for line in line_iter:
    if len(line.rstrip()) == 0:
        break
header = next(line_iter).rstrip().split('\t')
del header[1] # Extra tab in the header
reader = csv.DictReader(line_iter, fieldnames=header, delimiter='\t')
yield from reader

```

The `Extra tab` in the header comment handles the heading, which has an extra whitespace character in it. This header row has two `\t` characters between the Date and Day column names:

```
'Date \t\tDay\tTime\tPred\tHigh/Low\n'
```

See the *Slicing and dicing a list* recipe in *Chapter 4* for more on this technique for removing an item from a list.

This list of column names can be used to build a `DictReader` instance to consume the rest of the data. (See the *Reading delimited files with the CSV module* recipe in *Chapter 11* for more on CSV files.)

We can convert each from a dictionary to a class instance using **Pydantic** validation features.

## How to do it...

The core data model will validate rows of data, creating an instance of a class. We can add features to this class to handle the application-specific processing. Here's how we build this class:

1. Start with the imports for the the data types within each row, plus the `BaseModel` class and some related classes:

```

import datetime
from enum import StrEnum
from typing import Annotated
from pydantic import BaseModel, Field, PlainValidator

```

2. Define the domain of values for the High/Low column. The two codes are enumerated as an Enum subclass:

```
class HighLow(StrEnum):
    high = "H"
    low = "L"
```

3. Since the date text is not in the default format used by **Pydantic**, we need to define a validation function that will produce a date object from the given string:

```
def validate_date(v: str | datetime.date) -> datetime.date:
    match v:
        case datetime.date():
            return v
        case str():
            return datetime.datetime.strptime(v, "%Y/%m/%d").date()
        case _:
            raise TypeError("can't validate {v!r} of type {type(v)}")
```

The Pydantic validators can be used on internal Python objects as well as strings from source CSV files or JSON documents. When applied to a `datetime.date` object, no additional conversion is needed.

4. Define the model. The `validation_alias` parameter of the Field definition will pluck data from a source field in the dictionary that's not exactly the same as the target attribute name in the class:

```
class TideTable(BaseModel):
    date: Annotated[
        datetime.date,
        Field(validation_alias='Date '),
        PlainValidator(validate_date)]
    day: Annotated[
        str, Field(validation_alias='Day')]
    time: Annotated[
        datetime.time, Field(validation_alias='Time')]
    prediction: Annotated[
        float, Field(validation_alias='Pred')]
    high_low: Annotated[
```

```
HighLow, Field(validation_alias='High/Low')]
```

Each field uses an Annotated type to define the base type, and additional details required to validate strings and convert them to that type.

The day field – with the day of the week – is not actually useful. It’s derived data from the date. For debugging purposes, this is preserved.

Given this class, we can use it to validate model instances from a sequence of dictionary instances. It looks like this:

```
>>> tides = [TideTable.model_validate(row) for row in dict_rows]
>>> tides[0]
TideTable(date=datetime.date(2024, 4, 1), day='Mon', time=datetime.time(4,
30), prediction=-0.19, high_low=<HighLow.low: 'L'>)
>>> tides[-1]
TideTable(date=datetime.date(2024, 4, 30), day='Tue', time=datetime.time(19,
57), prediction=1.98, high_low=<HighLow.high: 'H'>)
```

This sequence of objects contains too much data. We can use **Pydantic** to also filter the data, and pass only the useful rows. We’ll do this by revising this class definition and creating an alternative that includes the rules for the data to be passed.

## How it works...

The BaseModel class includes a number of operations that work with the annotated type hints of the class attributes. Consider this type hint:

```
date: Annotated[
    datetime.date,
    Field(validation_alias='Date '),
    PlainValidator(validate_date)]
```

This provides a base type, `datetime.date`. It provides a `Field` object that will extract the field named `'Date '` from a dictionary, and apply validation rules to it. Finally, the `PlainValidator` object provides a one-step validation rule that’s applied to the source data.

The `validate_date()` function was written to accept date objects as already valid, and convert string objects into date objects. This allows the validation to be used for raw data as well as Python objects.

Our application involves some narrowing of the domains of data for this example. There are three important criteria:

- We're only interested in high tide predictions.
- We'd prefer the tide be at least 1.5 (45 cm) feet above baseline.
- We need this to occur after 10:00 and before 17:00.

We can leverage Pydantic to perform additional validations to narrow the data domains. These additional validations can reject high tides less than the minimum of 1.5 feet.

## There's more...

We can extend this model to add validation rules that narrow the domain of valid rows to those that match our selection criteria based on time of day and height of tide. We'll be applying these more narrow data validation rules after any data conversions. These rules will raise `ValidationError` exceptions. This expands the imports from the pydantic package.

We'll define a number of additional validation functions. Here's a validator that raises an exception for low-tide data:

```
    BaseModel, Field, PlainValidator, AfterValidator, ValidationError
)

def pass_high_tide(hl: HighLow) -> HighLow:
    assert hl == HighLow.high, f"rejected low tide"
    return hl
```

The `assert` statement is elegantly simple for this task. This can also be done with `if` and `raise`.

A similar validator can raise an exception for data outside the acceptable time window:

```
def pass_daylight(time: datetime.time) -> datetime.time:
    assert datetime.time(10, 0) <= time <= datetime.time(17, 0)
    return time
```

Finally, we can combine these additional validators into the annotated type definitions:

```
class HighTideTable(BaseModel):
    date: Annotated[
        datetime.date,
        Field(validation_alias='Date '),
        PlainValidator(validate_date)]
    time: Annotated[
        datetime.time,
        Field(validation_alias='Time'),
        AfterValidator(pass_daylight)] # Range check
    prediction: Annotated[
        float,
        Field(validation_alias='Pred', ge=1.5)] # Minimum check
    high_low: Annotated[
        HighLow,
        Field(validation_alias='High/Low'),
        AfterValidator(pass_high_tide)] # Required value check
```

The additional validators will reject data where the criteria don't match our narrow requirements. The output will only have high tides, greater than 1.5 feet, and during daylight hours.

This data forms a sequence of `HighTideTable` instances, like the following:

```
>>> from pathlib import Path
>>> data = Path("data") / "tide-table-2024.txt"
>>> with open(data) as tide_file:
...     for ht in high_tide_iter(tide_table_reader(tide_file)):
...         print(repr(ht))
HighTideTable(date=datetime.date(2024, 4, 7), time=datetime.time(15, 42),
prediction=1.55, high_low=<HighLow.high: 'H'>)
...
HighTideTable(date=datetime.date(2024, 4, 10), time=datetime.time(16, 42),
prediction=2.1, high_low=<HighLow.high: 'H'>)
...
```

```
HighTideTable(date=datetime.date(2024, 4, 26), time=datetime.time(16, 41),
prediction=2.19, high_low=<HighLow.high: 'H'>)
```

We've omitted some rows to show just the first row, a row from the middle, and the last row. These are `HighTideTable` objects with attributes that are Python objects, suitable for further analysis and processing.

The general approach to **Pydantic** design means individual rules for combining raw data fields, converting data, and filtering data are all separated. We can confidently change one of these rules without having to worry about breaking other parts of the application.

This recipe included three varieties of checks:

- A range check to be sure a continuous value is within the allowed range. `AfterValidator` is used to make sure a string is converted to a time.
- A minimum check to be sure a continuous value is above a limit. For numbers, this can be done by the `Field` definition directly.
- A required value check to be sure a discrete value has one of the required values. `AfterValidator` is used to make sure a string is converted the enumerated type.

These kinds of checks are performed after the essential type matching and used to apply narrower validation rules.

## See also

- In *Chapter 11* we'll look more deeply at reading files of data.
- See the *Implementing more strict type checks with Pydantic* recipe for additional examples of using **Pydantic**. `Pydantic` uses compiled Python extensions to apply the validation rules with little overhead.



## Join our community **Discord** space

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

<https://packt.link/dHrHU>



# 11

## Input/Output, Physical Format, and Logical Layout

Computing often works with persistent data. There may be source data to be analyzed, or output to be created using Python input and output operations. The map of the dungeon that's explored in a game is data that will be input to the game application. Images, sounds, and movies are data output by some applications and input by other applications. Even a request through a network will involve input and output operations. The common aspect to all of these is the concept of a file of data. The term **file** is overloaded with many meanings:

- The **operating system (OS)** uses a file as a way to organize bytes of data on a device. It's the responsibility of application software to make sense of the bytes. Two common kinds of devices offer variations in terms of the features of OS files:
  - **Block devices** such as disks or **solid-state drives (SSDs)**: A file on this kind of device can seek any specific byte, making them particularly good for databases, where any row can be processed at any time.

- **Character devices** such as a network connection, a keyboard, or a GPS antenna. A file on this kind of device is viewed as a stream of individual bytes in transit. There's no way to seek forward or backward; the bytes must be captured in a buffer and processed as they arrive.
- The word *file* also defines a data structure used by the Python runtime. A uniform Python file abstraction wraps the various OS file implementations. When we open a Python file, there is a binding between the Python abstraction, an OS implementation, and the underlying collection of bytes on a block device or stream of bytes of a character device.

Python gives us two common modes for working with a file's content:

- In “b” (**binary**) mode, our application sees the bytes, without further interpretation. This can be helpful for processing media data like images, audio, and movies, which have complex encodings. We'll often import libraries like `pillow` to handle the details of image file encoding into bytes and decoding from bytes.
- In “t” (**text**) mode, the bytes of the file are encodings of string values. Python strings are made of Unicode characters, and there are a variety of schemes for decoding bytes into text and encoding text into bytes. Generally, the OS has a preferred encoding and Python respects this. The UTF-8 encoding is popular. Pragmatically, a file can have any of the available Unicode encodings, and it may not be obvious which encoding was used to create a file.

Additionally, Python modules like `shelve` and `pickle` have unique ways of representing more complex Python objects than simple strings. There are a number of `pickle` protocols available; all of them are based on binary mode file operations.

Throughout this chapter, we'll talk about how Python objects are **serialized**. Serialization creates a representation of the Python object's state as a series of bytes. Deserialization is the reverse process: it recovers a Python object's state from the bytes of a file. Saving and transferring a representation of the object state is the foundational concept behind REST web services.

When we process data from files, we have two common concerns:

- **The physical format of the data:** We need to know how the bytes on the file are interpreted to reconstruct a Python object. The bytes could represent a JPEG-encoded image or an MPEG-encoded movie. One very common example is the bytes of the file representing Unicode text, organized into lines. Generally, physical format concerns are handled by Python libraries like `csv`, `json`, and `pickle`, among many others.
- **The logical layout of the data:** A given data collection may have flexible positions for storing data items. The arrangement of CSV columns or JSON fields can vary. In cases where the data includes labels, the logical layout is clear. Without labels, the layout is positional, and some additional schema information is required to identify which data items occupy the various positions.

Both the physical format decoding and logical layout schema are essential to interpreting the data on a file. We'll look at a number of recipes for working with different physical formats. We'll also look at ways to divorce our program from some aspects of the logical layout.

In this chapter, we'll look at the following recipes:

- *Using `pathlib` to work with filenames*
- *Replacing a file while preserving the previous version*
- *Reading delimited files with the `CSV` module*
- *Using `dataclasses` to simplify working with `CSV` files*
- *Reading complex formats using regular expressions*
- *Reading `JSON` and `YAML` documents*
- *Reading `XML` documents*
- *Reading `HTML` documents*

In order to work with files, we'll start with objects that help control the OS filesystem. The common features of the directory structure of files and devices are described by Python's

`pathlib` module. This module has consistent behavior across a number of operating systems, allowing a Python program to work similarly on Linux, macOS, and Windows.

## Using `pathlib` to work with filenames

Most operating systems use a hierarchical tree of directories that contain files. The path from the root directory to a specific file is often shown as a string. Here's an example path:

```
/Users/slott/Documents/Writing/Python Cookbook/src/ch11/recipe_01.py
```

This full path name lists seven named directories contained in the (unnamed) root directory. The final name has a stem of `recipe_01` and a suffix of `.py`.

We can represent this as a string, and parse the string to locate directory names, file stems, and suffix strings. Doing this isn't portable between the macOS and Linux operating systems, which use `/` for a separator, and Windows, which uses `\` for a separator. Further, Windows files may also have device names as a prefix to the path.

Dealing with edge cases like `/` in a filename or `.` in a directory name can make string processing needlessly difficult. We can simplify parsing and many filesystem operations by using `pathlib.Path` objects instead of strings.

## Getting ready

It's important to separate three concepts:

- A path that identifies a file, including the name
- Metadata for a file – like creation timestamps and ownership – kept in the directory tree
- The contents of the file

The contents of the files are independent of the directory information. It's common for multiple directory entries to be linked to the same content. This can be done with *hard* links, where the directory information is shared among multiple paths, and *soft* links,

where a special kind of file contains a reference to another file.

Often, a filename has a suffix (or extension) used as a hint as to what the physical format is. A filename ending in `.csv` is likely a text file that can be interpreted as rows and columns of data. This binding between name and physical format is not absolute. File suffixes are only a hint and can be wrong.

In Python, the `pathlib` module handles all path-related processing. The module makes several distinctions among paths:

- Pure paths that may or may not refer to an actual file
- Concrete paths that are resolved; these refer to an actual file

This distinction allows us to create pure paths for files that our application will possibly create or refer to. We can also create concrete paths for those files that actually exist on the OS. An application often resolves a pure path to a concrete path.

While the `pathlib` module can make a distinction between Linux path objects and Windows path objects, this distinction is rarely needed. An important reason for using `pathlib` is because we want processing that is isolated from the details of the underlying OS.

All of the mini recipes in this section will leverage the following:

```
>>> from pathlib import Path
```

We'll also presume the `argparse` module is used to gather the file or directory names. For more information on `argparse`, see the *Using `argparse` to get command-line input* recipe in *Chapter 6*. We'll use an options variable as a namespace that contains the input filename or directory name that the recipe works with. As an example, we'll use the following Namespace object:

```
>>> from argparse import Namespace
>>> options = Namespace(
...     input='/path/to/some/file.csv',
```

```
...     file1='data/ch11_file1.yaml',  
...     file2='data/ch11_file2.yaml',  
... )
```

Frequently, we'll define `argparse` options to use `type=Path` so that the argument parsing creates `Path` objects for us. For the purposes of showing how `Path` objects work, the path information is provided as string values.

## How to do it...

We'll show a number of common pathname manipulations in the following mini-recipes:

- *Making the output filename by changing the input filename's suffix*
- *Making a number of sibling output files with distinct names*
- *Comparing file dates to see which is newer*
- *Finding all files that match a given pattern*

The first two reflect techniques for working with the path of directories to a file; using a `Path` object is much easier than doing sophisticated string manipulation. The last two gather information about concrete paths and the related files on a computer.

### Making the output filename by changing the input filename's suffix

Perform the following steps to create the output filename from an input filename by changing the input name's suffix:

1. Create a `Path` object from the input filename string:

```
>>> input_path = Path(options.input)  
>>> input_path  
PosixPath('/path/to/some/file.csv')
```

The `PosixPath` class is displayed because the author is using macOS. On a Windows machine, the class would be `WindowsPath`.

2. Create the output `Path` object using the `with_suffix()` method:

```
>>> output_path = input_path.with_suffix('.out')
>>> output_path
PosixPath('/path/to/some/file.out')
```

All of the filename parsing is handled seamlessly by the Path class. This doesn't create the concrete output file; it merely creates a new Path for it.

## Making a number of sibling output files with distinct names

Perform the following steps to make a number of sibling output files with distinct names:

1. Create a Path object from the input filename string:

```
>>> input_path = Path(options.input)
```

2. Extract the parent directory and the stem from the filename. The stem is the name without the suffix:

```
>>> input_directory = input_path.parent
>>> input_stem = input_path.stem
```

3. Build the desired output name. For this example, we'll append `_pass` to the stem and build the complete Path object:

```
>>> output_stem_pass = f"{input_stem}_pass"
>>> output_stem_pass
'file_pass'
```

```
>>> output_path = (
...     input_directory / output_stem_pass
... ).with_suffix('.csv')
>>> output_path
PosixPath('/path/to/some/file_pass.csv')
```

The `/` operator assembles a new Path from Path components. We need to put the `/` operation in parentheses to be sure that it's performed first, to create a new Path object



before changing the suffix.

## Comparing file dates to see which is newer

The following are the steps to see newer file dates by comparing them:

1. Create the Path objects from the input filename strings. The Path class will properly parse the string to determine the elements of the path:

```
>>> file1_path = Path(options.file1)
>>> file2_path = Path(options.file2)
```

When exploring this example, be sure the names in the options object are actual files.

2. Use the `stat()` method of each Path object to get timestamps for the file. Within the stat object, the `st_mtime` attribute provides the most recent modification time for the file:

```
>>> file1_path.stat().st_mtime
1572806032.0
>>> file2_path.stat().st_mtime
1572806131.0
```

The values are timestamps measured in seconds. Your values will depend on the files on your system. If we want a timestamp that seems sensible to most people, we can use the `datetime` module to create a more useful object from this:

```
>>> import datetime
>>> mtime_1 = file1_path.stat().st_mtime
>>> datetime.datetime.fromtimestamp(mtime_1)
datetime.datetime(2019, 11, 3, 13, 33, 52)
```

We can use any of a number of methods to format the datetime object.

## Finding all files that match a given pattern

The following are the steps to find all the files that match a given pattern:

1. Create the Path object from the input directory name:

```
>>> directory_path = Path(options.file1).parent
>>> directory_path
PosixPath('data')
```

2. Use the `glob()` method of the Path object to locate all files in this directory that match a given pattern. For non-existent directories, the iterator will be empty. Using `**` as part of the pattern will recursively walk the directory tree:

```
>>> from pprint import pprint
>>> pprint(sorted(directory_path.glob("*.csv")))
[PosixPath('data/binned.csv'),
```

```
...
```

```
"""
```

We've elided a number of the files in the results.

The `glob()` method is an iterator, and we've used the `sorted()` function to consume the values from this iterator and create a single list object from them.

## How it works...

Inside the OS, the sequence of directories to find a file is a path through the filesystem. In some cases, a simple string representation can be used to summarize the path. The string representation, however, makes many kinds of path operations into complex string parsing problems. A string is an unhelpfully opaque abstraction for working with OS paths.

The Path class definition simplifies operations on paths. These attributes, methods, and operators on a Path instance include the following examples:

- `.parent` extracts the parent directory.
- `.parents` enumerates all the enclosing directories.
- `.name` is the final name.

- `.stem` is the stem of the final name (without any suffix).
- `.suffix` is the final suffix.
- `.suffixes` is the sequence of suffix values, used with `file.tag.gz` kinds of names.
- The `.with_suffix()` method replaces the suffix of the file with a new suffix.
- The `.with_name()` method replaces the name in the path with a new name.
- The `/` operator builds `Path` objects from `Path` and string components.

A concrete path represents an actual filesystem resource. For concrete path objects, we can do a number of additional manipulations of the directory information:

- Determine what kind of directory entry this is; that is, an ordinary file, a directory, a link, a socket, a named pipe (or FIFO), a block device, or a character device.
- Get the directory details, including information such as timestamps, permissions, ownership, size, and so on.
- Unlink (that is, remove) the directory entry. Note that unlinking ordinary files is distinct from removing an empty directory. We'll look at this in the *There's more...* section of this recipe.
- Rename the file to place it in a new path. We'll also look at this in the *There's more...* section of this recipe.

Just about anything we might want to do with directory entries for files can be done with the `pathlib` module. The few exceptions are part of the `os` module, because they are generally OS-specific.

## There's more...

In addition to manipulating the path and gathering information about a file, we can also make some changes to the filesystem. Two common operations are renaming a file and unlinking (or removing) a file. We can use a number of methods to make changes to the filesystem:

- The `.unlink()` method removes ordinary files. It doesn't remove directories.
- The `.rmdir()` method removes empty directories. Removing a directory with files requires a two-step operation to first unlink all the files in the directory, then remove the directory.
- The `.rename()` method renames a file to a new path.
- The `.replace()` method replaces a file without raising an exception if the target already exists.
- The `.symlink_to()` method creates a soft link file with a link to an existing file.
- The `.hardlink_to()` method creates an OS hard link; two distinct directory entries will now own the underlying file content.

We can open a `Path` either using the built-in `open()` function or the `open()` method. Some people like to see `open(some_path)`, where others prefer `some_path.open()`. Both do the same thing: create an open file object.

We can create directories using the `mkdir()` method. There are two keyword parameters for this method:

- `exist_ok=False` is the default; if the directory already exists, an exception is raised. Changing this to `True` makes the code tolerant of an existing directory.
- `parents=False` is the default; the parents are not created, only the child-most directory in the path. Changing this to `True` will create the entire path, parents and children.

We can also read and write files as large string or bytes objects:

- The `.read_text()` method reads a file as a single string.
- The `.write_text()` method creates or replaces a file with the given string.
- The `.read_bytes()` method reads a file as a single bytes instance.
- The `.write_bytes()` method creates or replaces a file with the given bytes.

There are yet more file system operations, like changing ownership or changing permissions. These operations are available in the `os` module.

## See also

- In the *Replacing a file while preserving the previous version* recipe, later in this chapter, we'll look at how to leverage the features of a `Path` object to create a temporary file and then rename the temporary file to replace the original file.
- In the *Using `argparse` to get command-line input* recipe in *Chapter 6*, we looked at one very common way to use a string to create a `Path` object.
- The `os` module offers a number of filesystem operations that are less commonly used than the ones provided by `pathlib`.

## Replacing a file while preserving the previous version

We can leverage the power of the `pathlib` module to support a variety of filename manipulations. In the *Using `pathlib` to work with filenames* recipe in this chapter, we looked at a few of the most common techniques for managing directories, filenames, and file suffixes.

One common file processing requirement is to create output files in a fail-safe manner; that is, the application should preserve any previous output file, no matter how or where the application fails.

Consider the following scenario:

- 1 At time  $T_0$ , there's a valid output `.csv` file from a previous run of the `long_complex.py` application.
- 2 At time  $T_1$ , we start running the `long_complex.py` application using new data. It begins overwriting the output `.csv` file. Until the program finishes, the bytes will be unusable.
- 3 At time  $T_2$ , the application crashes. The partial contents of the output `.csv` file are

useless. Worse, the valid file from time  $T_0$  is no longer available either because it was overwritten.

In this recipe, we'll look at an approach to creating output files that's safe in the event of a failure.

## Getting ready

For files that don't span across physical devices, fail-safe file output generally means creating a new copy of the file using a temporary name. If the new file can be created successfully, then the old file should be replaced using a single, atomic rename operation.

We want to have the following features:

- The important output file must be preserved in a valid state at all times.
- A temporary version of the file is written by the application. There are a variety of conventions for naming this file. Sometimes, extra characters such as `~` or `#` are placed on the filename to indicate that it's a temporary, working file; for example, output `.csv~`. We'll use a longer suffix, `.new`; for example, output `.csv.new`.
- The previous version of the file is also preserved. Sometimes, the previous version has a suffix of `.bak`, meaning "backup." We'll use a longer suffix and call it output `.csv.old`. This also means any previous `.old` file must be removed as part of finalizing the output; only a single version is preserved.

To create a concrete example, we'll work with a file that has a very small but precious piece of data: a sequence of Quotient objects. Here's the definition for the Quotient class:

```
from dataclasses import dataclass, asdict, fields

@dataclass
class Quotient:
    numerator: int
    denominator: int
```

The following function will write an object to a file in CSV notation:

```

import csv
from collections.abc import Iterable
from pathlib import Path

def save_data(
    output_path: Path, data: Iterable[Quotient]
) -> None:
    with output_path.open("w", newline="") as output_file:
        headers = [f.name for f in fields(Quotient)]
        writer = csv.DictWriter(output_file, headers)
        writer.writeheader()
        for q in data:
            writer.writerow(asdict(q))

```

If a problem arises when writing the data object to the file, we could be left with a corrupted, unusable file. We'll wrap this function with another to provide a reliable write.

## How to do it...

We start creating a wrapper function by importing the classes we need:

1. Define a function to encapsulate the `save_data()` function along with a few extra features. The function signature is the same as the `save_data()` function:
2. Save the original suffix and create a new name with `.new` at the end of the suffix. This is a temporary file. If it is written properly, with no exceptions, then we can rename it so that it's the target file:

```

ext = output_path.suffix
output_new_path = output_path.with_suffix(f'{ext}.new')
save_data(output_new_path, data)

```

The `save_data()` function is the original process to create the new file being wrapped by this function.

3. Before replacing the previous file with the new, good file, remove any previous backup copy. We'll unlink an `.old` file, if one exists:

```
output_old_path = output_path.with_suffix(f'{ext}.old')
output_old_path.unlink(missing_ok=True)
```

4. Now, we can preserve any previous good file with the name of `.old`:

```
try:
    output_path.rename(output_old_path)
except FileNotFoundError as ex:
    # No previous file. That's okay.
    pass
```

5. The final step is to make the temporary `.new` file the official output:

```
try:
    output_new_path.rename(output_path)
except IOError as ex:
    # Possible recovery...
    output_old_path.rename(output_path)
```

This multi-step process uses two rename operations:

- Rename the previous version to a backup version with `.old` appended to the suffix.
- Rename the new version, which had `.new` appended to the suffix, to be the current version of the file.

A `Path` object has a `replace()` method. This always overwrites the target file, with no warning if overwriting an existing file. The choice between `rename()` and `replace()` depends on how our application needs to handle cases where old versions of files may be left in the filesystem. We've used `rename()` in this recipe to try and avoid overwriting files in the case of multiple problems.

Because these are applied serially, there's a tiny span of time between preserving the old file and renaming the new file where an application failure would fail to put a new file in place. We'll look at this in the next section.



## How it works...

This process involves three separate OS operations: an unlink and two renames. This is designed to ensure that an `.old` file is preserved and can be used to recover the previously good state.

Here's a timeline that shows the state of the various files. We've labeled the content as version 0 (some previous data), version 1 (the current, valid data), and version 2 (the newly created data):

Time	Operation	<code>.csv.old</code>	<code>.csv</code>	<code>.csv.new</code>
$T_0$		version 0	version 1	
$T_1$	Mid-creation	version 0	version 1	Will appear corrupt if used
$T_2$	Post-creation, closed	version 0	version 1	version 2
$T_3$	After unlinking <code>.csv.old</code>		version 1	version 2
$T_4$	After renaming <code>.csv</code> to <code>.csv.old</code>	version 1		version 2
$T_5$	After renaming <code>.csv.new</code> to <code>.csv</code>	version 1	version 2	

*Table 11.1: Timeline of file operations*

While there are several opportunities for failure, there's no ambiguity about which file is valid:

- If there's a `.csv` file, it's the current, valid file.
- If there's no `.csv` file, then the `.csv.old` file is a valid backup copy, which should be used for recovery. See the  $T_4$  moment in time, for this condition.

Since none of these operations involve actually copying the files, the operations are all extremely fast and reliable. They are, however, not guaranteed to work. The state of the filesystem can be changed by any user with the right permissions, leading to the need for care when creating new files that replace old files.

To ensure the output file is valid, some applications will take an additional step and write a final checksum row in the file to provide unambiguous evidence that the file is complete and consistent.

## There's more...

In some enterprise applications, output files are organized into directories with names based on timestamps. These operations can be handled gracefully by the `pathlib` module. We might, for example, have an archive directory for old files. This directory has date-stamped subdirectories for keeping temporary or working files.

We can then do the following to define a working directory:

```
archive_path = Path("/path/to/archive")

import datetime
from datetime import timezone

today = datetime.datetime.now(timezone.utc).strftime("%Y%m%d_%H%M%S")
working_path = archive_path / today
working_path.mkdir(parents=True, exist_ok=True)
```

The `mkdir()` method will create the expected directory. By including the `parents=True` argument, any needed parent directories will also be created. This can be handy to create the `archive_path` the very first time an application is executed. The `exists_ok=True` will avoid raising an exception if the archive directory already exists.

For some applications it can be appropriate to use the `tempfile` module to create temporary files. This module can create filenames that are guaranteed to be unique. This allows a complex server process to create temporary files without regard to filename conflicts.

## See also

- In the *Using `pathlib` to work with filenames* recipe, earlier in this chapter, we looked at the fundamentals of the `Path` class.
- In *Chapter 15*, we'll look at some techniques for writing unit tests that can ensure that parts of this recipe's example code will behave properly.
- In *Chapter 6*, the *Creating contexts and context managers* recipe shows additional details regarding working with the `with` statement to ensure file operations complete

properly, and that all of the OS resources are released.

- The `shutil` module provides a number of methods for copying files and directories full of files. This package reflects features of Linux shell programs like `cp`, as well as Windows programs like `copy` and `xcopy`.

## Reading delimited files with the CSV module

One commonly used data format is **comma-separated values (CSV)**. We can generalize this to think of the comma character as simply one of many candidate separator characters. For example, a CSV file can use the `|` character as the separator between columns of data. This generalization for separators other than the literal `,` makes CSV files particularly powerful.

How can we process data in one of the wide varieties of CSV formats?

### Getting ready

A summary of a file's content is called a schema. It's essential to distinguish between two aspects of a schema.

The physical format of a CSV file's bytes encode lines of text. For CSV files, the text is organized into rows and columns using a row separator character (or characters) and a column separator character. Many spreadsheet products will use `,` (comma) as the column separator and the `\r\n` sequence of characters as the row separator. The specific combination of punctuation characters in use is called the *CSV dialect*.

Additionally, column data can be quoted when it contains one of the separators. The most common quoting rules are to surround the column value with `"` characters. In order to include the quote character in column data, the quote character is doubled. For example, `"He said, ""Thanks."""`.

The logical layout of the data in the file is a sequence of data columns that are present. There are several common cases for handling the logical layout in CSV files:

- The file may have one line of headings. This fits nicely with the way the `csv` module

works. It can be even more helpful when the headings are also proper Python variable names. The schema is stated explicitly in the first line of the file.

- The file has no headings, but the column positions are fixed. In this case, we can impose headings on the file when we open it. Pragmatically, this involves some risk because it's difficult to confirm that the data meets the imposed schema.
- If the file has no headings and the column positions aren't fixed. In this case, additional external schema information is required to interpret the columns of data.

There are, of course, some common complications that can arise with any data. Some files are not in **First Normal Form (1NF)**. In 1NF, each row is independent of all other rows. When a file is not in this normal form, we'll need to add a generator function to rearrange the data into 1NF rows. See the *Slicing and dicing a list* recipe in *Chapter 4*, and the *Using stacked generator expressions* recipe in *Chapter 9*, for other recipes that show how to normalize data structures.

We'll look at a CSV file that has some real-time data recorded from the log of a sailboat. This is the `waypoints.csv` file. The data looks as follows:

```
lat,lon,date,time
32.8321666666667,-79.9338333333333,2012-11-27,09:15:00
31.6714833333333,-80.93325,2012-11-28,00:00:00
30.7171666666667,-81.5525,2012-11-28,11:35:00
```

This data contains four columns named in the first line of the file: `lat`, `lon`, `date`, and `time`. These describe a waypoint and need to be reformatted to create more useful information.

## How to do it...

Before starting to write any code, examine the data file to confirm the following features:

- The column separator character is `,`, which is the default.
- The row separator characters are `\r\n`, also widely used in both Windows and Linux.

- There is a single-row heading. If this isn't present, the headings should be provided separately when the reader object is created.

Once the format has been confirmed, we can start creating the needed functions as follows:

1. Import the `csv` module and the `Path` class:

```
import csv
from pathlib import Path
```

2. Define a `raw()` function to read raw data from a `Path` object that refers to the file:

```
def raw(data_path: Path) -> None:
```

3. Use the `Path` object to open the file in a `with` statement. Build the reader from the open file:

```
with data_path.open() as data_file:
    data_reader = csv.DictReader(data_file)
```

4. Consume (and process) the rows of data from the iterable reader. This is properly indented inside the `with` statement:

```
for row in data_reader:
    print(row)
```

The output from the `raw()` function is a series of dictionaries that look as follows:

```
{'lat': '32.8321666666667', 'lon': '-79.9338333333333', 'date':
'2012-11-27', 'time': '09:15:00'}
```

We can now process the data by referring to the columns as dictionary items, using syntax like, for example, `row['date']`. Using the column names is more descriptive than referring to the column by position; for example, `row[0]` is hard to understand.

To be sure that we're using the column names correctly, the `typing.TypedDict` type hint can be used to provide the expected column names.

## How it works...

The `csv` module handles the work of parsing the physical format. This separates the rows from each other, and also separates the columns within each row. The default rules ensure that each input line is treated as a separate row and that the columns are separated by `,`.

What happens when we need to use the column separator character as part of data? We might have data like this:

```
lan,lon,date,time,notes
32.832,-79.934,2012-11-27,09:15:00,"breezy, rainy"
31.671,-80.933,2012-11-28,00:00:00,"blowing ""like stink"""
```

The `notes` column has data in the first row, which includes the `,` column separator character. The rules for CSV allow a column's value to be surrounded by quotes. By default, the quoting characters are `"`. Within these quoting characters, the column and row separator characters are ignored.

In order to embed the quote character within a quoted string, the character is doubled. The second example row shows how the value `blowing "like stink"` is encoded by doubling the quote characters when they are part of the value of a column.

The values in a CSV file are always strings. A string value like `7331` may look like a number to us, but it's always text when processed by the `csv` module. This makes the processing simple and uniform, but it can be awkward for our Python applications.

When data is saved from a manually prepared spreadsheet, the data may reveal the quirks of the desktop software's internal rules for data display. Data that is displayed as a date on the desktop software is stored as a floating-point number in the CSV file.

There are two solutions to the date-as-number problem. One is to add a column in the source spreadsheet to properly format the date data as a string. Ideally, this is done using ISO rules so that the date is represented in the `YYYY-MM-DD` format. The other solution is to recognize the spreadsheet date as a number of seconds past some epochal date. The epochal dates vary slightly with versions of various tools, but they're generally Jan 1, 1900.

(A few spreadsheet applications used Jan 1, 1904.)

## There's more...

As we saw in the *Combining the map and reduce transformations* recipe in *Chapter 9*, there's often a pipeline of processing that includes cleaning and transforming the source data. This idea of stacked generator functions lets a Python program process large volumes of data. Reading one row at a time can avoid reading all the data into a vast, in-memory list. In this specific example, there are no extra rows that need to be eliminated. However, each column needs to be converted into something more useful.

In *Chapter 10*, a number of recipes use **Pydantic** to perform these kinds of data conversions. See the *Implementing more strict type checks with Pydantic* recipe for an example of this alternative approach.

To transform the data into a more useful form, we'll define a row-level cleansing function. A function can apply this cleansing function to each row of the source data.

In this case, we'll create a dictionary object and insert additional values that are derived from the input data. The core type hints for this Waypoint dictionary are these:

```
import datetime
from typing import TypeAlias, Any

Raw: TypeAlias = dict[str, Any]

Waypoint: TypeAlias = dict[str, Any]
```

Based on this definition of a Waypoint type, a `clean_row()` function can look like this:

```
def clean_row(
    source_row: Raw
) -> Waypoint:
    ts_date = datetime.datetime.strptime(
        source_row["date"], "%Y-%m-%d").date()
    ts_time = datetime.datetime.strptime(
        source_row["time"], "%H:%M:%S").time()
```

```
    return dict(
        date=source_row["date"],
        time=source_row["time"],
        lat=source_row["lat"],
        lon=source_row["lon"],
        lat_lon=(
            float(source_row["lat"]),
            float(source_row["lon"])
        ),
        ts_date=ts_date,
        ts_time=ts_time,
        timestamp = datetime.datetime.combine(
            ts_date, ts_time
        )
    )
```

The `clean_row()` function creates several new column values from the raw string data. The column named `lat_lon` has a two-tuple with proper floating-point values instead of strings. We've also parsed the date and time values to create `datetime.date` and `datetime.time` objects, respectively. We've combined the date and time into a single, useful value, which is the value of the `timestamp` column.

Once we have a row-level function for cleaning and enriching our data, we can map this function to each row in the source data. We can use `map(clean_row, reader)` or we can write a function that embodies this processing loop:

```
def cleanse(reader: csv.DictReader[str]) -> Iterator[Waypoint]:
    for row in reader:
        yield clean_row(row)
```

This can be used to provide more useful data from each row:

```
def display_clean(data_path: Path) -> None:
    with data_path.open() as data_file:
        data_reader = csv.DictReader(data_file)
        clean_data_reader = cleanse(data_reader)
        for row in clean_data_reader:
```



```
pprint(row)
```

These cleansed and enriched rows look as follows:

```
>>> data = Path("data") / "waypoints.csv"
>>> display_clean(data)
{'date': '2012-11-27',
 'lat': '32.8321666666667',
 'lat_lon': (32.8321666666667, -79.9338333333333),
 'lon': '-79.9338333333333',
 'time': '09:15:00',
 'timestamp': datetime.datetime(2012, 11, 27, 9, 15),
 'ts_date': datetime.date(2012, 11, 27),
 'ts_time': datetime.time(9, 15)}
...
```

The new columns such as `lat_lon` have proper numeric values instead of strings. The `timestamp` value has a full date-time value that can be used for simple computations of elapsed time between waypoints.

## See also

- See the *Combining the map and reduce transformations* recipe in *Chapter 9* for more information on the idea of a processing pipeline or stack.
- See the *Slicing and dicing a list* recipe in *Chapter 4*, and the *Using stacked generator expressions* recipe in *Chapter 9*, for more information on processing a CSV file that isn't in a proper *1NF*.
- For more information on the `with` statement, see the *Creating contexts and context managers* recipe in *Chapter 7*.
- In *Chapter 10*, a number of recipes use **Pydantic** to perform these kinds of data conversions. See the *Implementing more strict type checks with Pydantic* recipe for an example of this alternative approach.
- See <https://www.packtpub.com/product/learning-pandas-second-edition>

/9781787123137 Learning pandas for an approach to CSV files using the **pandas** framework.

## Using dataclasses to simplify working with CSV files

One commonly used data format is known as **Comma-Separated Values (CSV)**. Python's `csv` module has a very handy `DictReader` class definition. When a file contains a one-row header, the header row's values become keys that are used for all the subsequent rows. This allows a great deal of flexibility in the logical layout of the data. For example, the column ordering doesn't matter, since each column's data is identified by a name taken from the header row.

Using a dictionary forces us to write, for example, `row['lat']` or `row['date']` to refer to data in specific columns. The built-in `dict` class has no provision for derived data. If we switch to a dataclass, we have a number of benefits:

- Nicer attribute syntax like `row.lat` or `row.date`.
- Derived values can be lazy properties.
- A frozen dataclass is immutable, and the objects can be keys to dictionaries and members of sets.

How can we improve data access and processing using dataclasses?

### Getting ready

We'll look at a CSV file that has some real-time data recorded from the log of a sailboat. This file is the `waypoints.csv` file. For more information, see the *Reading delimited files with the CSV module* recipe in this chapter. The data looks as follows:

```
lat,lon,date,time
32.8321666666667,-79.9338333333333,2012-11-27,09:15:00
31.6714833333333,-80.93325,2012-11-28,00:00:00
```

```
30.7171666666667, -81.5525, 2012-11-28, 11:35:00
```

The first line contains a header that names the four columns, `lat`, `lon`, `date`, and `time`. The data can be read by a `csv.DictReader` object. We'd like to do more sophisticated work, so we'll create a `@dataclass` class definition encapsulating the data and the processing we need to do.

## How to do it...

We need to start with a dataclass that reflects the available data, and then we can use this dataclass with a dictionary reader:

1. Import the definitions from the various libraries that are needed:

```
from dataclasses import dataclass, field
import datetime
from collections.abc import Iterator
```

2. Define a dataclass narrowly focused on the input, precisely as it appears in the source file. We've called the class `RawRow`. In a complex application, a more descriptive name than `RawRow` would be appropriate. This definition of the attributes may change as the source file organization changes:

```
@dataclass
class RawRow:
    date: str
    time: str
    lat: str
    lon: str
```

As a practical matter, enterprise file formats are likely to change whenever new software versions are introduced. It's often helpful to formalize file schema as class definitions to facilitate unit testing and problem resolution when changes occur.

3. Define a second dataclass where objects are built from the source dataclass attributes. This second class is focused on the real work of the application. The source data is in

a single attribute, `raw`, in this example. Fields computed from this source data are all initialized with `field(init=False)` because they'll be computed after initialization:

```
@dataclass
class Waypoint:
    raw: RawRow
    lat_lon: tuple[float, float] = field(init=False)
    ts_date: datetime.date = field(init=False)
    ts_time: datetime.time = field(init=False)
    timestamp: datetime.datetime = field(init=False)
```

4. Add the `__post_init__()` method to eagerly initialize all the derived fields:

```
def __post_init__(self) -> None:
    self.ts_date = datetime.datetime.strptime(
        self.raw.date, "%Y-%m-%d"
    ).date()
    self.ts_time = datetime.datetime.strptime(
        self.raw.time, "%H:%M:%S"
    ).time()
    self.lat_lon = (
        float(self.raw.lat),
        float(self.raw.lon)
    )
    self.timestamp = datetime.datetime.combine(
        self.ts_date, self.ts_time
    )
```

5. Given these two dataclass definitions, we can create an iterator that will accept individual dictionaries from a `csv.DictReader` object and create the needed `Waypoint` objects. The intermediate representation, `RawRow`, is a convenience so that we can assign attribute names to the source data columns:

```
def waypoint_iter(reader: csv.DictReader[str]) -> Iterator[Waypoint]:
    for row in reader:
        raw = RawRow(**row)
        yield Waypoint(raw)
```

The `waypoint_iter()` function creates `RawRow` objects from the input dictionary, and then

creates the final Waypoint objects from the RawRow instances. This two-step process is helpful for isolating code changes to the source or the processing.

We can use the following function to read and display the CSV data:

```
def display(data_path: Path) -> None:
    with data_path.open() as data_file:
        data_reader = csv.DictReader(data_file)
        for waypoint in waypoint_iter(data_reader):
            pprint(waypoint)
```

## How it works...

The source dataclass, the RawRow class in this example, is designed to match the input document. The field names and types match the CSV input types. Because the names match, the RawRow(\*\*row) expression will create an instance of the RawRow class from the DictReader dictionary.

From this initial, or raw, data, we can derive the more useful data, as shown in the Waypoint class definition. The \_\_post\_init\_\_() method transforms the initial value in the self.raw attribute into a number of more useful attribute values.

This separation lets us manage the following two kinds of common changes to application software:

1. The source data can change because the spreadsheet was adjusted manually. This is common: a person may change column names or change the order of the columns.
2. The required computations may change as the application's focus expands or shifts. More derived columns may be added, or the algorithms may change.

It's helpful to disentangle the various aspects of a program so that we can let them evolve independently. Gathering, cleaning, and filtering source data is one aspect of this separation of concerns. The resulting computations are a separate aspect, unrelated to the format of the source data.

## There's more...

In many cases, the source CSV file will have headers that do not map directly to valid Python attribute names. In these cases, the keys present in the source dictionary must be mapped to the column names. This can be managed by expanding the `RawRow` class definition to include a `@classmethod` that builds the `RawRow` dataclass object from the source dictionary.

The following example defines a class called `RawRow_HeaderV2`. This definition reflects a variant spreadsheet with different column names in the header:

```
@dataclass
class RawRow_HeaderV2:
    date: str
    time: str
    lat: str
    lon: str

    @classmethod
    def from_csv(cls, csv_row: dict[str, str]) -> "RawRow_HeaderV2":
        return RawRow_HeaderV2(
            date = csv_row['Date of Travel (YYYY-MM-DD)'],
            time = csv_row['Arrival Time (HH:MM:SS)'],
            lat = csv_row['Latitude (degrees N)'],
            lon = csv_row['Logitude (degrees W)'],
```

The instances of the `RawRow_HeaderV2` class are built using the expression `RawRow_HeaderV2.from_csv(row)`. The objects are compatible with the `RawRow` class. Either of these classes of objects can also be transformed into `Waypoint` instances.

For an application that works with a variety of data sources, these kinds of “raw data transformation” dataclasses can be handy for mapping the minor variations in a logical layout to a consistent internal structure for further processing. As the number of input transformation classes grows, additional type hints are required. For example, the following type hint provides a common name for the variations in input format:

```
Raw: TypeAlias = RawRow | RawRow_HeaderV2
```

This type hint helps to unify the original `RawRow` and the alternative `RawRow_HeaderV2` types, which are alternative definitions with compatible features. The most important feature is the use of generators to process rows individually and avoid creating large list objects with **all** of the data.

## See also

- The *Reading delimited files with the CSV module* recipe, earlier in this chapter, also covers CSV file reading.
- In *Chapter 6*, the *Using dataclasses for mutable objects* recipe also covers ways to use Python's dataclasses.

## Reading complex formats using regular expressions

Many file formats lack the elegant regularity of a CSV file. One common file format that's rather difficult to parse is a web server log file. These files tend to have complex data without a single, uniform separator character or consistent quoting rules.

When we looked at a simplified log file in the *Writing generator functions with the yield statement* recipe in *Chapter 9*, we saw that the rows look as follows:

```
[2016-05-08 11:08:18,651] INFO in ch09_r09: Sample Message One
[2016-05-08 11:08:18,651] DEBUG in ch09_r09: Debugging
[2016-05-08 11:08:18,652] WARNING in ch09_r09: Something might have gone
wrong
```

There are a variety of punctuation marks being used in this file. The `csv` module can't parse this complexity.

We'd like to write programs with the elegant simplicity of CSV processing. This means

we'll need to encapsulate the complexities of log file parsing and keep this aspect separate from analysis and summary processing.

## Getting ready

Parsing a file with a complex structure generally involves writing a function that behaves somewhat like the `reader()` function in the `csv` module. In some cases, it can be easier to create a small class that behaves like the `DictReader` class.

The core feature of reading a complex file is a function that will transform one line of text into a dictionary or tuple of individual field values. Parts of this job can often be done by the `re` package.

Before we can start, we'll need to develop (and debug) the regular expression that properly parses each line of the input file. For more information on this, see the *String parsing with regular expressions* recipe in *Chapter 1*.

For this example, we'll use the following code. We'll define a pattern string with a series of regular expressions for the various elements of the line:

```
import re

pattern_text = (
    r"\[(?P<date>.*?)\]\s+"
    r"(?P<level>\w+)\s+"
    r"in\s+(?P<module>\S+?)"
    r":\s+(?P<message>.+)"
)

pattern = re.compile(pattern_text, re.X)
```

We've used the `re.X` option so that we can include extra whitespace in the regular expression. This can help to make it more readable by separating prefix and suffix characters.

When we write a regular expression, we wrap the interesting sub-strings to capture in `()`. After performing a `match()` or `search()` operation, the resulting `Match` object will have the captured text for the matched substrings. The `groups()` method of a `Match` object and



the `groupdict()` method of a `Match` object will provide the captured strings.

Here's how this pattern works:

```
>>> sample_data = '[2016-05-08 11:08:18,651] INFO in ch10_r09: Sample
Message One'

>>> match = pattern.match(sample_data)
>>> match.groups()
('2016-05-08 11:08:18,651', 'INFO', 'ch10_r09', 'Sample Message One')

>>> match.groupdict()
{'date': '2016-05-08 11:08:18,651', 'level': 'INFO', 'module': 'ch10_r09',
 'message': 'Sample Message One'}
```

We've provided a line of sample data in the `sample_data` variable. The resulting `Match` object has a `groups()` method that returns each of the interesting fields. The value of the `groupdict()` method of a match object is a dictionary, with the name provided in the `?P<name>` preface to the regular expression in brackets, `()`.

## How to do it...

This recipe is split into two mini-recipes. The first part defines a `log_parser()` function to parse a single line, while the second part applies the `log_parser()` function for each line of input.

### Defining the parse function

Perform the following steps to define the `log_parser()` function:

1. Define the compiled regular expression object:

```
import re

pattern_text = (
    r"\[(?P<date>.*?)\]\s+"
    r"(?P<level>\w+)\s+"
    r"in\s+(?P<module>\S+)"
    r":\s+(?P<message>.+)"
)
```

```
pattern = re.compile(pattern_text, re.X)
```

2. Define a class to model the resulting complex data object. This can have additional derived properties or other complex computations. Minimally, a `NamedTuple` must define the fields that are extracted by the parser. The field names should match the regular expression capture name in the `(?P<name>...)` prefix:

```
from typing import NamedTuple

class LogLine(NamedTuple):
    date: str
    level: str
    module: str
    message: str
```

3. Define a function that accepts a line of text as an argument and produces a parsed `LogLine` instance:

```
def log_parser(source_line: str) -> LogLine:
```

4. Apply the regular expression to create a match object. We've assigned it to the `match` variable and also checked to see it is not `None`:

```
if match := pattern.match(source_line):
```

5. When the value of `match` is not `None`, return a useful data structure with the various pieces of data from this input line:

```
data = match.groupdict()
return LogLine(**data)
```

6. When the match is `None`, either log the problem or raise an exception to stop processing:

```
raise ValueError(f"Unexpected input {source_line}")
```

## Using the log\_parser() function

This portion of the recipe will apply the `log_parser()` function to each line of the input file:

1. From the `pathlib` module, import useful class and function definitions:

```
>>> from pathlib import Path
>>> from pprint import pprint
```

2. Create the `Path` object that identifies the file:

```
>>> data_path = Path("data") / "sample.log"
```

3. Use the `Path` object to open the file in a `with` statement. Create the log file reader from the open file object, `data_file`. In this case, we'll use the built-in `map()` function to apply the `log_parser()` function to each line from the source file:

```
>>> with data_path.open() as data_file:
...     data_reader = map(log_parser, data_file)
```

4. Read (and process) the various rows of data. For this example, we'll print each row:

```
...     for row in data_reader:
...         pprint(row)
```

The output is a series of `LogLine` tuples that looks as follows:

```
LogLine(date='2016-06-15 17:57:54,715', level='INFO', module='ch09_r10',
message='Sample Message One')
LogLine(date='2016-06-15 17:57:54,715', level='DEBUG', module='ch09_r10',
message='Debugging')
LogLine(date='2016-06-15 17:57:54,715', level='WARNING', module='ch09_r10',
message='Something might have gone wrong')
```

We can do more meaningful processing on these tuple instances than we can on a line of raw text. These allow us to filter the data by severity level, or create a counter based on the module providing the message.

## How it works...

This log file is in **First Normal Form (1NF)**: the data is organized into lines that represent independent entities or events. Each row has a consistent number of attributes or columns, and each column has data that is atomic or can't be meaningfully decomposed further. Unlike CSV files, however, this particular format requires a complex regular expression to parse.

In our log file example, the timestamp contains a number of individual elements – year, month, day, hour, minute, second, and millisecond – but there's little value in further decomposing the timestamp. It's more helpful to use it as a single `datetime` object and derive details (like the hour of the day) from this object, rather than assembling individual fields into a new piece of composite data.

In a complex log processing application, there may be several varieties of message fields. It may be necessary to parse these message types using separate patterns. When we need to do this, it reveals that the various lines in the log aren't consistent in terms of the format and number of attributes, breaking one of the 1NF assumptions.

We've generally followed the design pattern from the *Reading delimited files with the CSV module* recipe, so that reading a complex log is nearly identical to reading a simple CSV file. Indeed, we can see that the primary difference lies in one line of code:

```
data_reader_csv = csv.DictReader(data_file)
```

Compare that to the following:

```
data_reader_logs = map(log_parser, data_file)
```

This parallel construct allows us to reuse analysis functions across many input file formats.

This allows us to create a library of tools that can be used on a number of data sources. It can help to make analytic applications resilient when data sources change.

## There's more...

One of the most common operations when reading very complex files is to rewrite them into an easier-to-process format. We'll often want to save data in the CSV format for later processing.

Some of this is similar to the *Managing multiple contexts with multiple resources* recipe in *Chapter 7*. This recipe shows multiple open file-processing contexts. We'll read from one file and write to another file.

The file writing process looks as follows:

```
import csv

def copy(data_path: Path) -> None:
    target_path = data_path.with_suffix(".csv")
    with target_path.open("w", newline="") as target_file:
        writer = csv.DictWriter(target_file, LogLine._fields)
        writer.writeheader()
        with data_path.open() as data_file:
            reader = map(log_parser, data_file)
            writer.writerows(row._asdict() for row in reader)
```

The first portion of this script defines a CSV writer for the target file. The path for the output file, `target_path`, is based on the input name, `data_path`. The suffix is changed to `.csv`.

The target file is opened with the newline character turned off by the `newline=''` option. This allows the `csv.DictWriter` class to insert newline characters appropriate for the desired CSV dialect.

A `DictWriter` object is created to write to the given file. The sequence of column headings is provided by the `LogLines` class definition. This makes sure the output CSV file will contain correct, consistent column names.

The `writerows()` method writes the column names as the first line of output. This makes reading the file slightly easier because the column names are provided. The first row of a CSV file can contain an explicit schema definition that shows what data is present.

The source file is opened, as shown in the preceding recipe. Because of the way the `csv` module writers work, we can provide the reader generator expression to the `writerows()` method of the writer. The `writerows()` method will consume all of the data produced by the reader generator. This will, in turn, consume all the rows produced by the open file.

We don't need to write any explicit `for` statements to ensure that all of the input rows are processed. The `writerows()` function makes this guarantee for us.

The output file looks as follows:

```
date,level,module,message
"2016-06-15 17:57:54,715",INFO,ch09_r10,Sample Message One
"2016-06-15 17:57:54,715",DEBUG,ch09_r10,Debugging
"2016-06-15 17:57:54,715",WARNING,ch09_r10,Something might have gone wrong
```

The file has been transformed from the rather complex input format into a simpler CSV format, suitable for further analysis and processing.

## See also

- For more information on the `with` statement, see the *Creating contexts and context managers* recipe in *Chapter 7*.
- The *Writing generator functions with the yield statement* recipe in *Chapter 9* shows other processing of this log format.
- In the *Reading delimited files with the CSV module* recipe, earlier in this chapter, we looked at other applications of this general design pattern.
- In the *Using dataclasses to simplify working with CSV files* recipe, earlier in this chapter, we looked at other sophisticated CSV processing techniques.

## Reading JSON and YAML documents

**JavaScript Object Notation (JSON)** is often used for serializing data. For details, see <http://json.org>. Python includes the `json` module in order to serialize and deserialize data in this notation.

JSON documents are used widely by web applications. It's common to exchange data between RESTful web clients and servers using documents in JSON notation. These two tiers of an application stack communicate via JSON documents sent via the HTTP protocol.

The YAML format is a more sophisticated and flexible extension to JSON notation. For details, see <https://yaml.org>. Any JSON document is also a valid YAML document. The reverse is not true: YAML syntax is more complex and includes constructs that are not valid JSON.

To use YAML, an additional module has to be installed:

```
(cookbook3) % python -m pip install pyyaml
```

The PyYAML project offers a `yaml` module that is popular and works well. See <https://pypi.org/project/PyYAML/>.

In this recipe, we'll use the `json` module to parse JSON format data in Python.

### Getting ready

We've gathered some sailboat racing results in `race_result.json`. This file contains information on the teams, the legs of the race, and the order in which the various teams finished each individual leg of the race. JSON handles this complex data elegantly.

An overall score can be computed by summing the finish position in each leg: the lowest score is the overall winner. In some cases, there are null values when a boat did not start, did not finish, or was disqualified from the race.

When computing the team's overall score, the null values are assigned a score of one more than the number of boats in the competition. If there are seven boats, then the team is

given eight points for their failure to finish, a hefty penalty.

The data has the following schema. There are two fields within the overall document:

- **legs**: An array of strings that shows the starting port and ending port.
- **teams**: An array of objects with details about each team. Within each teams object, there are several fields of data:
  - **name**: String team name.
  - **position**: An array of integers and nulls with a position. The order of the items in this array matches the order of the items in the **legs** array.

The data looks as follows:

```
{
  "teams": [
    {
      "name": "Abu Dhabi Ocean Racing",
      "position": [
        1,
        3,
        2,
        2,
        1,
        2,
        5,
        3,
        5
      ]
    },
    ...
  ],
  "legs": [
    "ALICANTE - CAPE TOWN",
    "CAPE TOWN - ABU DHABI",
    "ABU DHABI - SANYA",
    "SANYA - AUCKLAND",
    "AUCKLAND - ITAJA\u00cd",
    "ITAJA\u00cd - NEWPORT",
    "NEWPORT - LISBON",
  ]
}
```



```
"LISBON - LORIENT",  
"LORIENT - GOTHENBURG"  
]  
}
```

We've only shown the first team's details. There were a total of seven teams in this particular race. Each team is represented by a Python dictionary, with the team's name and their history of finish positions on each leg. For the team shown here, Abu Dhabi Ocean Racing, they finished in first place in the first leg, and then third place in the next leg. Their worst performance was fifth place in both the seventh and ninth legs of the race, which were the legs from Newport, Rhode Island, USA, to Lisbon, Portugal, and from Lorient, France, to Gothenburg, Sweden.

The JSON-formatted data can look like a Python dictionary that contains lists within it. This overlap between Python syntax and JSON syntax can be thought of as a happy coincidence: it makes it easier to visualize the Python data structure that will be built from the JSON source document.

JSON has a small set of data structures: null, Boolean, number, string, list, and object. These map to objects of Python types in a very direct way. The `json` module makes the conversions from source text into Python objects for us.

One of the strings contains a Unicode escape sequence, `\u00cd`, instead of the actual Unicode character Í. This is a common technique used to encode characters beyond the 128 ASCII characters. The parser in the `json` module handles this for us.

In this example, we'll write a function to disentangle this document and show the team finishes for each leg.

## How to do it...

This recipe will start by importing the necessary modules. We'll then use these modules to transform the contents of the file into a useful Python object:

1. We'll need the `json` module to parse the text. We'll also need a `Path` object to refer

to the file:

```
import json
from pathlib import Path
```

2. Define a `race\_summary()` function to read the JSON document from a given `Path` instance:

```
def race_summary(source_path: Path) -> None:
```

3. Create a Python object by parsing the JSON document. It's often easiest to use `source_path.read_text()` to read the file named by the `Path` object. We provided this string to the `json.loads()` function for parsing. For very large files, an open file can be passed to the `json.load()` function:

```
document = json.loads(source_path.read_text())
```

4. Display the data: The document object contains a dictionary with two keys, `teams` and `legs`. Here's how we can iterate through each leg, showing the team's position in the leg:

```
for n, leg in enumerate(document['legs']):
    print(leg)
    for team_finishes in document['teams']:
        print(
            team_finishes['name'],
            team_finishes['position'][n])
```

The data for each team will be a dictionary with two keys: `name` and `position`. We can navigate down into the team details to get the name of the first team:

```
>>> document['teams'][6]['name']
'Team Vestas Wind'
```

We can look inside the `legs` field to see the names of each leg of the race:

```
>>> document['legs'][5]
'ITAJAÍ - NEWPORT'
```

## How it works...

A JSON document is a data structure in JavaScript Object Notation. JavaScript programs can parse the document trivially. Other languages must do a little more work to translate the JSON to a native data structure.

A JSON document contains three kinds of structures:

- **Objects that map to Python dictionaries:** JSON has a syntax similar to Python: `{"key": "value", ...}`.
- **Arrays that map to Python lists:** JSON syntax uses `[item, ...]`, which is also similar to Python.
- **Primitive values:** There are five classes of values: string, number, true, false, and null. Strings are enclosed in " and use a variety of \ escape sequences, which are similar to Python's. Numbers follow the rules for floating-point values. The other three values are simple literals; these parallel Python's True, False, and None literals.

As a special case, numbers with no decimal point become Python int objects. This is an extension of the JSON standard.

There is no provision for any other kinds of data. This means that Python programs must convert complex Python objects into a simpler representation so that they can be serialized in JSON notation.

Conversely, we often apply additional conversions to reconstruct complex Python objects from the simplified JSON representation. The `json` module has places where we can apply additional processing to the simple structures to create more sophisticated Python objects.

## There's more...

A file, generally, contains a single JSON document. The JSON standard doesn't provide an easy way to encode multiple documents in a single file. If we want to analyze a web log, for example, the original JSON standard may not be the best notation for preserving a huge volume of information.

There are common extensions, like Newline Delimited JSON (<http://ndjson.org>) and JSON Lines, <http://jsonlines.org>, to define a way to encode multiple JSON documents into a single file.

While these approaches handle collections of documents, there is an additional problem that we often have to tackle: serializaing (and deserializing) complex objects, for example, `datetime` objects.

When we represent a Python object's state as a string of text characters, we've serialized the object's state. Many Python objects need to be saved in a file or transmitted to another process. These kinds of transfers require a representation of the object state. We'll look at serializing and deserializing separately.

## Serializing a complex data structure

The serialization to JSON works out the best if we create Python objects limited to values of the built-in types `dict`, `list`, `str`, `int`, `float`, `bool`, and the special type for `None`. This subset of Python types can be used to build objects the `json` module can serialize and can be used widely by a number of programs, written in different languages.

One commonly used data structure that doesn't serialize easily is the `datetime.datetime` object.

Avoiding the `TypeError` exception exceptions when trying to serialize an unusual Python object can be done in one of two ways. We can either convert the data into a JSON-friendly structure before building the document, or we can add a default type handler to the JSON serialization process that gives us a way to provide a serializable version of the data.

To convert a `datetime` object into a string prior to serializing it as JSON, we need to make

a change to the underlying data. It seems awkward to mangle the data or Python's data types because of a serialization concern.

The other technique for serializing complex data is to provide a function that's used by the `json` module during serialization. This function must convert a complex object into something that can be safely serialized. In the following example, we'll convert a `datetime` object into a simple string value:

```
def default_date(object: Any) -> Any:
    match object:
        case datetime.datetime():
            return {"$date$": object.isoformat()}
    return object
```

We've defined a function, `default_date()`, which will apply a special conversion rule to `datetime` objects. Any `datetime.datetime` instance will be replaced with a dictionary with an obvious key – `"$date$"` – and a string value. This dictionary can then be serialized by the `json` module.

We provide this serialization helper function to the `json.dumps()` function. This is done by assigning the `default_date()` function to the `default` parameter, as follows:

```
>>> example_date = datetime.datetime(2014, 6, 7, 8, 9, 10)
>>> document = {'date': example_date}
>>> print(
...     json.dumps(document, default=default_date, indent=2)
... )
{
  "date": {
    "$date$": "2014-06-07T08:09:10"
  }
}
```

When the `json` module can't serialize an object, it passes the object to the given default function, `default_date()`. In any given application, we'll need to expand this function to handle a number of Python object types that we might want to serialize in JSON notation.

If there is no default function provided, an exception is raised when an object can't be serialized.

## Deserializing a complex data structure

When deserializing JSON to create Python objects, there's a *hook* that can be used to convert data from a JSON dictionary into a more complex Python object. This is called `object_hook` and it is used during processing by the `json.loads()` function. This hook is used to examine each JSON dictionary to see if something else should be created from the dictionary instance.

The function we provide will either create a more complex Python object, or it will simply return the original dictionary object unmodified:

```
def as_date(object: dict[str, Any]) -> Any:
    if {'$date$'} == set(object.keys()):
        return datetime.datetime.fromisoformat(object['$date$'])
    return object
```

This function will check each object that's decoded to see if the object has a single field, and if that single field is named "\$date\$". If that is the case, the value of the entire object is replaced with a `datetime.datetime` object. The return type is a union of `Any` and `dict[str, Any]` to reflect the two possible results: either some object or the original dictionary.

We provide a function to the `json.loads()` function using the `object_hook` parameter, as follows:

```
>>> source = '{"date": {"$date$": "2014-06-07T08:09:10"}}'
>>> json.loads(source, object_hook=as_date)
{'date': datetime.datetime(2014, 6, 7, 8, 9, 10)}
```

This parses a very small JSON document. All objects are provided to the `as_date()` object hook. Of these objects, one dictionary meets the criteria for containing a date. A Python

object is built from the string value found in the JSON serialization.

The **Pydantic** package offers a number of serialization features. Recipes are shown in *Chapter 10* for working with this package.

## See also

- The *Reading HTML documents* recipe, later in this chapter, will show how we prepared this data from an HTML source.
- The *Implementing more strict type checks with Pydantic* recipe in *Chapter 10* covers some features of the **Pydantic** package.

## Reading XML documents

The XML markup language is widely used to represent the state of objects in a serialized form. For details, see <http://www.w3.org/TR/REC-xml/>. Python includes a number of libraries for parsing XML documents.

XML is called a markup language because the content of interest is marked with tags, written with a start `<tag>` and an end `</tag>`, used to define the structure of the data. The overall file text includes both the content and the XML markup.

Because the markup is intermingled with the text, there are some additional syntax rules that must be used to distinguish markup from text. A document must use `&lt;` instead of `<`, `&gt;` instead of `>`, and `&amp;` instead of `&` in text. Additionally, `&quot;` is also used to embed a `"` character in an attribute value. For the most part, XML parsers will handle this transformation when consuming XML.

The example document, then, will have items as follows:

```
<team><name>Team SCA</name><position>...</position></team>
```

The `<team>` tag contains the `<name>` tag, which contains the text of the team's name. The `<position>` tag contains more data about the team's finish position in each leg of a race.

The overall document forms a large, nested collection of containers. We can think of a document as a tree with a root tag that contains all the other tags and their embedded content. Between tags, there can be additional content. In some applications, the additional content between the ends of tags is entirely whitespace.

Here's the beginning of the document we'll be looking at:

```
<?xml version="1.0"?>
<results>
  <teams>
    <team>
      <name>
        Abu Dhabi Ocean Racing
      </name>
      <position>
        <leg n="1">
          1
        </leg>
        ...
      </position>
      ...
    </team>
    ...
  </teams>
  <legs>
    <leg n="1">
      ALICANTE - CAPE TOWN
    </leg>
    ...
  </legs>
</results>
```

The top-level container is the `<results>` tag. Within this is a `<teams>` tag. Within the `<teams>` tag are many repetitions of data for each individual team, each enclosed in the `<team>` tag. We've used `...` to show where parts of the document were elided.

It's very, very difficult to parse XML with regular expressions. Regular expressions don't cope well with the kinds of recursion and repetition present in XML. We need more sophisticated parsers to handle the syntax of nested tags.



There are two binary libraries, part of the modules `xml.sax` and `xml.parsers.expat`, to parse XML. These have the advantage of being very fast.

In addition to these, there's a very sophisticated set of tools in the `xml.etree` package. We'll focus on using the `ElementTree` class in this package to parse and analyze XML documents. This has the advantage of offering a large number of useful features like XPath searching to find tags in a complicated document.

## Getting ready

We've gathered some sailboat racing results in `race_result.xml`. This file contains information on teams, legs, and the order in which the various teams finished each leg. For more information on this data, see the *Reading JSON and YAML documents* recipe in this chapter.

The root tag for this data is a `<results>` document. This has the following schema:

- The `<legs>` tag contains individual `<leg>` tags that name each leg of the race. Each `<leg>` tag will contain both a starting port and an ending port in the text.
- The `<teams>` tag contains a number of `<team>` tags with details of each team. Each team has data structured with internal tags:
  - The `<name>` tag contains the team name.
  - The `<position>` tag contains a number of `<leg>` tags with the finish position for the given leg. Each leg is numbered, and the numbering matches the leg definitions in the `<legs>` tag.

In XML notation, the application data shows up in two kinds of places. The first is between the start and the end tags – for example, `<name>Abu Dhabi Ocean Racing</name>`, has text, "Abu Dhabi Ocean Racing", as well as `<name>` and `</name>` tags.

Also, data will also show up as an attribute of a tag; for example, in `<leg n="1">`. The tag is `<leg>`, with an attribute, `n`, with a value of "1". A tag can have an indefinite number of attributes.

The `<leg>` tags point out an interesting problem with XML. These tags include the leg number given as an attribute, while the position for the leg is given as the text inside the tag. There's no real pattern or preference to where useful data is located. Ideally, it's always between tags, but that's not generally true.

XML permits a **mixed content model**. This reflects the case where XML is mixed in with text and there is text inside and outside XML tags. Here's an example of mixed content:

```
<p>  
This has <strong>mixed</strong> content.  
</p>
```

The content of the `<p>` tag is a mixture of text and a tag. The data we're working with in this recipe does not rely on this kind of mixed content model, meaning all the data is within a single tag or an attribute of a tag. The whitespace between tags can be ignored.

## How to do it...

We'll define a function to convert the XML document to a dictionary with leg descriptions and team results:

1. We'll need the `xml.etree` module to parse the XML text. We'll also need a `Path` object to refer to the file. We've assigned a shorter name of `XML` to the `ElementTree` class:

```
import xml.etree.ElementTree as XML  
from pathlib import Path  
from typing import cast
```

The `cast()` function is needed to force tools like **mypy** to treat the result as if it were a given type. This lets us ignore the possibility of `None` results.

2. Define a function to read the XML document from a given `Path` instance:

```
def race_summary(source_path: Path) -> None:
```

3. Create a Python ElementTree object by parsing the XML text. It's often easiest to use `source_path.read_text()` to read the file named by path. We provided this string to the `XML.fromstring()` method for parsing. For very large files, an incremental parser is sometimes more helpful. Here's the version for smaller files:

```
source_text = source_path.read_text(encoding='UTF-8')
document = XML.fromstring(source_text)
```

4. Display the data. The XML element objects has two useful methods for navigating the XML structure, the `find()` and `findall()` methods, to locate the first instance of a tag and locate all instances of a tag, respectively. Using these, we can create a dictionary with two keys, "teams" and "legs":

```
legs = cast(XML.Element, document.find('legs'))
teams = cast(XML.Element, document.find('teams'))
for leg in legs.findall('leg'):
    print(cast(str, leg.text).strip())
    n = leg.attrib['n']
    for team in teams.findall('team'):
        position_leg = cast(XML.Element,
            team.find(f"position/leg[@n='{n}']"))
        name = cast(XML.Element, team.find('name'))
        print(
            cast(str, name.text).strip(),
            cast(str, position_leg.text).strip()
        )
```

Within the `<legs>` tag, there are a number of individual `<leg>` tags. Each of those tags has the following structure:

```
<leg n="1">ALICANTE - CAPE TOWN</leg>
```

The Python expression `leg.attrib['n']` extracts the value of the attribute named `n` from the given element. The expression `leg.text.strip()` will find all the text within the `<leg>` tag, stripped of extra whitespace.

The `find()` and `findall()` methods of an element use XPath notation to locate tags.

We'll examine the features in detail in the *There's more...* section of this recipe.

It's important to note that the results of the `find()` function have a type hint of `XML.Element | None`. We have two choices for handling the possibility of a `None` result:

- Use an `if` statement to handle the cases where the result is `None`.
- Use `cast(XML.Element, tag.find(...))` to claim that the result is never going to be `None`. If the tag is missing, the exception raised will help diagnose the mismatch between the source document and processing expectations by our consumer application.

For each leg of the race, we need to print the finish positions, which are contained within the `<teams>` tag. Within this tag, we need to find the proper `<leg>` tag with the finish position for this team on the given leg. For this, we use a complex XPath search, `f"position/leg[@n='{n}']"`, to locate a specific instance of the `<position>` tag based on the presence of a `<leg>` tag with a specific attribute value. The value of `n` is the leg number. For the ninth leg, `n=9`, the `f`-string will be `"position/leg[@n='9']"`. This will locate the `<position>` tag containing a `<leg>` tag that has an attribute `n` equal to 9.

Because XML supports a mixed content model, all the `\n`, `\t`, and space characters in the content are perfectly preserved by the parsing operation. We rarely want any of this whitespace, and it makes sense to use the `strip()` method to remove any extraneous characters before and after the meaningful content.

## How it works...

The XML parser modules transform XML documents into a fairly complex tree structure based on a standardized **Document Object Model (DOM)**. In the case of the `xml.etree` module, the document will be built from `Element` objects, which generally represent tags and text.

XML can also include processing instructions and comments. We'll ignore them and focus on the document structure and content here.

Each `Element` instance has the text of the tag, the text within the tag, attributes that are

part of the tag, and a tail. The tag is the name inside <tag>. The attributes are the fields that follow the tag name, for example, the <leg n="1"> tag has a tag name of leg and an attribute named n. Values are always strings in XML; any conversion to a different data type is the responsibility of the application using the data.

The text is contained between the start and end of a tag. Therefore, a tag such as <name>Team SCA</name> has "Team SCA" for the value of the text attribute of the Element that represents the <name> tag.

Note that a tag also has a tail attribute. Consider this sequence of two tags:

```
<name>Team SCA</name>
<position>...</position>
```

There's a \n whitespace character after the closing </name> tag and before the opening of the <position> tag. This extra text is collected into the tail attribute of the <name> tag. These tail values can be important when working with a mixed content model. The tail values are generally whitespace when working in an element content model.

## There's more...

Because we can't trivially translate an XML document into a Python dictionary, we need a handy way to search through the document's content. The ElementTree class provides a search technique that's a partial implementation of the **XML Path Language (XPath)** for specifying a location in an XML document. The XPath notation gives us considerable flexibility.

The XPath queries are used with the find() and findall() methods. Here's how we can find all of the team names:

```
>>> for tag in document.findall('teams/team/name'):
...     print(tag.text.strip())
Abu Dhabi Ocean Racing
Team Brunel
Dongfeng Race Team
```

```
MAPFRE
Team Alvimedica
Team SCA
Team Vestas Wind
```

The XPath query looks for the top-level `<teams>` tag. Within that tag, we want `<team>` tags. Within those tags, we want the `<name>` tags. This will search for all the instances of this nested tag structure.

## See also

- There are a number of security issues related to XML documents. See the OWASP XML Security Cheat Sheet for more information.
- The `lxml` library extends the core features of the element tree library, offering additional capabilities.
- The *Reading HTML documents* recipe, later in this chapter, shows how we prepared this data from an HTML source.

## Reading HTML documents

A great deal of content on the web is presented using HTML. A browser renders the data very nicely. We can write applications to extract content from HTML pages.

Parsing HTML involves two complications:

- Ancient HTML dialects that are distinct from modern XML
- Browsers that tolerate HTML that's incorrect and create a proper display

The first complication is the history of HTML and XML. Modern HTML is a specific document type of XML. Historically, HTML started with its own unique document type definitions, based on the older SGML. These original SGML/HTML concepts were revised and extended to create a new language, XML. During the transition from legacy HTML to XML-based HTML, web servers provided content using a variety of transitional document

type definitions. Most modern web servers use a `<DOCTYPE html>` preamble to state that the document is properly structured XML syntax, using the HTML document model. Some web servers will use other DOCTYPE references in the preamble and provide HTML that's not proper XML.

A further complication to parsing HTML is the design of browsers. Browsers are obligated to render a web page in spite of poorly structured or even outright invalid HTML. The design objective is to provide something to the user that reflects the content – not display an error message stating the content is invalid.



HTML pages may be filled with problems and still display a good-looking page in a browser.

We can use the standard library `html.parser` module, but it's not as helpful as we'd like. The **Beautiful Soup** package provides more helpful ways to parse HTML pages into useful data structures. This is available from the **Python Package Index (PyPI)**. See <https://pypi.python.org/pypi/beautifulsoup4>.

This must be downloaded and installed with the following terminal command:

```
(cookbook3) % python -m pip install beautifulsoup4
```

## Getting ready

We've gathered some historical sailboat racing results in `Volvo Ocean Race.html`. This file contains information on teams, legs, and the order in which the various teams finished each leg. It's been scraped from the Volvo Ocean Race website, and it looks wonderful when opened in a browser. For more information on this data, see the *Reading JSON and YAML documents* recipe in this chapter.

While Python's standard library has the `urllib` package to acquire documents, it's common to use the **Requests** package to read web pages.

Generally, an HTML page has the following overall structure:

```
<html>
<head>...</head>
<body>...</body>
</html>
```

Within the <head> tag, there will be metadata, links to JavaScript libraries, and links to **Cascading Style Sheet (CSS)** documents. The content is in the <body> tag.

In this case, the race results are in an HTML <table> tag inside the <body> tag. The table has the following structure:

```
<table>
  <thead>
    ...
  </thead>
  <tbody>
    ...
  </tbody>
</table>
```

The <thead> tag defines the column titles for a table. There's a single row tag, <tr>, with table heading tags, <th>, that include the column titles. For the example data, each of the <th> tags look like this:

```
<th tooltipster data-title="<strong>ALICANTE - CAPE TOWN</strong>"
data-theme="tooltipster-shadow" data-htmlcontent="true" data-position="top">
LEG 1</th>
```

The essential display is an identifier for each leg of the race; LEG 1, in this example. This is the text content of the <th> tag. There's also an attribute value, data-title, that's used by a JavaScript function. This attribute value has the name of the leg, and it is displayed when the cursor hovers over a column heading.

The <tbody> tag includes rows with the results for each team and race. Each <tr> table



row tag contains `<td>` table data tags with the name of a team and its results. Here's a typical `<tr>` row from the HTML:

```
<tr class="ranking-item">
  <td class="ranking-position">3</td>
  <td class="ranking-avatar"></td>
  <td class="ranking-team"> Dongfeng Race Team</td>
  <td class="ranking-number">2</td>
  <td class="ranking-number">2</td>
  <td class="ranking-number">1</td>
  <td class="ranking-number">3</td>

  <td class="ranking-number" tooltipster
  data-title="<center><strong>RETIRED</strong><br> Click for more
  info</center>" data-theme="tooltipster-3"
  data-position="bottom" data-truecontent="true">
  <a href="/en/news/8674_Dongfeng-Race-Team-breaks-mast-crew-safe.html"
  target="_blank">8</a>
  <div class="status-dot dot-3"></div></td>
  ... more columns ...
</tr>
```

The `<tr>` tag has a `class` attribute that defines the CSS style for this row. This `class` attribute can help our data-gathering application locate the relevant content.

The `<td>` tags also have `class` attributes. For this well-designed data, the class clarifies what the content of the `<td>` cell is. Not all CSS class names are as well defined as these.

One of the cells – with the `tooltipster` attribute – has no text content. Instead, this cell has an `<a>` tag and an empty `<div>` tag. That cell also contains several attributes, including `data-title`, among others. These attributes are used by a JavaScript function to display additional information in the cell.

Another complexity here is that the `data-title` attribute contains text that's actually HTML content. Parsing this bit of text will require creating a separate BeautifulSoup parser.

## How to do it...

We'll define a function to convert the HTML `<table>` to a dictionary with leg descriptions and team results:

1. Import the `BeautifulSoup` class from the `bs4` module to parse the text. We'll also need a `Path` object to refer to the file:

```
from bs4 import BeautifulSoup
from pathlib import Path
from typing import Any
```

2. Define a function to read the HTML document from a given `Path` instance:

```
def race_extract(source_path: Path) -> dict[str, Any]:
```

3. Create the soup structure from the HTML content. We'll assign it to a variable, `soup`. As an alternative, we could also read the content using the `Path.read_text()` method:

```
with source_path.open(encoding="utf8") as source_file:
    soup = BeautifulSoup(source_file, "html.parser")
```

4. From the `soup` object, we need to navigate to the first `<table>` tag. Within that, we need to find the first `<thead>` and `<tr>` tags. Navigating to the first instance of a tag is done by using the tag name as an attribute:

```
thead_row = soup.table.thead.tr # type: ignore [union-attr]
```

A special comment is used to silence a **mypy** warning. The `# type: ignore [union-attr]` is needed because each tag property has a type hint of `Tag | None`. For some applications, additional `if` statements can be used to confirm the expected combinations of tags that are present.

5. We must accumulate heading data from each `<th>` cell within the row:

```

legs: list[tuple[str, str | None]] = []
for tag in thead_row.find_all("th"): # type: ignore [union-attr]
    leg_description = (
        tag.string, tag.attrs.get("data-title")
    )
    legs.append(leg_description)

```

6. To find the table's content, we navigate down into the <table> and <tbody> tags:

```
tbody = soup.table.tbody # type: ignore [union-attr]
```

7. We need to visit all of the <tr> tags. Within each row, we want to convert the content of all <td> tags into team names and a collection of team positions, depending on the attributes of the td tag:

```

teams: list[dict[str, Any]] = []
for row in tbody.find_all("tr"): # type: ignore [union-attr]
    team: dict[str, Any] = {
        "name": None,
        "position": []}
    for col in row.find_all("td"):
        if "ranking-team" in col.attrs.get("class"):
            team["name"] = col.string
        elif (
            "ranking-number" in col.attrs.get("class")
        ):
            team["position"].append(col.string)
        elif "data-title" in col.attrs:
            # Complicated explanation with nested HTML
            # print(col.attrs, col.string)
        pass
    teams.append(team)

```

8. Once the legs and teams have been extracted, we can create a useful dictionary that will contain the two collections:

```

document = {
    "legs": legs,
    "teams": teams,
}

```

```

}
return document

```

We've created a list of legs showing the order and names for each leg, and we parsed the body of the table to create a dict-of-list structure with each leg's results for a given team. The resulting object looks like this:

```

>>> source_path = Path("data") / "Volvo Ocean Race.html"
>>> race_extract(source_path)
{'legs': [(None, None),
          ('LEG 1', '<strong>ALICANTE - CAPE TOWN'),
          ('LEG 2', '<strong>CAPE TOWN - ABU DHABI</strong>'),
          ('LEG 3', '<strong>ABU DHABI - SANYA</strong>'),
          ('LEG 4', '<strong>SANYA - AUCKLAND</strong>'),
          ('LEG 5', '<strong>AUCKLAND - ITAJAÍ</strong>'),
          ('LEG 6', '<strong>ITAJAÍ - NEWPORT</strong>'),
          ('LEG 7', '<strong>NEWPORT - LISBON</strong>'),
          ('LEG 8', '<strong>LISBON - LORIENT</strong>'),
          ('LEG 9', '<strong>LORIENT - GOTHENBURG</strong>'),
          ('TOTAL', None)],
 'teams': [
   {'name': 'Abu Dhabi Ocean Racing',
    'position': ['1', '3',
                '2', '2',
                '1', '2',

```

...

```

{'name': 'Team Vestas Wind',
 'position': ['4',
              None,
              None,
              None,
              None,
              None,
              None,
              '2',
              '6',
              '60']}]}}

```

Within the body of the table, many cells have None for the final race position and a complex value in data-title attribute for the specific `<TD>` tag. Parsing the HTML embedded in this text follows the pattern shown in the recipe, using another `BeautifulSoup` instance.

## How it works...

The `BeautifulSoup` class transforms HTML documents into fairly complex objects based on a **Document Object Model (DOM)**. The resulting structure will be built from instances of the `Tag`, `NavigableString`, and `Comment` classes.

Each `Tag` object has a name, string, and attributes. The name is the word inside `<` and `>` characters. The attributes are the fields that follow the tag name. For example, `<td class="ranking-number">1</td>` has a tag name of `td` and an attribute named `class`. Values are often strings, but in a few cases, the value can be a list of strings. The string attribute of the `Tag` object is the content enclosed by the tag; in this case, it's a very short string, `1`.

HTML is a mixed content model. When looking at the children of a given tag, there will be a sequence of child `Tags` and child `NavigableText` objects freely intermixed.

The `BeautifulSoup` parser class depends on a lower-level library to do some of the parsing work. It's easiest to use the built-in `html.parser` module for this. The alternatives offer some advantages, like better performance or better handling of damaged HTML.

## There's more...

The `Tag` objects of Beautiful Soup represent the hierarchy of the document's structure. There are several kinds of navigation among tags. In this recipe, we relied on the way `soup.html` is the same as `soup.find("html")`. We can also search by attribute values, including `class` and `id`. These often provide semantic information about the content.

In some cases, a document will have a well-designed organization, and a search by the `id` attribute or `class` attribute will find the relevant data. Here's a typical search for a given structure using the HTML `class` attribute:

```
>>> ranking_table = soup.find('table', class_="ranking-list")
```

Note that we have to use `class_` in our Python query to search for the attribute named `class`. The token `class` is a reserved word in Python and cannot be used as a parameter name. Given the overall document, we're searching for any `<table class="ranking-list">` tag. This will find the first such table in a web page. Since we know there will only be one of these, this attribute-based search helps distinguish between what we are trying to find and any other tabular data on a web page.

## See also

- The Requests package can greatly simplify the code required to interact with complex websites.
- See the <https://www.robotstxt.org> website for information on the `robots.txt` file and the RFC 9309 Robots Exclusion Protocol.
- The *Reading JSON and YAML documents* and *Reading XML documents* recipes, shown earlier in this chapter, both use similar data. The example data was created for them by scraping the original HTML page using these techniques.

## Join our community Discord space

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

<https://packt.link/dHrHU>





# 12

## Graphics and Visualization with Jupyter Lab

A great many problems are simplified through visualization of the data. The human eye is particularly suited to identifying relationships and trends. Given a display of a potential relationship (or trend), it makes sense to turn to more formal statistical methods to quantify the relationship.

Python offers a number of graphical tools. For data analytics purposes, one of the most popular is **matplotlib**. This package offers numerous graphic capabilities. It integrates well with **Jupyter Lab**, providing us an interactive environment to visualize and analyze data.

It's possible to do a great deal of Python development in Jupyter Lab. While wonderful, this is not a perfect **Integrated Development Environment (IDE)**. The one minor drawback is the interactive notebook relies on global variables, something that isn't ideal for writing modules or applications. The use of global variables can lead to confusion when transforming a notebook into a module intended for reuse.



In addition to running Python code and displaying graphics, a Jupyter Lab notebook can also render cells in Markdown notation. This lets us write very good-looking documentation around the graphical analysis of data. This includes the ability to render mathematical formulae properly. It lets us include a cell with  $e^{\pi i} + 1 = 0$  kinds of mathematics close to the relevant code.

The two packages we'll use aren't part of the standard Python distribution. We'll need to install them. In some cases, using a tool like **Conda** can help to install these large and complex packages. In many cases, however, a PIP install will work when our computer is one of the widely supported varieties.

```
python -m pip install matplotlib jupyterlab
```

Using the `python -m pip` command ensures that we will use the `pip` command that is compatible with the currently active virtual environment.

This will conclude with a line something like the following:

```
Installing collected packages: pyparsing, pillow, numpy, kiwisolver, fonttools,  
    cyclcr, contourpy, matplotlib
```

This line will be followed by a list of new packages added to the current virtual environment.

Note that Jupyter Lab will make use of the IPython implementation of Python. This implementation includes some additional features that are very handy for managing the complex client-server connection between a browser and a Jupyter Lab server.

The most visible difference from the standard Python implementation is a distinct prompt. IPython uses `In [n]:` as a prompt. The number,  $n$ , increases during the session. It's possible to recall specific previous commands and outputs using the numbering of the prompts.

In this chapter, we'll look at the following recipes:

- *Starting a Notebook and creating cells with Python code*

- *Ingesting data into a notebook*
- *Using pyplot to create a scatter plot*
- *Using axes directly to create a scatter plot*
- *Adding details to markdown cells*
- *Including Unit Test Cases in a Notebook*

## Starting a Notebook and creating cells with Python code

We'll use a terminal window to enter a command to start the lab server. The `jupyter lab` command will do two things:

- It will start the backend, number-crunching server component of Jupyter Lab.
- It will also try to launch a browser window that's connected to that server component.

The rest of our interaction with the various notebooks in the lab will be through the browser.

In the rare case that a browser isn't launched, the log will provide links that can be used in your browser of choice. The IPython runtime will make use of the various installed packages.

For this first recipe, we'll focus on the administrative aspects of starting and stopping the Jupyter Lab server, and creating notebooks.

### Getting ready

We're going to start a Jupyter Lab session and create a notebook to make sure that the environment works and all of the needed components are installed.

The use of a terminal window to start Jupyter Lab is sometimes confusing. Many programmers are used to working inside an IDE to create and test code. Starting Jupyter Lab is

generally done from a terminal window, not a Python editor or interactive Python REPL session.

## How to do it...

1. Open a terminal window. Change to a working directory that has access to data and a folder for notebooks. Be sure the proper virtual environment is active. The Jupyter Lab server is limited to working in the directory in which it was started. Enter the following command to start Jupyter Lab:

```
(coobook3) % python -m jupyter lab
```

This will output a log of actions taken by the server. In the block of lines will be URLs to connect to the server securely. Generally, a browser window will also open.

2. In the browser window, the Jupyter Lab window will show the launcher tab. It looks like the page shown in *Figure 12.1*.

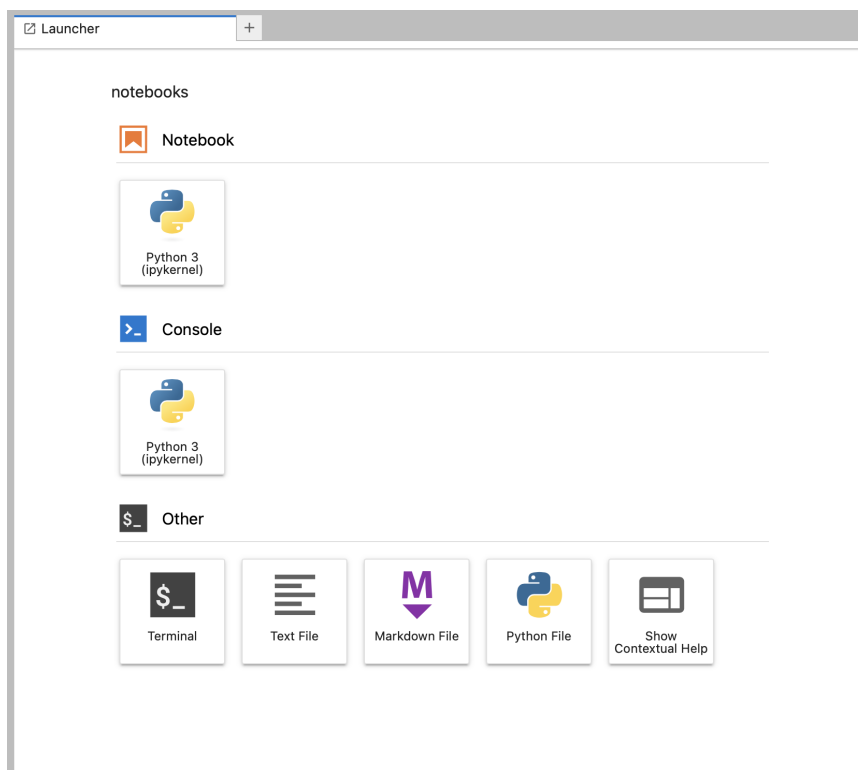


Figure 12.1: Jupyter Lab launcher

3. Click the Python 3 (ipykernel) icon in the Notebook section of the launcher. This will open a Jupyter Notebook named `Untitled.ipynb`.

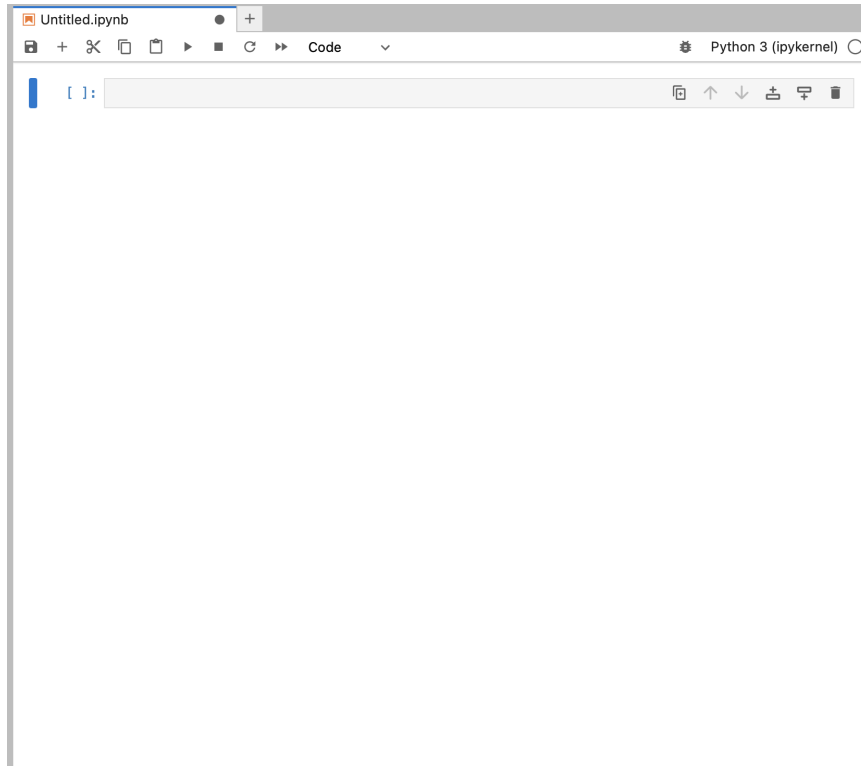


Figure 12.2: Jupyter notebook ready for work

The top of this tab has a notebook-level menu bar with a series of icons to save the notebook, add a cell, cut a cell, copy a cell, and paste a cell, among other things.

The ► icon on the notebook’s menu bar will execute the cell’s Python or format the cell’s Markup content. This can also often be done with the *Shift+Enter* keyboard combination.

The remaining icons will stop the running kernel and restart it. The ►► icon will restart the notebook, running all the cells. A drop-down menu lets you choose between the **code**, **markdown**, and **raw** cells. We’ll spend most of our time creating code and markdown cells.

The initial content has a label, [ ]:, and a text box into which we can enter code. As

we add more commands, this area will be filled with cells of code and their output.

4. To be sure we have all the required packages, enter the following code into the first cell:

```
from matplotlib import pyplot as plt
from pydantic import BaseModel
```

Use the *Shift+Enter* keyboard combination or click the ► icon on the notebook's menu bar to execute the code in this cell. If it works, that confirms everything we need is installed in the current virtual environment.

## How it works...

One of the many features of Jupyter Lab is creating and editing notebooks. We can enter code and execute code directly, saving the results for others to review. This permits a great deal of flexibility in how to acquire, analyze, and share data.

Because we can also open a Python console and edit Python modules, we can do a tremendous amount of development work using Jupyter Lab as an IDE. It's possible to export a notebook as a script file. This permits a transition from a sequence of cells representing a number of good ideas into a module or application.

When confronted with a new problem or new data, using a notebook as a way to record experiments is often encouraged. Cells that show failures reflect lessons learned and are worth preserving. Cells that show success, of course, serve to guide colleagues through the learning process.

The Jupyter Lab environment is designed to be used in a variety of container configurations.

Two common container architectures are:

- A large analytical host separate from an analyst's laptop.
- An individual laptop acting as both the host for the Jupyter Lab server process and **also** the host for the browser session. This is the environment created for this recipe.

The idea is to be able to scale up and process a very large dataset on a very expensive,

very large host. This number-crunching host runs the Jupyter Lab and the kernel for our notebook. Our laptop only runs a browser and a terminal window.

## There's more...

The working notebook needs to be saved or the current state will be lost. Save early and save often to make sure no precious results are lost. There are two ways to stop the Jupyter Lab server when we're finished using it.

- From the Jupyter Lab browser window.
- From the terminal window.

To stop processing from the browser window, use the **File** menu. At the bottom of this menu is the **Shut Down** menu item. This stops the server and disconnects the browser session.

To stop processing from the terminal window, use *Control+C* ( $\hat{C}$ ) twice to stop the processing. Entering  $\hat{C}$  once will get a `Shutdown this Jupyter server (y/[n])?` prompt. A second  $\hat{C}$  (or a `y` answer) is required to stop the process.

## See also

- In the *Ingesting data into a notebook* section, we'll move beyond the basics and load a notebook with data.
- See *Learning Jupyter* for an in-depth book on Jupyter Lab.

## Ingesting data into a notebook

As a sample data analytics problem, we'll look at a data collection containing four closely related series of samples. The file is named `anscombe.json`. Each series of data is a sequence of  $(x, y)$  data pairs and a name for the series, represented as a Python dictionary. The `series` key has the name of the series. The `data` key is the list of data pairs. The four series are sometimes called *Anscombe's Quartet*.

We'll create a notebook to ingest the data. To begin the work, this initial recipe will focus

on ordinary Python expressions to confirm the data was loaded properly. In later recipes, we'll use visualization and statistical methods to see if there are correlations between the two variables.

## Getting ready

There are a few preliminary steps:

1. Make sure the Jupyter Lab server is running. If it isn't, see the *Starting a Notebook and creating cells with Python code* recipe to start the server and open a notebook.
2. Locate the browser window for this server. When starting a server, a browser window is often displayed that connects to the server.
3. Start a new notebook for this recipe.

If we close the browser application, or if Python can't launch our preferred browser, then the Jupyter Lab server will run, but there's no **obvious** browser window to connect to the server. We can locate the server by making an inquiry, using this jupyter command:

```
(cookbook3) % python -m jupyter server list
```

This will identify all running servers. It will provide a URL to connect to the server. The output might look something like the following:

```
(cookbook3) % jupyter server list
Currently running servers:
http://localhost:8888/?token=85e8ad8455cd154bd3253ba0339c783ea60c56f836f7b81c :: /Users/
slott/Documents/Writing/Python/Python Cookbook 3e
```

The actual URL shown in the output on your computer can be used to connect to the server.

## How to do it...

1. Since we're going to read a file in the JSON format, the first cell can be a code cell with the import statements required.



```
import json
from pathlib import Path
```

Using the *Enter* (or *Return*) key in the cell adds lines of code.

Using *Shift+Enter* (or *Shift+Return*) will execute the cell's code and open a new cell for more code. This also happens when we click the ► icon on the menu bar to execute the code, followed by the + icon on the menu bar to add a new, empty cell.

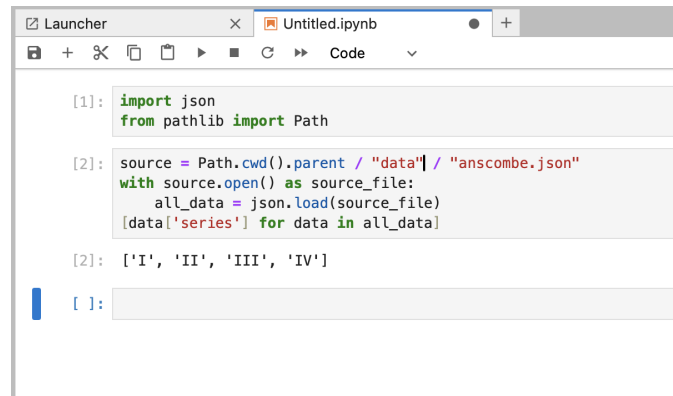
After a cell is executed, the number is filled in for the cell label; the first cell shows [1]: when the cell has been executed.

2. It makes sense to put the whole “read and extract data” process into a single cell:

```
source_path = Path.cwd().parent.parent / "data" / "anscombe.json"
with source_path.open() as source_file:
    all_data = json.load(source_file)
[data['series'] for data in all_data]
```

The path from the current working directory to data assumes the notebook is in a `src/ch12` folder, and the `src` folder is a peer of the `data` folder. If your project isn't structured like this, then the computation of `source_path` will need to change.

When we execute this cell, we'll see the names of the four data series in this collection. They look like this:



```

[1]: import json
    from pathlib import Path

[2]: source = Path.cwd().parent / "data" / "anscombe.json"
    with source.open() as source_file:
        all_data = json.load(source_file)
        [data['series'] for data in all_data]

[2]: ['I', 'II', 'III', 'IV']

[ ]:

```

Figure 12.3: Jupyter notebook with two cells of code

3. We can look at the  $(x, y)$  pairs in this source. A cell with an expression is sufficient to show the value of the expression:

```
all_data[0]['data']
```

4. We can define a class to hold these data pairs. Because the data is JSON-formatted, and because the pydantic package offers really good JSON parsing, we can consider extending the BaseModel class.

This requires rewriting cell 1 to extend the sequence of imports to include:

```

import json
from pathlib import Path
from pydantic import BaseModel

```

We can click the ►► icon to restart the notebook, running all the cells. This is essential when we go back toward the top and make a change.

At the top of the page, over the tabs for the launcher at the notebooks, there's a higher-level menu bar with items like **File**, **Edit**, **View**, **Run**, **Kernel**, **Tabs**, **Settings**, and **Help**. The **Kernel** menu has a **Restart Kernel and Run All Cells...** item that has the same functionality as the ►► icon in the notebook menu bar.

5. Define a class for an X-Y pair. Then, define a class for the series of individual pairs:

```
class Pair(BaseModel):
    x: float
    y: float

class Series(BaseModel):
    series: str
    data: list[Pair]
```

6. Then, we can populate the class instances from the `all_data` object:

```
clean_data = [Series.model_validate(d) for d in all_data]
```

7. Use an expression like `clean_data[0]` to see a specific series.
8. We have an awkward problem where each series has a position in the `clean_data` sequence, and a name. Using a mapping is better than using a sequence:

```
quartet = {s.series: s for s in clean_data}
```

9. Use an expression like `quartet['I']` to see a specific series.

## How it works...

The Python in each cell is not dramatically different from the code used in the *Reading JSON and YAML documents* recipe in *Chapter 11*. We've used Jupyter Lab to start an IPython kernel that evaluates the code in each cell.

When a cell's code ends with an expression, the Jupyter Notebook will display any non-None output. This is similar to the command-line REPL. A `print()` function isn't needed to display the results of the final expression in a cell.

The Jupyter notebook interface is a distinct way to access Python and provides a richly interactive environment. Underneath the clever editing and display features, the language and libraries are still Python.

## There's more...

After reviewing the notebook, it's clear we can optimize some of the processing. There's no real benefit in creating the `all_data` and `clean_data` objects. The real goal is to work with the `quartet` object.

We can use the following cell to parse and load the series:

```
source_path = Path.cwd().parent.parent / "data" / "anscombe.json"
with source_path.open() as source_file:
    json_document = json.load(source_file)
    source_data = (Series.model_validate(s) for s in json_document)
    quartet = {s.series: s for s in source_data}
```

It helps to insert a new cell after the Pydantic class definition with this code. We can then execute the code. A cell with an expression like `quartet['IV']` can be used to confirm that the data was loaded.

A more complete check would be the following cell:

```
for name in quartet:
    print(f"{name:3s} {len(quartet[name].data):d}")
```

This shows each series name and the number of points in the data for the series.

Once this works, we can remove prior cells, add Markdown, and rerun the notebook to make sure it works properly and neatly displays the data to be analyzed.

## See also

- See the *Reading JSON and YAML documents* recipe in *Chapter 11* for more on JSON-format files.
- The *Implementing more strict type checks with Pydantic* recipe in *Chapter 10* covers some features of the **Pydantic** package.

## Using pyplot to create a scatter plot

The **matplotlib** project has an immense variety of graph and plot types that it can produce. It is extremely sophisticated, which makes it challenging to use for some kinds of analysis.

A particularly useful subset of features is collected in a sub-package called `pyplot`. This group of features reflects some common assumptions and optimizations that work out very nicely when working in Jupyter Lab. In other contexts, these assumptions are often limiting.

To make things easier, the `pyplot` package will automatically manage the figure being created. It will track any sub-plots that fill in the picture. It will track the various axes and artists that comprise those subplots.

For more information, see *Parts of a Figure* in the **matplotlib** tutorial. This diagram identifies the various elements of a figure and what parts of **matplotlib** are used to create or control those elements.

In the *Ingesting data into a notebook* recipe, we looked at a collection of data that had four data series. The file that contains this data section is named `anscombe.json`. Each series of data is a sequence of  $(x, y)$  data pairs, and each series has a name.

The general approach will be to define some useful classes that provide helpful definitions of the series and the samples within a series. Given those definitions, we can read the `anscombe.json` file to acquire the data. Once it is loaded, we can then create a figure that shows the data pairs.

## Getting ready

There are a few preliminary steps:

1. Start the Jupyter Lab server and locate the browser window for it. If it isn't running, see the *Starting a Notebook and creating cells with Python code* recipe to start a server. See *Ingesting data into a notebook* for advice on locating a server that runs in the background.

2. Start a new notebook for this recipe.

## How to do it...

1. Start the notebook with a cell that describes what the notebook will have in it. This should be a Markdown cell to record some notes:

```
# Anscombe's Quartet

The raw data has four series. The correlation coefficients are high.
Visualization shows that a simple linear regression model is
misleading.

## Raw Data for the Series
```

2. Create the imports required to use **Pydantic** to load JSON-formatted data. This will be a code cell:

```
import json
from pathlib import Path
from pydantic import BaseModel
```

3. Define two classes that define each series of data. One class has the individual  $(x, y)$  pair. The other is the series of pairs, along with the series name:

```
class Pair(BaseModel):
    x: float
    y: float

class Series(BaseModel):
    series: str
    data: list[Pair]
```

4. Write a cell with the code required to read the data, and create a global variable with the cleaned data:

```
source = Path.cwd().parent.parent / "data" / "anscombe.json"
with source.open() as source_file:
    json_document = json.load(source_file)
```

```
source_data = (Series.model_validate(s) for s in json_document)
quartet = {s.series: s for s in source_data}
```

The value of `clean_data` contains a list of four individual `Series` objects. An expression like `quartet['I']` will reveal one of the series.

5. Add a Markdown cell showing what the next part of this notebook will contain:

```
## Visualization of each series
```

6. Write the needed import for the `pyplot` package. This is often renamed `plt` to simplify the code that is written in the Jupyter notebook cells:

```
from matplotlib import pyplot as plt
```

7. For an individual series, we'll need to extract two parallel sequences of numbers. We'll use list comprehensions on `clean_data[0].data` to extract the `x` values for one sequence and the `y` values for a second, parallel sequence:

```
x = [p.x for p in quartet['I'].data]
y = [p.y for p in quartet['I'].data]
```

8. The `scatter()` function creates the essential scatter plot. We provide two parallel sequences: one has the `x` values and one has the `y` values. The `title()` function will place a label above the plot. We've constructed a string from the series name. While not always necessary, a `plt.show()` is sometimes needed to display the resulting plot:

```
plt.scatter(x, y)
plt.title(f"Series {quartet['I'].series}")
plt.show()
```

The resulting notebook will include cells like this:

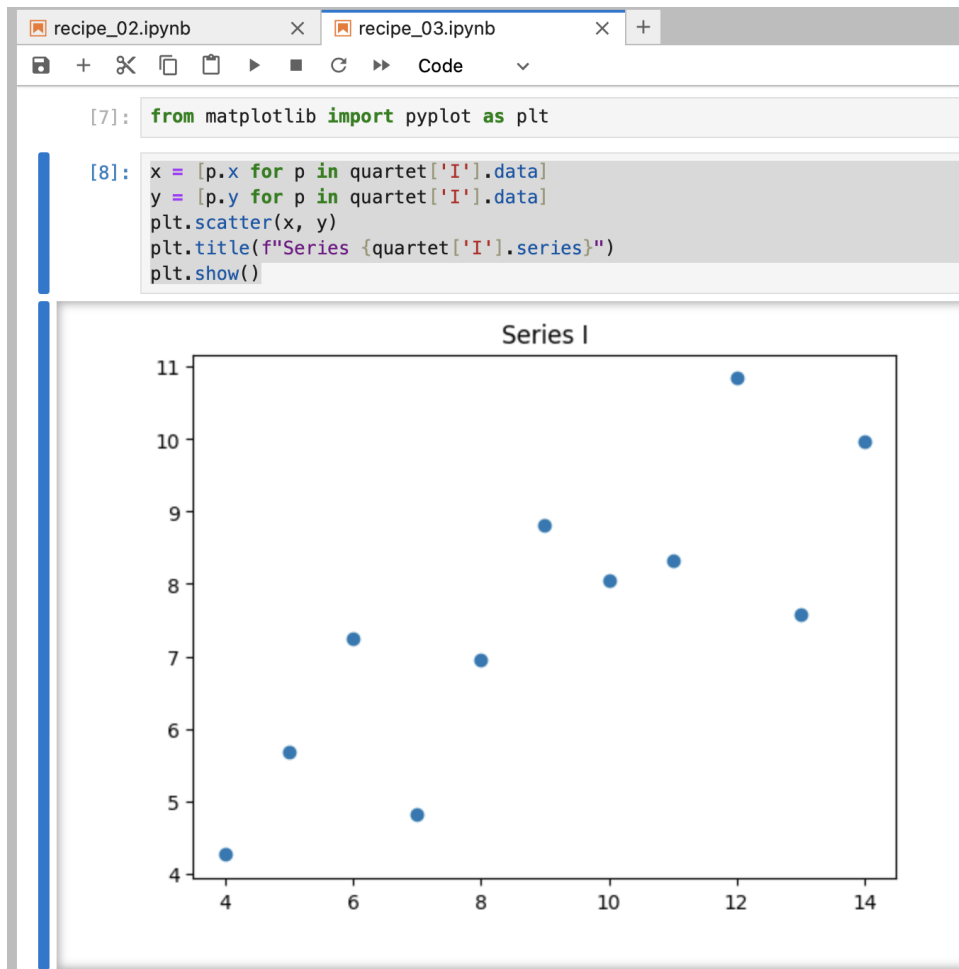


Figure 12.4: Jupyter notebook with a figure for Series I

## How it works...

The underlying matplotlib package has a tall stack of components to support graphics and visualizations of data. One foundation is the idea of a *backend* component to integrate with the wide variety of contexts, frameworks, and platforms in which Python is used. This also includes interactive environments like Jupyter Lab. It also includes static, non-interactive



backends that can produce a variety of image file formats.

Another foundational element of `matplotlib` is the API that lets us create a `Figure` object populated with `Artist` objects. The collection of `Artist` objects will draw the titles, axes, data points, and lines we expect to see in the figure. The `Artist` objects can be interactive, permitting them to refresh a figure when the data changes, or when the display shape changes.

In this kind of analysis notebook, we're often more interested in a static figure that's drawn once. Our objective is to save the notebook for other people to view and understand the data. The ultimate goal is to communicate relationships or trends in the data.

The **pyplot** package contains a number of simplifications for the overall `matplotlib` API. These simplifications save us from some of the tedious details of keeping track of the various axes instances that are used to create a plot displayed in a figure.

## There's more...

We'll often want to see several closely related plots as part of a single figure. In this case, where we have four series of data in a single file, it seems particularly helpful to put all four plots together.

This is done using `plt.figure()` to create an overall figure. Within this figure, each `plt.subplot()` function can create a distinct subplot. The layout of the figure is provided as part of each subplot request as three numbers: the number of plots arranged vertically within the figure, the number plots horizontally in the figure, and this particular plot's location within that layout.

We might use `2, 2, n` to state that the figure has a  $2 \times 2$  arrangement, and this specific subplot has position  $n$ . The positions are counted across the figure from top left to bottom right. The position can also span more than one section of the figure so that we have one large plot and a number of smaller plots.

To make it easier to extract the `x` and `y` attributes of each plot, we'll modify the definition of the `Series` class. We'll add two properties, `x` and `y`, that will extract all of the series

values. This redefines the `Series` class as follows:

```
class Series(BaseModel):
    series: str
    data: list[Pair]

    @property
    def x(self) -> list[float]:
        return [p.x for p in self.data]

    @property
    def y(self) -> list[float]:
        return [p.y for p in self.data]
```

Adding these properties permits some slight simplifications in the `plt.scatter()` function. The overall figure can be created by a cell with the following code:

```
plt.figure(layout='tight')
for n, series in enumerate(quartet.values(), start=1):
    title = f"Series {series.series}"
    plt.subplot(2, 2, n)
    plt.scatter(series.x, series.y)
    plt.title(title)
```

After changing the new `Series` class definition, the notebook's cells must be rerun from the beginning. In the **Run** menu, the item to **Restart the kernel and run all cells** will incorporate the revised class and reload the data.

There are numerous options and parameters for the definition of the figure overall, the scatter plot, and the axes for the scatter plot. Additionally, there are numerous alternative kinds of plots available. The Matplotlib Examples Gallery shows dozens of varieties of plots. For example, in a case where we have an `Counter` object, we can create a bar chart using a sequence of  $x$  values and a sequence of height values, using the `bar()` function instead of the `scatter()` function. See the *Creating dictionaries – inserting and updating* recipe in *Chapter 5* for examples of creating a `Counter` object to summarize frequencies in source data.

## See also

- See the *Reading JSON and YAML documents* recipe in *Chapter 11* for more on JSON-format files.
- The *Implementing more strict type checks with Pydantic* recipe in *Chapter 10* covers some features of using the **Pydantic** package.
- The *Ingesting data into a notebook* recipe earlier in this chapter looks at JSON loading in more depth.
- See the Matplotlib Examples Gallery for dozens of varieties of plots.

## Using axes directly to create a scatter plot

Many ordinary graphics visualizations can be done using the functions available directly in the `matplotlib` module. In the previous recipes, we used the `scatter()` function to draw a scatter plot showing the relationship between two variables. Other functions like `bar()`, `pie()`, and `hist()` will create other kinds of plots from our raw data. However, there are times when the readily available functions on the `matplotlib` module aren't completely appropriate, and we'd like to do a few more things in our images.

In this recipe, we'll add a legend box to each of the sub-plots to show the linear regression parameters that fit the scatter-plot data.

In the *Ingesting data into a notebook* recipe, we looked at a collection of data that had four series of data. The file that contained this data is named `anscombe.json`. Each series of data is a dictionary with a sequence of  $(x, y)$  data pairs, and a name for the series.

The general approach will be to define some classes that provide helpful definitions of the series and the samples within a series. Given those definitions, we can read the `anscombe.json` file to acquire the data. Once it is loaded, we can then create a figure that shows the data pairs.

The `statistics` module in Python provides two handy functions, `correlation()` and `regression()`, that help us annotate each plot with some parameters.

## Getting ready

There are a few preliminary steps:

1. Ensure the Jupyter Lab server is running, and locate the browser window for it. The *Starting a Notebook and creating cells with Python code* recipe shows how to start a server. The *Ingesting data into a notebook* has some additional advice on locating a server running in the background.
2. Start a new notebook for this recipe.

## How to do it...

1. Start the notebook with a cell that describes what the notebook will have in it. This should be a cell with Markdown content instead of code content:

```
# Anscombe's Quartet

Visualization with correlation coefficients and linear regression
model.

## Raw Data for the Series
```

2. Create the imports required to use **Pydantic** to load JSON-formatted data. This will be a code cell:

```
import json
from pathlib import Path
import statistics

from pydantic import BaseModel
```

3. Define two classes that define each series of data. One class has the individual  $(x, y)$  pair:

```
class Pair(BaseModel):
    x: float
    y: float
```

The second class is series of pairs, along with the series name. This includes two methods to compute the useful statistical summaries like correlation and regression:

```

        return [p.y for p in self.data]

    @property
    def correlation(self) -> float:
        return statistics.correlation(self.x, self.y)

```

4. Create a cell to ingest the data, creating a dictionary that maps a series name to the related Series instance:

```

source = Path.cwd().parent.parent / "data" / "anscombe.json"
with source.open() as source_file:
    json_document = json.load(source_file)
    source_data = (Series.model_validate(s) for s in json_document)
    quartet = {s.series: s for s in source_data}

```

5. Confirm that the previous cells all work. Create a cell to evaluate an expression like `quartet['I'].correlation`. Rounded, the result will be 0.816. Interestingly, the result is almost the same for all four series.
6. Start a cell with an expression to create a figure using the `figure()` function. Providing a value of 'tight' for the layout produces a good-looking figure. It's essential to assign this to a variable so that the object persists until it can be displayed by the `plt.show()` function:

```
fig = plt.figure(layout='tight')
```

Add a line to the cell to create a collection of axes for displaying four subplots within the overall figure. The `subplot_mosaic()` function provides a great deal of sophisticated layout capabilities. The list-of-lists structure will create a grid that's square. The axes will be assigned to a dictionary, `ax_dict` with four distinct keys. We've chosen the keys to match the series names and positioned them in rows and columns of the resulting figure, using lists of lists:

```
ax_dict = fig.subplot_mosaic(
    [
        ["I", "II"],
        ["III", "IV"],
    ],
)
```

7. Add the scatter plot and the text caption. We can also build a string with the correlation coefficient,  $r$ . This can be placed near the bottom-right corner of the plot using a relative position that is specified as  $(.95, .05)$ ; this is transformed by the `ax.transAxes` transformer into coordinates based on the sizes of axes.

```
for name, ax in ax_dict.items():
    series = quartet[name]
    ax.scatter(series.x, series.y)
    ax.set_title(f"Series {name}")
    eq1 = rf"$r = {series.correlation:.3f}$"
    ax.text(.95, .05, f"{eq1}",
            fontfamily='sans-serif',
            horizontalalignment='right', verticalalignment='bottom',
            transform=ax.transAxes)
plt.show()
```

The use of  $\$$  around the string, `rf"$r = {...}$"`, forces **matplotlib** to apply  $\TeX$  formatting rules to the text, creating a properly formatted mathematical equation.

8. When creating an explicit figure, a final call to the `show()` function is needed to display the image:

```
plt.show()
```

## How it works...

The graphics technology stack in **matplotlib** includes an immense variety of Artist subclasses. Each of these will create some part of the final image. In this case, we've used the `subplot_mosaic()` function to create four subplot objects, each with a set of axes.

We've used the axes object to display the data, specifying that a scatter plot organization should be used. The title for the plot and the text block with the correlation coefficient also draw details in the plot.

At some point, the display can become cluttered with details. A good presentation of data needs to have a message. There are many interesting books and articles written about good (and bad) ways to present data to an audience. Consider *Matplotlib for Python Developers* from Packt Publishing as a way to learn more about data visualization.

## There's more...

The correlation value suggests there's a relationship between the  $x$  and  $y$  variables in the series. We can use linear regression to compute the parameters for a linear model that predicts the  $y$  value when given an  $x$  value.

A `linear_regression()` function is part of the `statistics` module in the standard library. The result of this function is a tuple with slope and intercept values that describe a linear relationship,  $y = mx + b$ , where  $m$  is the slope and  $b$  is the intercept.

We can update the cells of this notebook to add the linear regression computation. There are several changes:

1. Change the `Series` class to add another property that performs the linear regression computation:

```
@property
def regression(self) -> tuple[float, float]:
    return statistics.linear_regression(self.x, self.y)
```

2. Add a cell to confirm that the regression works. The cell can display the expression `quartet['I'].regression`. The result will have a slope of almost 0.5 and an intercept of almost 3.0. Interestingly, this is almost identical for all four series.
3. Change the subplot label to include the regression parameters:

```
lr = series.regression
eq1 = rf"$r = {series.correlation:.3f}$"
eq2 = rf"$Y = {lr.slope:.1f} \times X + {lr.intercept:.2f}$"
ax.text(.95, .05, f"{eq1}\n{eq2}",
        fontfamily='sans-serif',
        horizontalalignment='right', verticalalignment='bottom',
        transform=ax.transAxes)
ax.axline((0, lr.intercept), slope=lr.slope)
```

After these changes, restarting the kernel and running all cells will show that each of the four subplots shows both the correlation coefficient and the equation for a line that predicts  $y$  values from given  $x$  values.

The `axline()` function can be used to add the regression line to each subplot. We've provided a known point, the  $(0, b)$  intercept, and the slope,  $m$ . The line is automatically constrained to fit within the range of the axes. This may be little more than more visual clutter, or it may be helpful for understanding the relationship between the variables:

```
ax.axline((0, lr.intercept), slope=lr.slope)
```

## See also

- The *Ingesting data into a notebook* recipe earlier in this chapter has more details on ingesting data.
- The **matplotlib** examples pages include dozens of plots suitable for statistical analysis. See the <https://matplotlib.org/stable/gallery/statistics/index.html> Statistics Gallery web page for a large number of examples of data visualizations.
- See the **matplotlib** <https://matplotlib.org/stable/users/resources/index.html#books-chapters-and-articles> books, chapters and articles web page for books on data visualization.



## Adding details to markdown cells

The point of data analytics is to offer deeper insights into numeric measures to show trends and relationships. The object, in general, is to help someone make a decision that's fact-based. The decision can be as simple as deciding to recharge a vehicle before a trip, based on the anticipated distance and time available for charging. Alternatively, it can be as profound as responding to a medical diagnosis with effective therapies.

Visualization is one aspect of presenting data for an audience to help their understanding. Adjacent to visualization is organizing the presented material into a coherent story. Further, we also need to provide supplemental details beyond figures and images. See *The Manager's Guide to Presentations* from Packt Publishing for more on this topic.

## Getting ready

We'll update a notebook with some cells that contain Markdown formatting. We can start with a notebook created in one of the recipes earlier in this chapter. An alternative is to create a new, empty notebook that contains formatted text.

A notebook can be exported as a PDF file directly from Jupyter Lab. This is the quickest and easiest publication route. We may want to hide some code cells from these kind of publications.

For more polished results, it helps to use separate formatting tools. A notebook can be exported as a Markdown file (or Restructured Text or  $\LaTeX$ ). Appropriate programs like Pandoc, Docutils, or a  $\TeX$ toolset can then create documents from the exported notebook.

Tools like **Quarto** and **Jupyter {Book}** can also be used to create polished output.

The essential basics, however, are a sensible organization, clear writing, and Markdown-formatting in the notebook cells. An interesting part of using Markdown is that a cell's content is essentially static. The notebook's syntax won't take computed values and inject them into a Markdown cell.

There are two ways to create dynamic content:

- Install the Python Markdown extension. See Extensions in the Jupyter Lab documentation. After installing this extension, code can be included in a Markdown cell by surrounding it with `{` and `}`.
- Build the Markdown content in a code cell, and then render the results as Markdown. We'll look at this in a little more depth next.

## How to do it...

1. Import the required function and class from the `IPython.display` module:

```
from IPython.display import display, Markdown
```

2. Create a Markdown object with the text to render:

```
m = Markdown(rf"""
We can see that  $r = \text{quartet['I']}.correlation:.2f$ ; this is a strong
correlation.

This leads to a linear regression result with  $y = \text{r.slope:.1f}$ 
\times x +  $\text{r.intercept:.1f}$  as the best fit
for this collection of samples.
Interestingly, this is true for all four series in spite of the
dramatically distinct scatter plots.
""")
```

The triple-quoted string has two prefix characters, `r` and `f`. This is a “raw” formatted string. The formatted string is essential for injecting Python objects into text. See *Building complicated strings with f-strings* in *Chapter 1*.

A raw string is required because  $\LaTeX$  math formatting requires extensive use of `\` characters. In this context, we emphatically do not want Python to consider the `\` as an escape character; we need to ensure these characters are left alone, untouched, and provided to the Markdown engine without change.

Using raw strings means it's very difficult to include a newline character. Therefore, it's best to use a triple-quoted string that can span multiple lines.

3. Use the `display()` function to render the cell results as Markdown instead of unformatted text:

```
display(m)
```

This creates output in Markdown that includes the results of a computation.

## How it works...

Given a code cells that computes a result value, a notebook uses an object's `__repr__()` method to display the object. The object can have additional methods defined that are used by IPython to format the object in distinct ways. In this case, the use of the `Markdown` class creates an object that is rendered as nicely formatted text.

The `IPython.display` package contains a number of helpful functions. The `display()` function allows a Python code cell to interact with the browser-based rendering of the notebook.

The creation of the block of text and the `Markdown` object is part of the back-end number-crunching kernel that's running the notebook's code. From this, the rendered text is sent to the browser. This text can also be send to the other external tools for publishing a notebook, giving us nicely formatted cells with content computed by the notebook.

## There's more...

When we turn to sharing a notebook, we often have two distinct venues:

- Presentations, where the notebook has key points to back up a presenter's remarks to stakeholders.
- Publications, where the notebook – or a document produced from the notebook – is distributed to stakeholders.

In some instances, we'll need to create both a slide deck and a report. This requires some care to be sure that computed results are consistent among all variants of the notebook. One approach is to have two final report notebooks built around importing a core notebook that has data ingestion and computation features.

The `%run` magic command can be put into a cell to run a notebook and collect the result variables. This will also display the output from `print()` and any plots that are created. Because the output is displayed separately, the core notebook should focus on ingesting and computing results without any display features.

For presentations, use the **Property Inspector** on the right side of the Jupyter Lab page. This lets us set a cell's **Slide Type** for a presentation.

We can create Markdown content with the key points, the visualizations, and all the necessary supporting information. Once we have the content, we can mark the cells using the Property Inspector. Finally, we need to save the notebook as a presentation. In the **File** menu, the **Save and Export Notebook As...** menu item presents a list of alternatives. The **Reveal.js Slides** will create an HTML file with the slide presentation.

The exported HTML document can be opened in a browser to provide the supporting visuals for a presentation. It can be emailed to attendees who only want the presentation materials.

To create a final document (often in the PDF format), we have an array of choices:

- Export as AsciiDoc, Markdown, or Restructured Text. From these formats, tools like **Pandoc**, **Docutils**, or **Sphinx** can be used to create a final output file.
- Export as LaTeX. From this format, the  $\text{\TeX}$ tools need to be used to create a final PDF. These tools can be rather complicated to install and maintain, but the results are stellar.
- Export as a PDF. There may be a `webpdf` option, which uses Playwright and the Chromium library to render a PDF. There may also be a `Qtpdf` option, which uses the Qt library to create a PDF.

Tools like **Quarto** and **Jupyter {Book}** can also be used to create polished output. These include their own publication tools to create final, outstanding PDF documents from the Markdown in a notebook.

An important thing to remember about this publication pipeline is this imperative: don't

copy and paste from a notebook.



Copying results from a notebook into a word-processing document is a way to introduce errors and omissions.

Publishing directly from a notebook eliminates the possible errors caused by having two – potentially conflicting – copies of the computation results.

## See also

- See *Writing clear documentation strings with RST markup* in *Chapter 3* for more on using ReStructured Text to document code.
- See *Including descriptions and documentation* in *Chapter 2* for more on documentation for Python modules.
- See *Using pyplot to create a scatter plot* in this chapter for data analysis examples that require publication.

## Including Unit Test Cases in a Notebook

It's difficult to be sure that any software is trustworthy without a test suite. It can be awkward to unit test code in a Jupyter Notebook. One of the primary reasons testing is difficult is that a notebook is often used to ingest a very large volume of data. This means that computations in individual cells can take a very long time to complete. For a sophisticated machine learning model, this kind of time-consuming processing is typical.

One approach to creating test cases is to create a “template” notebook used for unit testing. The template can be cloned and a source Path value changed to read the large data that is of real interest.

Since notebook `.ipynb` files are in the JSON format, it's relatively easy to write a program to confirm that the cells of a notebook used to produce the desired results are (nearly) identical to the template notebook used for testing. Cells with specific filenames are expected to change; the rest are expected to remain intact.

A good notebook design transforms multi-statement cells into function (and class) definitions. This means the important results are computed by functions that have test cases. These test cases can be included in the function's docstring. We'll address doctest in depth in *Chapter 15*.

In addition to doctest examples for functions and classes, we can use the `assert` statement in a cell to confirm that a notebook's cells work as expected. This statement is a shorthand for an `if-raise` statement pair. If the expression in the `assert` statement is not true, an `AssertionError` is raised. This will stop the notebook, revealing a problem.

## Getting ready

We'll start with the notebook from *Using axes directly to create a scatter plot*, as it has a complicated cell to ingest data that can be converted into a function and some class definitions that can be supplemented with doctest examples.

## How to do it...

1. Refactor the data ingestion cell to be a function with a name like `ingest()`. The parameter should be the `Path` and the return value should be the dictionary with the four Anscombe series. The original side-effect of this cell will be created in a cell below. Here's the function definition:

```
def ingest(source: Path) -> dict[str, Series]:
    """
    >>> doctest example
    """
    with source.open() as source_file:
        json_document = json.load(source_file)
        source_data = (Series.model_validate(s) for s in
            json_document)
        quartet = {s.series: s for s in source_data}
    return quartet
```

We haven't filled in the doctest example; we've only left reminder text. These kinds of examples are the subject of recipes in *Chapter 15*.

2. Add a cell to ingest the test data:

```
source = Path.cwd().parent.parent / "data" / "anscombe.json"
quartet = ingest(source)
```

3. Add some assert statements to show the expected properties of the quartet object. These combine an expression and the expected output into a single statement:

```
assert len(quartet) == 4, f"read {len(quartet)} series"
assert list(quartet.keys()) == ["I", "II", "III", "IV"], f"keys were {list(quartet.keys())}"
```

Often, we'll replace an informal test cell with a more formal assertion. It's common to have a cell with an expression like `quartet.keys()`. When developing a notebook, we'll look at the results of this expression to confirm that the data ingestion worked. This manual test case can be upgraded with an automated test in the form of assert statements.

4. Be sure to save the notebook. We'll assume it's called `recipe_06.ipynb`.
5. Open a new terminal window and enter the following command:

```
(cookbook3) % jupyter execute src/ch12/recipe_06.ipynb
```

The notebook should execute flawlessly. There are two important lines in the output:

```
[NbClientApp] Executing src/ch12/recipe_06.ipynb
[NbClientApp] Executing notebook with kernel: python3
```

These lines confirm the file and the kernel being used. The absence of other output tells us no exceptions were raised.

## How it works...

The `jupyter execute` command will start a kernel and run the notebook's cells to completion. This is handy for confirming that it works.

We have to be sure to reject the *false negative* of a test procedure that fails to uncover a problem. To be sure the testing approach is sound, we can inject a failing assertion into the notebook and observe the expected error.

Add a cell like the following:

```
value = 355/113
assert value == 3.14, f"invalid {value}"
```

This will compute a value and then make a demonstrably false assertion about it. This will lead to a very visible failure when we use the `jupyter execute` command. The output will result in the following:

```
-----
AssertionError                                Traceback (most recent call last)
Cell In[4], line 2
      1 value = 355/113
----> 2 assert value == 3.14, f"invalid {value}"

AssertionError: invalid 3.1415929203539825
```

The OS status code will also be non-zero, indicating a failure to execute properly. This provides ample confirmation that an error will produce a noisy, explicit failure. Once we're sure this works, we can remove it from the notebook, confident that the other tests really will discover problems.

## There's more...

For the special case of comparing float values, we shouldn't use simple `==` comparison. As noted in *Choosing between float, decimal, and fraction*, float values are an approximation, and small changes to the order of operations can influence the right-most digits.

For float values, the `math.isclose()` function is essential. Look back at the notebook for *Using axes directly to create a scatter plot*. The `Series` class definition computed a correlation and a linear regression value. We might create a cell like the following to test this:



```
from math import isclose
test = Series(
    series="test",
    data=[Pair(x=2, y=4), Pair(x=3, y=6), Pair(x=5, y=10)]
)
assert isclose(test.correlation, 1.0)
assert isclose(test.regression.slope, 2.0)
assert isclose(test.regression.intercept, 0.0)
```

This test case creates a sample Series object. It then confirms that the results are very close to the target values. The default settings have a relative tolerance value of  $10^{-9}$ , which will include nine digits.

## See also

- *Chapter 15* covers testing and unit tests in some depth.
- In *Writing better docstrings with RST markup* in *Chapter 2*, the idea of doctest examples is also mentioned.
- See *Ingesting data into a notebook* for the seed notebook to which we want to add assertions.

## Join our community Discord space

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

<https://packt.link/dHrHU>



# 13

## Application Integration: Configuration

Python's concept of an extensible library gives us rich access to numerous computing resources. The language provides avenues to make even more resources available. This makes Python programs particularly strong at integrating components to create sophisticated composite processing. In this chapter, we'll address the fundamentals of creating complex applications: managing configuration files, logging, and a design pattern for scripts that permits automated testing.

These new recipes leverage ideas shown in recipes in other chapters. Specifically, in the *Using argparse to get command-line input*, *Using cmd to create command-line applications*, and *Using the OS environment settings* recipes in *Chapter 6*, some specific techniques for creating top-level (main) application scripts were shown. It may help to review those recipes to see examples of Python application scripts. In *Chapter 11*, we looked at filesystem input and output.

In this chapter, we'll look at a number of ways to handle configuration files. There are many file formats that can be used to store long-term configuration information:

- The INI file format as processed by the `configparser` module.
- The TOML file format is very easy to work with but requires an add-on module that's not currently part of the Python distribution. We'll look at this in the *Using TOML for configuration files* recipe.
- The properties file format is typical of Java programming and can be handled in Python without writing too much code. Some of the syntax overlaps with Python scripts and TOML files. A switch from the properties file format to TOML only requires changing any `name: value` to `name = "value"`, permitting use of the TOML parser.
- For Python scripts, a file with assignment statements looks a lot like a properties file, and is very easy to process using the `compile()` and `exec()` functions. We'll look at this in the *Using Python for configuration files* recipe.
- A Python module with class definitions is a variation that uses Python syntax but isolates the settings into separate classes. This can be processed with the `import` statement. We'll look at this in the *Using a class as a namespace for configuration* recipe.

Some recipes in this chapter will extend some of the concepts from *Chapter 7*, and *Chapter 8*. This chapter will apply those concepts to defining configuration files using classes.

It's important to consider the kinds of information required in configuration files. Carelessly including passwords or security tokens in a configuration file can be fatal to secure use of data. Including personal information in a configuration file is also a common security weakness. See the **Common Weakness Enumeration** for other more specific issues with poorly designed configuration files.

In this chapter, we'll look at the following recipes:

- *Finding configuration files*

- *Using TOML for configuration files*
- *Using Python for configuration files*
- *Using a class as a namespace for configuration*
- *Designing scripts for composition*
- *Using logging for control and audit output*

We'll start with a recipe for handling multiple configuration files that must be combined. This gives users some helpful flexibility. From there, we can dive into the specifics of a few of the common configuration file formats.

## Finding configuration files

Many applications will have a hierarchy of configuration options. The foundation of the hierarchy is often the default values built into the application. These might be supplemented by server-wide (or cluster-wide) values from centralized configuration files. There might also be user-specific files, or perhaps even configuration files provided when starting a program.

In many cases, configuration parameters are written in text files, so they are persistent and easy to change. The common tradition in Linux is to put system-wide configuration in the `/etc` directory. A user's personal changes would be in their home directory, often named `~username` or `$HOME`.

In this recipe, we'll see how an application can support a rich hierarchy of locations for configuration files.

## Getting ready

The example we'll use is an application to simulate dice rolling. The application is shown in several recipes throughout *Chapter 6*. Specifically, look at *Using argparse to get command-line input* and *Using cmd to create command-line applications*.

We'll follow the design pattern of the Bash shell, which looks for configuration files in the

following places:

1. It starts with the `/etc/profile` file, applicable to everyone using the system.
2. After reading that file, it looks for one of these files, in this order:
  - (a) `~/.bash_profile`
  - (b) `~/.bash_login`
  - (c) `~/.profile`

Other shells, like **zsh**, use some additional files but follow this pattern of working through a sequence of files in order.

In a POSIX-compliant operating system, the shell expands the `~` to be the home directory for the logged-in user. In general, the Python `pathlib` module handles this for Windows, Linux, and macOS automatically via the `Path.home()` method.

In later recipes, we'll look at ways to parse and process specific formats of configuration files. For the purposes of this recipe, we won't pick a specific format. Instead, we'll assume that an existing function, `load_config_file()`, has been defined that will load a specific configuration mapping from the contents of the configuration files.

The function looks like this:

```
def load_config_file(config_path: Path) -> dict[str, Any]:
    """Loads a configuration mapping object with the contents
    of a given file.
    :param config_path: Path to be read.
    :returns: mapping with configuration parameter value
    """
    # Details omitted.
```

We'll look at a number of different ways to implement this function.

## Why so many choices?

There's a side topic that sometimes arises when discussing this kind of design – why have so many choices? Why not specify exactly one place?

It's common to offer variations typical for one distribution, but atypical for another. Also, user expectations depend on software with which they're already familiar; this is very difficult to anticipate. And, of course, when dealing with Windows, there will be the possibility of yet more variant file paths that are unique to that platform. For these reasons, it's easier to offer multiple locations and permit the user or administrator to pick the one they prefer.

## How to do it...

We'll make use of the `pathlib` module to provide a handy way to work with files in various locations. We'll also use the `collections` module to provide the very useful `ChainMap` class:

1. Import the `Path` class and the `ChainMap` class. There are several type hints that are also required:

```
from pathlib import Path
from collections import ChainMap
from typing import TextIO, Any
```

2. Define an overall function to get the configuration files:

```
def get_config() -> ChainMap[str, Any]:
```

3. Create paths for the various locations of the configuration files. These are called pure paths and start with the names of *potential* files. We can decompose these locations into a system path and a sequence of local paths. Here are the two assignment statements:

```
system_path = Path("/etc") / "some_app" / "config"
local_paths = [
    ".some_app_settings",
    ".some_app_config",
]
```

4. Define the application's built-in defaults as a list of dictionaries:

```
configuration_items = [  
    dict(  
        some_setting="Default Value",  
        another_setting="Another Default",  
        some_option="Built-In Choice",  
    )  
]
```

Each individual configuration file is a mapping from keys to values. Each of these mapping objects is combined to form a list; this becomes the final ChainMap configuration mapping.

5. If the system-wide configuration file exists, load this file:

```
if system_path.exists():  
    configuration_items.append(  
        load_config_file(system_path))
```

6. Iterate through other locations looking for a file to load. This loads the first file that it finds and uses a break statement to stop after the first file is found:

```
for config_name in local_paths:  
    config_path = Path.home() / config_name  
    if config_path.exists():  
        configuration_items.append(  
            load_config_file(config_path))  
    break
```

7. Reverse the list and create the final ChainMap mapping:

```
configuration = ChainMap(  
    *reversed(configuration_items)  
)
```

The list needs to be reversed so that the local file (appended last) is searched first, then the system settings, and finally the application default settings. It's certainly possible to assemble the list in the reverse order to avoid the `reversed()` function;

we've left this possible change as an exercise for you.

Once we've built the configuration object, we can use the final configuration like a simple mapping. This object supports all of the expected dictionary operations.

## How it works...

In the *Creating dictionaries – inserting and updating* recipe in *Chapter 5*, we looked at the basics of using a dictionary. Here, we've combined several dictionaries into a chain. When a key is not located in the first dictionary of the chain, then later dictionaries in the chain are checked. This is a handy way to provide default values for each key in the mapping. Because the `ChainMap` is nearly indistinguishable from the built-in `dict` class, it permits a lot of flexibility in the implementation details: any kind of configuration file that can be read to create a dictionary is perfectly acceptable. The rest of the application can be based on the dictionary without being exposed to the details of how the configuration was built.

## There's more...

The subtle distinction between the single system-wide configuration file and the collection of alternative names for the local configuration files isn't ideal. This distinction between a singleton and a list of choices doesn't seem to serve any particular purpose. Often, we want to extend this design and the tiny asymmetry leads to complications.

We'll consider changing the configuration to have the following four tiers:

1. The built-in defaults.
2. A host-wide configuration in a central directory like `/etc` or `/opt`. This is often used for details of the OS or network context for this container.
3. A home directory configuration for the user running the app. This may be used for distinguishing test and production instances.
4. A local file in the current working directory. This may be used by a developer or tester.

This suggests a modification to the recipe to use nested lists of paths. The outer list contains



all of the tiers of configuration. Within each tier, a list will contain the alternative locations for a configuration file.

```

local_names = ('.some_app_settings', '.some_app_config')
config_paths = [
    [
        base / 'some_app' / 'config'
        for base in (Path('/etc'), Path('/opt'))
    ],
    [
        Path.home() / name
        for name in local_names
    ],
    [
        Path.cwd() / name
        for name in local_names
    ],
]

```

This `list[list[Path]]` structure provides three tiers of configuration files. Each of the tiers has a number of alternative names. The order of the tiers and the names within each tier are important. The lower tiers provide overrides to the upper tiers. We can then use nested `for` statements to examine all of the alternative locations.

```

def get_config_2() -> ChainMap[str, Any]:
    configuration_items = [
        DEFAULT_CONFIGURATION
    ]
    for tier_paths in config_paths:
        for alternative in tier_paths:
            if alternative.exists():
                configuration_items.append(
                    load_config_file(alternative))
                break
    configuration = ChainMap(
        *reversed(configuration_items)
    )
    return configuration

```

We've factored out the default configuration into a global variable with the name

DEFAULT\_CONFIGURATION. We have conspicuously left the collection of configuration paths with the name `config_paths`. It's not perfectly clear if this should be global (and have a global variable name in ALL-CAPITALS) or if this should be part of the `get_config()` function. We've adopted a bit of both by using a lowercase name and putting it outside the function.

The value of `config_paths` is unlikely to be needed elsewhere, making it a bad choice for being a global variable. It is, however, something that may change – perhaps in the next major release – and deserves to be exposed so it can be changed.

## See also

- In the *Using TOML for configuration files* and *Using Python for configuration files* recipes in this chapter, we'll look at ways to implement the `load_config_file()` function.
- In the *Mocking external resources* recipe in *Chapter 15*, we look at ways to test functions such as this, which interact with external resources.
- The `pathlib` module is central to this processing. This module provides the `Path` class definition, which provides a great deal of sophisticated information about the OS's files. For more information, see the *Using pathlib to work with filenames* recipe in *Chapter 11*.

## Using TOML for configuration files

Python offers a variety of ways to package application inputs and configuration files. We'll look at writing files in TOML notation because this format is elegant and simple. For more information on this format, see <https://toml.io/en/>.

Most TOML files look quite a bit like INI-format files. This overlap is intentional. When parsed in Python, a TOML file will be a nested dictionary structure.

We might have a file like this:

```
[some_app]
  option_1 = "useful value"
  option_2 = 42

[some_app.feature]
  option_1 = 7331
```

This will become a dictionary like the following:

```
{'some_app': {'feature': {'option_1': 7331},
              'option_1': 'useful value',
              'option_2': 42}}
```

The `[some_app.feature]` is called a “table”. The use of a `.` in the key creates a nested table.

## Getting ready

We’ll often use the *Finding configuration files* recipe, shown earlier in this chapter, to check a variety of locations for a given configuration file. This flexibility is often essential for creating an application that’s easy to use on a variety of platforms.

In this recipe, we’ll build the missing part of the *Finding configuration files* recipe, the `load_config_file()` function. Here’s the template that needs to be filled in:

```
def load_config_file_draft(config_path: Path) -> dict[str, Any]:
    """Loads a configuration mapping object with contents
    of a given file.

    :param config_path: Path to be read.

    :returns: mapping with configuration parameter values
    """
    # Details omitted.
```

In this recipe, we’ll fill in the space held by the `Details omitted` line to load configuration files in TOML format.

## How to do it...

This recipe will make use of the `tomllib` module to parse a YAML-TOML file:

1. Import the `tomllib` module along with the `Path` definition and the type hints required by the `load_config_file()` function definition:

```
from pathlib import Path
from typing import Any
import tomllib
```

2. Use the `tomllib.load()` function to load the TOML-syntax document:

```
import tomllib

def load_config_file(config_path: Path) -> dict[str, Any]:
    """Loads a configuration mapping object with contents
    of a given file.
    :param config_path: Path to be read.
    :returns: mapping with configuration parameter values
    """
    with config_path.open('b') as config_file:
        document = tomllib.load(config_file)
```

An unusual requirement of TOML parsing in Python requires us to open the file in “binary” mode when using the `load()` function. We can use `'rb'` as the mode to be explicit that the file is opened for reading.

The alternative is to use the `loads()` function on a block of text. It looks like this:

```
document = tomllib.loads(config_path.read_text())
```

This `load_config_file()` function produced the required dictionary structure. It can be fit into the design from the *Finding configuration files* recipe to load a configuration file using TOML syntax.

## How it works...

As noted above, the idea of TOML syntax is to be easy to read and map directly to a Python dictionary. There is some intentional overlap between TOML notation and INI file syntax. There is also some overlap with some aspects of property file syntax.

The core of TOML syntax is key-value pairings generally written as `key = value`. The keys include valid Python symbols. This means that any sort of dataclass or Pydantic structure with a dictionary mapping can be mapped into TOML syntax, too.

It's valid for a TOML key to have a hyphen, which is not part of allowed Python names. A key can be a quoted string, too. This permits quite a wide variety of alternative keys. These features may need some caution depending on ultimate use for the configuration dictionary objects.

A key can also be *dotted*; this will create sub-dictionaries. Here's an example of dotted keys:

```
some_app.option_1 = "useful value"  
some_app.option_2 = 42  
some_app.feature.option_1 = 7331
```

This looks quite a bit like a properties file often used with Java applications. This creates nested dictionaries by decomposing the keys at the `.` character.

A wide variety of values are available, including string values, integer values, float values, and Boolean values (using `true` and `false` as the literal values). Additionally, TOML recognizes ISO date-time strings; see RFC 3339 for the formats supported.

TOML permits two data structures:

- An array is enclosed in `[` and `]`. We can use `sizes = [1, 2, 3]` to create a Python `list` value.
- An in-line table can be created using `{` and `}` around one or more `key = value` items. For example, `sample = {x = 10, y = 8.4}` creates a nested `dict` value.

One of the most important features of TOML syntax is using `[table]` as the key for a nested dictionary. We'll often see this:

```
[some_app]
  option_1 = "useful value"
  option_2 = 42

[some_app.feature]
  option_1 = 7331
```

The `[some_app]` is a key for a dictionary containing the indented key-value pairs. TOML syntax of `[some_app.feature]` defines a more deeply nested dictionary. The use of a dotted key means the string "some\_app" will be a key for a dictionary containing the key `feature`. The value associated with this key will be a dictionary with the key "option\_1". In TOML the `[table]` prefix for nested values creates a visual organization, making it easier to find and change configuration settings.

## There's more...

TOML notation is used for a the overall `pyproject.toml` file that can be used to describe a Python project. This file often has two top-level tables: `[project]` and `[build-system]`. The `project` table will have some metadata about the `[project]`. Here's an example:

```
[project]
name = "python_cookbook_3e"
version = "2024.1.0"
description = "All of the code examples for Modern Python Cookbook, 3rd Ed."
readme = "README.rst"
requires-python = ">=3.12"
license = {file = "LICENSE.txt"}
```

The `[build-system]` table provides information on tools needed to install the module, package, or application. Here's an example:

```
[build-system]
build-backend = 'setuptools.build_meta'
requires = [
    'setuptools',
]
```

This file provides a few essential pieces of information about the project. The use of TOML notation makes it relatively easy to read and change.

## See also

- See the *Finding configuration files* recipe earlier in this chapter to see how to search multiple filesystem locations for a configuration file. We can easily have application defaults, system-wide settings, and personal settings built into separate files and combined by an application.
- For more information on TOML syntax, see <https://toml.io/en/>.
- For more information on the `pyproject.toml` file, see the **Python Packaging Authority** document <https://pip.pypa.io/en/stable/reference/build-system/pyproject-toml/>.

## Using Python for configuration files

In addition to syntax like TOML for providing configuration data, we can also write files in Python notation; it's elegant and simple. It offers tremendous flexibility, since the configuration file is a Python module.

### Getting ready

Python assignment statements are particularly elegant for creating configuration files. The syntax can be simple, easy to read, and extremely flexible. If we use assignment statements, we can import an application's configuration details from a separate module. This could have a name like `settings.py` to show the module's focus on configuration parameters.

Because Python treats each imported module as a global **Singleton** object, we can have

several parts of an application all use an `import settings` statement to get a consistent view of the current, global application configuration parameters. We don't need to worry about managing an object using the **Singleton** design pattern, since that's already part of Python.

We'd like to be able to provide definitions in a text file that look like this:

```
"""Weather forecast for Offshore including the Bahamas
"""

query = {'mz':
         ['ANZ532',
          'AMZ117',
          'AMZ080']}

base_url = "https://forecast.weather.gov/shmrn.php"
```

This configuration is a Python script. The parameters include two variables, `query` and `base_url`. The value of the `query` variable is a dictionary with a single key, `'mz'`, and a sequence of values.

This can be seen as a specification for a number of related URLs that are all similar to `http://forecast.weather.gov/shmrn.php?mz=ANZ532`.

We'll often use the *Finding configuration files* recipe to check a variety of locations for a given configuration file. This flexibility is often essential for creating an application that's easily used on a variety of platforms.

In this recipe, we'll build the missing part of the *Finding configuration files* recipe, the `load_config_file()` function. Here's the template that needs to be filled in:

```
def load_config_file_draft(config_path: Path) -> dict[str, Any]:
    """Loads a configuration mapping object with contents
    of a given file.

    :param config_path: Path to be read.
```



```
:returns: mapping with configuration parameter values
"""
# Details omitted.
```

In this recipe, we'll fill in the space held by the `# Details omitted` line to load configuration files in Python format.

## How to do it...

We can make use of the `pathlib` module to locate the files. We'll also leverage the built-in `compile()` and `exec()` functions to process the code in the configuration file:

1. Import the `Path` definition and the type hints required by the `load_config_file()` function definition:

```
from pathlib import Path
from typing import Any
```

2. Use the built-in `compile()` function to compile the Python module into an executable form. This function requires the source text as well as the filename from which the text was read. The filename is essential for creating trace-back messages that are useful and correct:

```
def load_config_file(config_path: Path) -> dict[str, Any]:
    code = compile(
        config_path.read_text(),
        config_path.name,
        "exec")
```

In rare cases where the code doesn't come from a file, the general practice is to provide a name such as `<string>` for the filename.

3. Execute the code object created by the `compile()` function. This requires two contexts. The global context provides any previously imported modules, plus the `__builtins__` module. The local context is the `locals` dictionary; this is where new

variables will be created:

```
locals: dict[str, Any] = {}
exec(
    code,
    {"__builtins__": __builtins__},
    locals
)
return locals
```

This `load_config_file()` function produces the required dictionary structure. It can be fit into the design from the *Finding configuration files* recipe to load a configuration file using Python syntax.

## How it works...

The details of the Python language – the syntax and semantics – are embodied in the built-in `compile()` and `exec()` functions. The three essential steps are these:

1. Read the text.
2. Compile the text with the `compile()` function to create a code object.
3. Use the `exec()` function to execute the code object.

The `exec()` function reflects the way Python handles global and local variables. There are two namespaces (mappings) provided to this function. These are visible via the `globals()` and `locals()` functions.

We can provide two distinct dictionaries to the `exec()` function:

- A dictionary of global objects. The most common use is to provide access to the imported modules, which are always global. The `__builtins__` module can be provided in this dictionary. In some cases, other modules like `pathlib` should be added.
- A dictionary for the locals that will be created (or updated) by each assignment statement. This local dictionary allows us to capture the variables created when executing the `settings` module.

The `locals` dictionary will be updated by the `exec()` function. We don't expect the `globals` to be updated and will ignore any changes that happen to this collection.

## There's more...

This recipe suggests a configuration file is entirely a sequence of `name = value` assignment statements. The assignment statement is in Python syntax, as are the variable names and the literal syntax. This permits the configuration to leverage Python's large collection of built-in types. Additionally, the full spectrum of Python statements is available. This leads to some engineering trade-offs.

Because any statement can be used in the configuration file, it can lead to complexity. If the processing in the configuration file becomes too complex, the file ceases to be configuration and becomes a first-class part of the application. Very complex features should be implemented by modifying the application programming, not hacking around with the configuration settings. Python applications include the full source, as it is generally easier to fix the source than create hyper-complex configuration files. The goal is for a configuration file to provide values to tailor operations, not provide plug-in functionality.

We might want to include the OS environment variables as part of the global variables used for configuration. Doing this helps ensure the configuration values match the current environment settings. This can be done with the `os.environ` mapping.

It can also be sensible to do some processing for related settings. For example, it can be helpful to write a configuration file with a number of adjacent paths like this:

```
"""Config with related paths"""
base = Path(os.environ.get("APP_HOME", "/opt/app"))
log = base / 'log'
out = base / 'out'
```

In many cases, the `settings` file is edited by a person who can be trusted. Mistakes do happen, though, and it's wise to be careful about what functions are available in the dictionary of `globals` provided to the `exec()` function. Providing the narrowest set of

functions to support configuration is the recommended practice.

## See also

- See the *Finding configuration files* recipe earlier in this chapter to learn how to search multiple filesystem locations for a configuration file.

## Using a class as a namespace for configuration

Python offers a variety of ways to package application inputs and configuration files. We'll continue to look at writing files in Python notation because it's elegant and the familiar syntax can lead to easy-to-read configuration files. A number of projects allow us to use a class definition to provide configuration parameters. This uses Python syntax, of course. It also uses the class definition as a namespace to allow multiple configurations to be provided in a single module. The use of a class hierarchy means that inheritance techniques can be used to simplify the organization of parameters.

This avoids the use of a `ChainMap` to permit user-specific overrides of generic settings. Instead, this uses ordinary inheritance.

We're never going to create instances of these classes. We're going to use the attributes of the class definition and rely on class inheritance methods to track down the appropriate value for an attribute. This will differ from the other recipes in this chapter because it will produce a `ConfigClass` object, instead of a `dict[str, Any]` object.

In this recipe, we'll look at how we can represent configuration details in Python class notation.

## Getting ready

Python notation for defining the attributes of a class can be simple, easy to read, and reasonably flexible. We can, with a little work, define a sophisticated configuration language that allows someone to change configuration parameters for a Python application quickly and reliably.

We can base this language on class definitions. This allows us to package a number of

configuration alternatives in a single module. An application can load the module and pick the relevant class definition from the module.

We'd like to be able to provide definitions that look like this:

```
class Configuration:
    """
    Generic Configuration with a sample query.
    """
    base = "https://forecast.weather.gov/shmrn.php"
    query = {"mz": ["GMZ856"]}
```

We can create this class definition in a `settings.py` file to create a settings module. To use the configuration, the main application could do this:

```
>>> from settings import Configuration
>>> Configuration.base
'https://forecast.weather.gov/shmrn.php'
```

The application will gather the settings using the module name of settings with a class name of Configuration.

The configuration file locations follow Python's rules for finding modules. Rather than implementing our own search for the configuration, we can leverage Python's built-in search of `sys.path`, and the use of the `PYTHONPATH` environment variable.

In this recipe, we'll build a missing part that's similar to the *Finding configuration files* recipe, the `load_config_file()` function. However, there will be an important difference: we'll return an object instead of a dictionary. We can then refer to configuration values by attribute name instead of using the more cumbersome notation of a dictionary. Here's the revised template that needs to be filled in:

```
ConfigClass = type[object]

def load_config_file_draft(
    config_path: Path, classname: str = "Configuration"
```

```
) -> ConfigClass:
    """Loads a configuration mapping object with contents
    of a given file.

    :param config_path: Path to be read.

    :returns: mapping with configuration parameter values
    """
    # Details omitted.
```

We've used a similar template in a number of recipes in this chapter. For this recipe, we've added a parameter to this definition and changed the return type. The `classname` parameter is not present in previous recipes, but it is used here to select one of the classes from a module at the location in the filesystem named by the `config_path` parameter.

## How to do it...

We can make use of the `pathlib` module to locate the files. We'll leverage the built-in `compile()` and `exec()` functions to process the code in the configuration file:

1. Import the `Path` definition and the type hints required by the `load_config_file()` function definition:

```
from pathlib import Path
import platform
```

2. Use the built-in `compile()` function to compile the Python module into an executable form. This function requires the source text as well as a filename from which the text was read. The filename is essential for creating trace-back messages that are useful and correct:

```
def load_config_file(
    config_path: Path, classname: str = "Configuration"
) -> ConfigClass:
    code = compile(
        config_path.read_text(),
        config_path.name,
```

```
"exec")
```

- Execute the code object created by the `compile()` method. We need to provide two contexts. The global context can provide the `__builtins__` module, plus the `Path` class and the `platform` module. The local context is where new variables will be created:

```
globals = {
    "__builtins__": __builtins__,
    "Path": Path,
    "platform": platform}
locals: dict[str, ConfigClass] = {}
exec(code, globals, locals)
return locals[classname]
```

This locates the named class in the `locals()` mapping and returns the class as the configuration object. This does not return a dictionary.

This variation on the `load_config_file()` function produces a useful structure that can be accessed using attribute names. It does not provide the design expected by the *Finding configuration files* recipe. The resulting configuration object does – because it uses attribute names – more useful than a simple dictionary.

## How it works...

We can load a Python module by using `compile()` and `exec()`. From the module, we can extract an individual class name that contains the various application settings. Overall, it looks like the following example:

```
>>> configuration = load_config_file(
... Path('src/ch13/settings.py'), 'Chesapeake')

>>> configuration.__doc__.strip()
'Weather for Chesapeake Bay'
>>> configuration.query
{'mz': ['ANZ532']}
```

```
>>> configuration.base
'https://forecast.weather.gov/shmrn.php'
```

We can put any kind of object into the attributes of the configuration class. Our example showed lists of strings and strings. Any object of any class becomes a possibility when using class definitions.

We can have complex calculations within the class statement. We can use this to create attributes that are derived from other attributes. We can execute any kind of statement, including if statements and for statements, to create attribute values.

We will not, however, create an instance of the given class. Tools like **Pydantic** will validate instances of a class, but aren't helpful for validating a class definition. Any kind of validation rules would have to be defined in a metaclass that is used to build the resulting configuration class. Additionally, ordinary methods of the class will not be used. If a function-like definition is needed, it would have to be decorated with `@classmethod` to be useful.

## There's more...

Using a class definition means that we will leverage inheritance to organize the configuration values. We can easily create multiple subclasses of `Configuration`, one of which will be selected for use in the application.

The configuration might look like this:

```
class Configuration:
    """
    Generic Configuration with a sample query.
    """
    base = "https://forecast.weather.gov/shmrn.php"
    query = {"mz": ["GMZ856"]}

class Bahamas(Configuration):
    """
```



```

    Weather forecast for Offshore including the Bahamas
    """
    query = {"mz": ["AMZ117", "AMZ080"]}

class Chesapeake(Configuration):
    """
    Weather for Chesapeake Bay
    """
    query = {"mz": ["ANZ532"]}

```

Our application must choose an appropriate class from the available classes in the `settings` module. We might use an OS environment variable or a command-line option to specify the class name to use. The idea is that our program can be executed like this:

```
(cookbook3) % python3 some_app.py -c settings.Chesapeake
```

This would locate the `Chesapeake` class in the `settings` module. Processing would then be based on the details in that particular configuration class. This idea leads to an extension to the `load_config_class()` function.

In order to pick one of the available classes, we can separate the module name and class name by looking for a `"."` separator in the command-line argument value:

```

import importlib

def load_config_class(name: str) -> ConfigClass:
    module_name, _, class_name = name.rpartition(".")
    settings_module = importlib.import_module(module_name)
    result: ConfigClass = vars(settings_module)[class_name]
    return result

```

Rather than manually compiling and executing the module, we've used the higher-level `importlib` module. This module contains functions implementing the `import` statement semantics. The requested module is imported, then compiled and executed, and the resulting module object is assigned to the variable named `result`.

Now we can use this function as follows:

```
>>> configuration = load_config_class(
... 'settings.Chesapeake')

>>> configuration.__doc__.strip()
'Weather for Chesapeake Bay'
>>> configuration.query
{'mz': ['ANZ532']}
>>> configuration.base
'https://forecast.weather.gov/shmrn.php'
```

We've located the Chesapeake configuration class in the `settings` module and extracted the various settings the application needs from this class.

## See also

- We'll look at class definitions in detail in *Chapter 7*, and *Chapter 8*.
- See the *Finding configuration files* recipe in this chapter for an alternative approach that doesn't use class definitions.

## Designing scripts for composition

An important part of overall application design is creating a script that can process command-line arguments and configuration files. Further, it's very important to design a script so that it can be tested as well as combined with other scripts into a composite application.

The idea is that many good ideas evolve through a series of stages. One such evolution might be the following path:

1. The idea starts as a collection of separate notebooks for separate parts of a larger task.
2. After the initial period of exploration and experimentation, this becomes a simple repetitive task. Rather than open and click manually to run the notebook, it's saved into a script file. Then the script files can be run from the command line.

3. After an initial period of making this a regular part of the organization's operations, the three-part script needs to be consolidated into a single script. At this point, refactoring is needed.

The most painful time to refactor is *after* combining a number of scripts into a single application and uncovering unexpected problems. This often happens because global variables will be shared when multiple scripts are integrated.

A much less painful time is earlier in the life of the project. As soon as a script is created, some effort should be made to design the script for testing and composition into a larger application.

## Getting ready

In this recipe, we'll look at what constitutes a good design for a script. In particular, we want to be sure that parameters and configuration files are considered in the design.

The target is to have a structure like the following:

- A docstring for the module or script as a whole.
- The imports. There's an internal ordering to these. Tools like **isort** and **ruff** can handle this.
- The class and function definitions that apply to the script.
- A function to gather the configuration file options and runtime parameters into a single object that can be used by other classes and functions.
- A single function that does the useful work. This is often called `main()`, but there's nothing sacred about this name.
- A small block of code that is executed **only** when the module is run as a script, and **never** when the module is imported:

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

## How to do it...

With the target design as our goal, here is one approach:

1. Start by writing a summary docstring at the top of the file. It's important to start with something and add details later. Here's an example:

```
"""  
    Some Script.  
    What it does. How it works.  
    Who uses it. When do they use it.  
"""
```

2. The `import` statements go after the docstring. It's not always possible to foresee all of the imports in advance. As the module is being written and modified, imports will be added and removed.
3. The class and function definitions go next. The order is important for resolving the type names in the `def` or `class` statements. This means the most fundamental type definitions must go first.

Again, it's not always possible to write all the definitions in their proper order during the first wave of design. What's important is keeping them together in a logical organization and rearranging them so the order makes sense to someone reading the code.

4. Write a function (with a name like `get_config()`) to get all of the configuration parameters. Generally, there are two parts to this; sometimes they need to be decomposed into two separate functions because each part can be rather complicated.
5. Then we have the `main()` function. This does the essential work of the script. When evolving from a notebook, this can be built from the sequence of cells.
6. Add the *Main-Import Switch* code block at the end:

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

The resulting module will work properly as a script. It can also be tested more easily because testing tools like **pytest** can import the module without it making changes to the filesystem when it tries to starting processing data. It can be integrated with other scripts to create a useful composite application.

## How it works...

The core consideration in designing a script is distinguishing between two use cases for a module:

- When run from the command line. In this case, the built-in global variable `__name__` will have a value of `"__main__"`.
- When imported for testing or as part of a larger, composite application. In this case, `__name__` will have a value that is the name of the module.

When a module is imported, we do not want it to start doing work. During import, we don't want the module to open files, read data, do computations, or produce output. All of this work is something that can only happen when the module is run as the main program.

The original cells of the notebook or script statements are now part of the body of the `main()` function, so the script will work properly. It will, however, it will also be in a form that can be tested. It can also be integrated into a larger and more sophisticated application.

## There's more...

When starting the conversion to an application, the `main()` function is often quite lengthy. There are two ways to make the processing clear:

- Large, prominent *billboard* comments
- Refactoring to create a number of smaller functions

We might start with a script that has comments like these:

```
# # Some complicated process
#
# Some additional markdown details.

# In[12]:

print("Some useful code here")

# In[21]:

print("More code here")
```

The In[n] : comments are provided by JupyterLab to identify the cells in a notebook. We can create billboard comments like these:

```
#####
# Some complicated process      #
#                               #
# Some additional markdown details.#
#####

print("Some useful code here")

#####
# Another step in the process    #
#####

print("More code here")
```

This is less than ideal. It's an acceptable temporary measure, but these steps should be proper functions, each with a docstring and test cases. Billboard comments are traditional in languages that don't have proper docstrings and lack documentation generators that exploit the docstrings.

Python has docstrings and several tools – like **Sphinx** – to create documentation from the docstrings.

## See also

- See the *Using argparse to get command-line input* recipe in *Chapter 6*, for background on using `argparse` to get inputs from a user.
- See *Using TOML for configuration files*, *Using Python for configuration files*, and *Using a class as a namespace for configuration* in this chapter for recipes related to configuration files.
- The *Using logging for control and audit output* recipe later in this chapter looks at logging.
- In the *Combining two applications into one* recipe in *Chapter 14*, we'll look at ways to combine applications that follow this design pattern.
- The details of testing and integration are covered in separate chapters. See *Chapter 15* for details on creating tests. See *Chapter 14* for details on combining applications.

## Using logging for control and audit output

When we consider an application, we can decompose the overall computation into three distinct aspects:

- Gathering input
- The essential processing that transforms the input into the output
- Producing output

There are several different kinds of output that applications produce:

- The main output that helps a user make a decision or take action. In some cases, this might be a JSON-formatted document downloaded by a web server. It might be a more complicated collection of documents that – together – will create a PDF file.
- Control information that confirms that the program worked completely and correctly.
- Audit summaries that can be used to track the history of state changes in a persistent databases.

- Any error messages that indicate why the application didn't work.

It's less than optimal to lump all of these various aspects into `print()` requests that write to standard output. Indeed, it can lead to confusion because too many different outputs can be interleaved in a single stream.

The OS provides each running process with two output files, standard output and standard error. These are visible in Python through the `sys` module with the names `sys.stdout` and `sys.stderr`. By default, the `print()` function writes to the `sys.stdout` file. We can change the target file and write the control, audit, and error messages to `sys.stderr`. This is an important step in the right direction.

Python also offers the logging package, which can be used to direct the ancillary output to a separate file (and/or other output channels, such as a database). It can also be used to format and filter that additional output.

In this recipe, we'll look at good ways to use the logging module.

## Getting ready

One approach to meeting a variety of output needs is to create multiple loggers, each with a different intent. It's common to name loggers around the module or class associated with the logger. We can also name loggers around an overall purpose, like audit or control.

The names of loggers form a hierarchy, punctuated by `..`. The *root* logger is the parent of all loggers and has a name of `""`. This suggests that we can have families of loggers focused on particular classes, modules, or features.

A set of top-level loggers can include a number of separate focus areas, including:

- `error` will preface all loggers for warnings and errors.
- `debug` will preface all loggers for debugging messages.
- `audit` will name loggers with counts and totals used to confirm that data was processed fully.



- `control` will name loggers that provide information about when the application was run, the environment, configuration files, and command-line argument values.

In most cases, it is helpful to have errors and debugging in a single logger. In other cases – for example, a web server – the request error response log should be separate from any internal error or debugging log.

A complicated application might have several loggers with names like `audit.input` and `audit.output` to show the counts of data consumed and the counts of data produced. Keeping these separate can help focus attention on problems with data providers.

A severity level serves as a kind of filter for each logger. The severity levels defined in the logging package include the following:

**DEBUG** : These messages are not generally shown since their intent is to support debugging. Above, we suggested this is a distinct variety of debugging. We suggest an application create a logging debugger, and use ordinary **INFO** messages for the debugging entries.

**INFO** : These messages provide information on the normal, happy-path processing.

**WARNING** : These messages indicate that processing may be compromised in some way. The most sensible use case for a warning is when functions or classes have been deprecated: they still work, but they should be replaced.

**ERROR** : Processing is invalid and the output is incorrect or incomplete. In the case of a long-running server, an individual request may have problems, but the server as a whole can continue to operate.

**CRITICAL** : A more severe level of error. Generally, this is used by long-running servers where the server itself can no longer operate and is about to crash.

Each logger has method names that are similar to the severity levels. We use the `info()` method to write a message with the **INFO** severity level.

For error handling, the severity levels are mostly appropriate. A debugging logger, however, often produces volumes of data that need to be kept separate. Further, any audit and control

output doesn't seem to have a severity level. The severity level seems to be focused only on error logging. For this reason, it seems to be better to have distinct logs with names like `debug.some_function`. We can then configure debugging by enabling or disabling the output from these loggers, and configure the severity level to be `INFO`.

## How to do it...

We'll look at logging in a class as well as a function in two mini-recipes.

### Logging in a class

1. Be sure the logging module is imported.
2. In the `__init__()` method, include the following to create an error and debug loggers:

```
self.err_logger = logging.getLogger(
    f"error.{self.__class__.__name__}")
self.dbg_logger = logging.getLogger(
    f"debug.{self.__class__.__name__}")
```

3. In any method that might require future debugging, use the debug logger's methods to write details to the log. While f-strings can be used to write log messages, they involve a bit of overhead to interpolate values into the text. Using the logger's formatting options and separate argument values involves slightly less computation when the configuration silences the logger's output:

```
self.dbg_logger.info(
    "Some computation with %r", some_variable)
# Some complicated computation with some_variable
self.dbg_logger.info(
    "Result details = %r", result)
```

4. In a few key places, include overall status messages. These are often in the overall application control classes:

```
# Some complicated input processing and parsing
self.err_logger.info("Input processing completed.")
```

## Logging in a function

1. Be sure the logging module is imported.
2. For larger and more complicated functions, it makes sense to include the logger inside the function:

```
def large_and_complicated(some_parameter: Any) -> Any:
    dbg_logger = logging.getLogger("debug.large_and_complicated")
    dbg_logger.info("some_parameter= %r", some_parameter)
```

Because the loggers are cached, only the first request to `get_logger()` involves any significant overhead. All subsequent requests are dictionary lookups.

3. For smaller functions, it can make sense to have a globally defined logger. This can help to reduce visual clutter within a function's body:

```
very_small_dbg_logger = logging.getLogger("debug.very_small")

def very_small(some_parameter: Any) -> Any:
    very_small_dbg_logger.info("some_parameter= %r", some_parameter)
```

Note that without any further configuration, no output will be produced. This is because the default severity level for each logger will be `WARNING`, which means the handler will not show `INFO`- or `DEBUG`-level messages..

## How it works...

There are three parts to introducing logging into an application:

- Creating `Logger` objects with the `getLogger()` function.
- Placing log messages near important state changes with one of the methods similar to `info()` or `error()` for each logger.
- Configuring the logging system as a whole when the application is run. This is essential for seeing output from the loggers. We'll look at this in the *There's more...* section of this recipe.

Creating loggers can be done in a variety of ways. A common approach is to create one logger with the same name as the module:

```
logger = logging.getLogger(__name__)
```

For the top-level main script, this will have the name `__main__`. For imported modules, the name will match the module name.

In more complex applications, there could be a variety of loggers serving a variety of purposes. In these cases, simply naming a logger after a module may not provide the required level of flexibility.

It's also possible to use the logging module itself as the root logger. This means a module can use the `logging.info()` function, for example. This isn't recommended because the root logger is anonymous, and we sacrifice the possibility of using the logger name as an important source of information.

This recipe suggests naming loggers based on the audience or use case. The topmost name – for example, `debug`. – will distinguish the audience or purpose for the log. This can make it easy to route all loggers under a given parent to a specific handler.



It's helpful to associate logging messages with the important state changes made by the code.

The third aspect of logging is configuring the loggers so that they route the requests to the appropriate destination. By default, with no configuration at all, the logger instances will quietly ignore the various messages being created.

With a minimal configuration, we can see all of the log events on the console. This can be done with the following:

```
if __name__ == "__main__":  
    logging.basicConfig(level=logging.INFO)
```

## There's more...

In order to route the different loggers to different destinations, we'll need a more sophisticated configuration. Often, this goes beyond what we can build with the `basicConfig()` function. We'll need to use the `logging.config` module and the `dictConfig()` function. This can provide a complete set of configuration options. The easiest way to use this function is to write the configuration in TOML:

```
version = 1
[formatters.default]
    style = "{"
    format = "{levelname} : {name} : {message}"

[formatters.timestamp]
    style = "{"
    format = "{asctime} // {levelname} // {name} // {message}"

[handlers.console]
    class = "logging.StreamHandler"
    stream = "ext://sys.stderr"
    formatter = "default"

[handlers.file]
    class = "logging.FileHandler"
    filename = "data/write.log"
    formatter = "timestamp"

[loggers]
    overview_stats.detail = {handlers = ["console"]}
    overview_stats.write = {handlers = ["file", "console"]}
    root = {level = "INFO"}
```

Here are some key points in this TOML configuration:

- The value of the `version` key must be 1. This is required.
- The values in the `formatters` table define the log formats available. If a formatter is not specified, a built-in formatter will display the message body only:
  - The default formatter defined in the example mirrors the format created by

the `basicConfig()` function. This includes the message severity level and the logger name.

- The new `timestamp` formatter defined in the example is a more complex format that includes the date-time stamp for the record. To make the file easier to parse, a column separator of `//` was used.
- The `handlers` table defines the handlers available for loggers to use:
  - The `console` handler writes to the `sys.stderr` stream and uses default formatter. The text starting `"ext://..."` is how a configuration file can refer to objects defined in the Python environment – in this case, the `sys.stderr` value from the `sys` module.
  - The `file` handler uses the `FileHandler` class to write to a file. The default mode for opening the file is `a`, which will append to any existing log file. The configuration specifies the `timestamp` formatter that will be used for the file.
- The `loggers` table provides a configuration for two specific named loggers that the application will use. Any logger name that begins with `overview_stats.detail` will be handled only by the `console` handler. Any logger name that begins with `overview_stats.write` will go to both the `file` handler and the `console` handler.
- The special `root` key defines the top-level logger. Within an application, it has a name of `"` (empty string) when referred to in code. Within the configuration file, it has the key `root`.

Setting the severity level on the root logger will set the level used to show – or hide – messages for all of the children of this logger. This will show messages with the severity `INFO` or higher, which includes warnings, errors, and severe errors.

Assuming the contents of this file are present in a variable named `config_toml`, the configuration to wrap the `main()` function will look like this:

```
if __name__ == "__main__":  
    logging.config.dictConfig(  
        tomlib.loads(config_toml))  
    main()  
    logging.shutdown()
```

This will start the logging in a known state. It will do the processing of the application. It will finalize all of the logging buffers and properly close any files.

## See also

- See the *Designing scripts for composition* recipe earlier in this chapter for the complementary part of this application.
- See the *Using TOML for configuration files* recipe in this chapter for more on parsing TOML documents.

## Join our community Discord space

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

<https://packt.link/dHrHU>



# 14

## Application Integration: Combination

The Python language is designed to permit extensibility. We can create sophisticated programs by combining a number of smaller components. In this chapter, we'll look at ways to combine modules and scripts.

We'll look at the complications that can arise from composite applications and the need to centralize some features, like command-line parsing. This will enable us to create uniform interfaces for a variety of closely related programs.

We'll extend some of the concepts from *Chapter 7* and *Chapter 8*, and apply the idea of the **Command** design pattern to Python programs. By encapsulating features in class definitions, we'll find it easier to combine and extend programs.

In this chapter, we'll look at the following recipes:

- *Combining two applications into one*



- *Combining many applications using the **Command** design pattern*
- *Managing arguments and configuration in composite applications*
- *Wrapping and combining CLI applications*
- *Wrapping a program and checking the output*

We'll start with a direct approach to combining multiple Python applications into a single, more sophisticated application. We'll expand this to apply object-oriented design techniques and create an even more flexible composite. Then, we'll apply uniform command-line argument parsing for composite applications.

## Combining two applications into one

For this recipe, we'll look at two scripts that need to be combined. One script emits data from a Markov chain process, and the second script summarizes those results.

What's important here is the Markov chain application is (intentionally) a bit mysterious. For the purposes of several recipes, we'll treat this as opaque software, possibly written in another language.

(The GitHub repository for this book has the Markov chain written in Pascal to be reasonably opaque.)

For reference, here's a depiction of the Markov chain state changes:

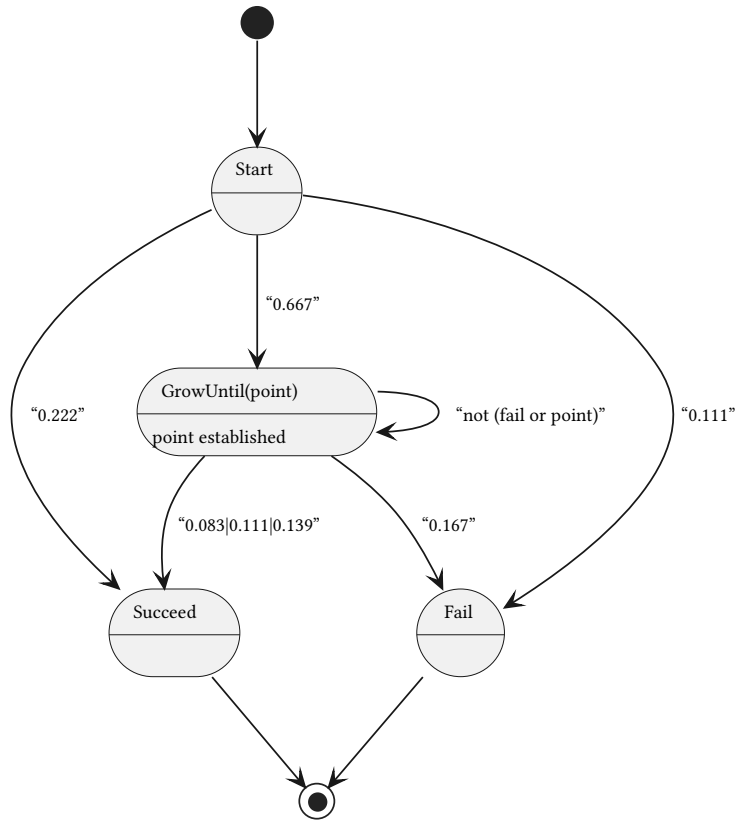


Figure 14.1: Markov chain states

The **Start** state will either succeed, fail, or generate a “point” value. There are a number of values, each with distinct probabilities that sum to  $P = 0.667$ . The **GrowUntil** state generates values that may match the point, not match the point, or indicate failure. In the cases of a non-match and non-failure, the chain transitions back to this state. The exact probability of a match depends on the starting-point value, which is why the state transition is labeled with three probabilities.

The generator application emits a TOML-format file with some configuration details and a collection of individual samples. The file looks like this:

```
# file = "../../data/ch14/data.csv"
# samples = 10
# randomize = 0
# -----
outcome,length,chain
"Success",1,"7"
"Success",2,"10;10"
```

A summary application reads all of these generated files to create some simple statistics to describe the raw data. This summary was originally done with Jupyter Notebook. While these can be executed with the `jupyter execute` command, an alternative approach is to save the notebook as a script and then execute the script.

We want to be able to combine this generator and the summary applications to reduce the manual steps in using the generator. There are several common approaches to combining multiple applications:

- A shell script can run the generator application and then run the summary application.
- A Python program can implement the high-level operation, using the `runpy` module to run each of the two applications.
- We can build a composite application from the essential components of each application.

In this recipe, we'll look at the third path of combining the essential components of each application by writing a new composite application.

## Getting ready

In the *Designing scripts for composition* and *Using logging for control and audit output* recipes in *Chapter 13*, we followed a design pattern that separated the input gathering, the essential processing, and the production of output. The objective of that design pattern was to gather the interesting pieces together to combine and recombine them into higher-level constructs.

Note that we have a tiny mismatch between the two applications. We can borrow a phrase

from database engineering (and also electrical engineering) and call this an “impedance mismatch.”

When building this composite application, the impedance mismatch is a cardinality problem. The data generator process is designed to run more frequently than the statistical summary process. We have a couple of choices for addressing issues such as this one:

- **Total redesign:** We can rewrite the generator to an iterator as desired, producing multiple sets of samples.
- **Add the iterator:** We can build the composite application to do bulk data generation processing. After all the data is produced, the composite application can then summarize it.

The choice between these design alternatives depends on the user stories for this application. It may also depend on the established base of users. For this recipe, the users would like to follow the *Add the iterator* design to create a composite process without touching the underlying generator.

Looking inside the two module implementation choices, we see two distinct design patterns for top-level applications:

- The `markov_gen` module has the following `main()` function definition:

```
def main(argv: list[str] = sys.argv[1:]) -> None:
```

- The `markov_summ` module, on the other hand, is a script, exported from a notebook. A direct **Command-Line Interface (CLI)** is not part of this script, and some rewriting is required. See the *Designing scripts for composition* recipe in *Chapter 13* for details on this.

To create a more useful script, we need to add a `def main():` line and indent the entire script inside the body of this function. At the end of the indented `main()` function, the `if __name__ == "__main__":` block can be added. Without creating a function that can be imported, the script is very difficult to test and integrate.

## How to do it...

1. Import the other modules required:

```
import argparse
import contextlib
import logging
from pathlib import Path
import sys
```

2. Import the modules with the constituent applications. This is generally done after all standard library modules:

```
import markov_gen
import markov_summ
```

3. Create a new function to combine the existing functions from the other applications. We're including the iteration in this function to meet the expectation of generating 1,000 sample files. It looks like this:

```
def gen_and_summ(iterations: int, samples: int) -> None:
    for i in range(iterations):
        markov_gen.main(
            [
                "--samples", str(samples),
                "--randomize", str(i + 1),
                "--output", f"data/ch14/markov_{i}.csv",
            ]
        )
    markov_summ.main()
```

4. The overall problem statement has two parameters, with fixed values: the users would like 1,000 iterations of 1,000 samples. This provides a large collection of large files to work with. We can define command-line arguments with these values as defaults:

```
def get_options(argv: list[str]) -> argparse.Namespace:
    parser = argparse.ArgumentParser(description="Markov Chain
    Generator and Summary")
    parser.add_argument("-s", "--samples", type=int, default=1_000)
    parser.add_argument("-i", "--iterations", type=int, default=10)
    return parser.parse_args(argv)
```

For more on how to use the `argparse` module, see the recipes in *Chapter 6*.

5. The final report is sent to standard output, `sys.stdout`, by the `print()` function in the `markov_summ` application. This isn't ideal, so we'll use a `contextlib` context manager to redirect the output to a file:

```
def main(argv: list[str] = sys.argv[1:]) -> None:
    options = get_options(argv)
    target = Path.cwd() / "summary.md"
    with target.open("w") as target_file:
        with contextlib.redirect_stdout(target_file):
            gen_and_summ(options.iterations, options.samples)
```

6. The combined functionality is now a new module with a function, `main()`, that we can invoke from a block of code like the following:

```
if __name__ == "__main__":
    logging.basicConfig(stream=sys.stderr, level=logging.INFO)
    main()
```

This gives us a combined application written entirely in Python. We can write unit tests for this composite, as well as for each of the two steps that make up the overall application.

## How it works...

The central feature of this design is importing useful functionality from existing, working, and tested modules. This avoids the problems with copy-and-paste programming. Copying code from one file and pasting it into another means that any change made to one is unlikely to be made to any of the copies. As the various copies of a function slowly diverge,

problems fixed in one place surface in another. This phenomenon is sometimes called *code rot*.

The copy-and-paste approach is made more complicated when a class or function does several things. Too many features reduces the potential for reuse. We summarize this as the Inverse Power Law of Reuse – the reusability of a class or function,  $R(c)$ , is related to the inverse of the number of features in that class or function,  $F(c)$ :

$$R(c) \propto \frac{1}{F(c)}$$

The idea of counting features depends, of course, on the level of abstraction. It can help to consider the processing that maps inputs to outputs. Too many input-process-output mappings will limit reuse.

The **SOLID** design principles provide guidance for keeping components small and narrowly focused. These principles apply to applications as well as components. In particular, the **Single Responsibility Principle** suggests that an application should do one thing. It's better to have many small applications – like bricks – that can easily be combined than to have one large application that's an imponderable big ball of mud.

## There's more...

We'll look at two additional areas of rework of the application:

- **Structure:** Using the top-level `main()` function treats each component as an opaque container. When trying to create a composite application, we may need to refactor the component modules to look for better organization of the features.
- **Logging:** When multiple applications are combined, the combined logging can become complicated. To improve observability, we may need to refactor the logging.

We'll go through these in turn.

## Structure

In some cases, it becomes necessary to rearrange software to expose useful features. For example, the `main()` function inside the `markov_gen.py` module relies on a `write_samples()` function. This function creates a single file with the required number of samples, which are `(outcome, chain)` two-tuples.

The input to the summary processing is a sequence of these `(outcome, chain)` two-tuples. The composite application doesn't really need to process 1,000 separate files. It needs to process 1,000 collections of 1,000 two-tuples.

Doing the refactoring to expose this detail will make this feature available for the composite application. This can make the composite easier to understand and maintain.

## Logging

In the *Using logging for control and audit output* recipe in *Chapter 13*, we looked at how to use the logging module for control, audit, and error outputs. When we build a composite application, we'll have to combine the logging features from each of the original applications.

The logging configuration for a composite application needs to be examined carefully. If we don't ensure that the logging configuration is done only once in the top-level application, then combining applications can lead to multiple, conflicting logging configurations. There are two approaches that a composite application can follow:

- The composite application manages the logging configuration. This may mean overwriting all previously defined loggers. This is the default behavior and can be stated explicitly via `incremental = false` in a TOML configuration document.
- The composite application can preserve other application loggers and merely modify the configuration. This is not the default behavior and requires including `incremental = true` in the TOML configuration document.

The use of incremental configuration can be helpful when combining Python applications that don't properly isolate the logging configuration into the `__name__ == "__main__"`



block of code. It's often easier to refactor logging configuration to put it into the top-level block of code; this permits the composite application to more simply configure logging for all of the components.

## See also

- In the *Designing scripts for composition* recipe in *Chapter 13*, we looked at the core design pattern for a composable application.
- Books and articles on **Clean Architecture** and **Hexagonal Architecture** can be very helpful. Titles on design patterns are also helpful, such as *Mastering Python Design Patterns – Third Edition*.

## Combining many applications using the Command design pattern

Many complex suites of applications follow a design pattern similar to the one used by the Git program. There's a base command, `git`, with a number of subcommands. These include `git pull`, `git commit`, and `git push`.

What's central to this design is the idea of a collection of individual commands under a common parent command. Each of the various features of Git can be thought of as a separate subclass definition that performs a given function.

## Getting ready

We'll build a composite application from two commands. This is based on the *Combining two applications into one* recipe from earlier in this chapter.

These features are based on modules with names such as `markov_gen`, `markov_summ`, and `markov_analysis`. The idea is that we can restructure separate modules into a single class hierarchy following the **Command** design pattern.

There are two key ingredients to this design pattern:

1. A client class depends only on the methods of the abstract superclass, `Command`.

2. Each individual subclass of the `Command` superclass has an identical interface. We can substitute any one of them for any other.

An overall application script can then create and execute any one of the `Command` subclasses.

Note that **any** feature that can be wrapped into a class is a candidate for this design. Consequently, some rework to create a single **Facade** class is sometimes required for sprawling, poorly designed applications.

## How to do it...

We'll start by creating a superclass for all of the related commands. We'll then extend that superclass for the subcommands that are part of the overall application.

1. Here's the `Command` superclass:

```
import argparse

class Command:
    def __init__(self) -> None:
        pass

    def execute(self, options: argparse.Namespace) -> None:
        pass
```

It helps to rely on `argparse.Namespace` to provide a very flexible collection of options and arguments to each subclass.

We'll use this also in the *Managing arguments and configuration in composite applications* recipe in this chapter.

2. Create a subclass of the `Command` superclass for the `Generate` class. This will wrap the processing and output from the example module in the `execute()` method of this class:

```
from pathlib import Path
from typing import Any
import markov_gen
```

```

class Generate(Command):
    def __init__(self) -> None:
        super().__init__()
        self.seed: Any | None = None
        self.output: Path

    def execute(self, options: argparse.Namespace) -> None:
        self.output = Path(options.output)
        with self.output.open("w") as target:
            markov_gen.write_samples(target, options)
        print(f"Created {str(self.output)}")

```

3. Create a subclass of the Command superclass for the Summarize class. For this class, we've wrapped the file creation and file processing into the execute() method of the class:

```

import contextlib
import markov_summ_2

class Summarize(Command):
    def execute(self, options: argparse.Namespace) -> None:
        self.summary_path = Path(options.summary_file)
        with self.summary_path.open("w") as result_file:
            output_paths = [Path(f) for f in options.output_files]
            outcomes, lengths =
                markov_summ_2.process_files(output_paths)
            with contextlib.redirect_stdout(result_file):
                markov_summ_2.write_report(outcomes, lengths)

```

4. The overall composite processing can be performed by the following main() function:

```

def main() -> None:
    options_1 = argparse.Namespace(samples=1000, output="data/x.csv")
    command1 = Generate()
    command1.execute(options_1)

    options_2 = argparse.Namespace(
        summary_file="data/report.md", output_files=["data/x.csv"]
    )
    command2 = Summarize()

```

```
command2.execute(options_2)
```

We've created two commands: one is an instance of the `Generate` class, and the other is an instance of the `Summarize` class. These commands can be executed to provide a combined feature that both generates and summarizes data.

## How it works...

Creating interchangeable, polymorphic classes for the various subcommands is a handy way to provide an extensible design. The **Command** design pattern strongly encourages each individual subclass to have an identical signature. Doing this makes it easier for the command subclasses to be created and executed. Also, new commands can be added that fit this pattern.

One of the SOLID design principles is the **Liskov Substitution Principle (LSP)**. It suggests any of the subclasses of the `Command` abstract class can be used in place of the parent class.

Each `Command` instance has a consistent interface. The use of the **Command** design pattern makes it easy to be sure that `Command` subclasses can be interchanged with each other. The overall `main()` script can create instances of the `Generate` or `Summarize` classes. The substitution principle means that either instance can be executed because the interfaces are the same. This flexibility makes it easy to parse the command-line options and create an instance of either of the available classes. We can extend this idea and create sequences of individual command instances.

## There's more...

One of the more common extensions to this design pattern is to provide for composite commands. In the *Combining two applications into one* recipe, we showed one way to create composites. This is another way, based on defining a new `Command` class that implements a combination of existing `Command` class instances:

```
class CmdSequence(Command):
    def __init__(self, *commands: type[Command]) -> None:
        super().__init__()
        self.commands = [command() for command in commands]

    def execute(self, options: argparse.Namespace) -> None:
        for command in self.commands:
            command.execute(options)
```

This class will accept other Command classes via the `*commands` parameter. From the classes, it will build the individual class instances.

We might use this `CmdSequence` class like this:

```
>>> from argparse import Namespace
>>> options = Namespace(
...     samples=1_000,
...     randomize=42,
...     output="data/x.csv",
...     summary_file="data/y.md",
...     output_files=["data/x.csv"]
... )

>>> both_command = CmdSequence(Generate, Summarize)
>>> both_command.execute(options)
Created data/x.csv
```

This design exposes some implementation details. In particular, the two class names and the intermediate `x.csv` file are details that seem superfluous.

We can create a slightly nicer subclass of the `CmdSequence` argument if we focus specifically on the two commands being combined. This will have an `__init__()` method that follows the pattern of other `Command` subclasses:

```
class GenSumm(CmdSequence):
    def __init__(self) -> None:
        super().__init__(Generate, Summarize)
```

```
def execute(self, options: argparse.Namespace) -> None:
    self.intermediate = Path("data") / "ch14_r02_temporary.toml"
    new_namespace = argparse.Namespace(
        output=str(self.intermediate),
        output_files=[str(self.intermediate)],
        **vars(options),
    )
    super().execute(new_namespace)
```

This class definition incorporates two other classes into the already defined `CmdSequence` class structure. The `super().__init__()` expression invokes the parent class initialization with the `Generate` and `Summarize` classes as argument values.

This provides a composite application definition that conceals the details of how a file is used to pass data from the first step to a subsequent step. This is purely a feature of the composite integration and doesn't lead to any changes in either of the original applications that form the composite.

## See also

- In the *Designing scripts for composition* and *Using logging for control and audit output* recipes in *Chapter 13*, we looked at the constituent parts of this composite application.
- In the *Combining two applications into one* recipe earlier in this chapter, we looked at the constituent parts of this composite application. In most cases, we'll need to combine elements of all of these recipes to create a useful application.
- We'll often need to follow the *Managing arguments and configuration in composite applications* recipe, which comes next in this chapter.
- For other advanced design patterns, see *Mastering Python Design Patterns – Third Edition*.

## Managing arguments and configuration in composite applications

When we have a complex suite (or system) of individual applications, they may share common features. The coordination of common features among many applications can become awkward. As a concrete example, imagine defining the various one-letter abbreviated options for command-line arguments. We might want all of our applications to use the `-v` option for verbose output. Ensuring that there are no conflicts among all the applications might require keeping some kind of master list of all options.

This kind of common configuration should be kept in only one place. Ideally, it would be in a common module, used throughout a family of applications.

Additionally, we often want to divorce the modules that perform useful work from the CLI. This lets us refactor the internal software design without changing the user's understanding of how to use the application.

In this recipe, we'll look at ways to ensure that a suite of applications can be refactored without creating unexpected changes to the CLI.

### Getting ready

We'll imagine an application suite built from three commands. This is based on the applications shown in the *Combining two applications into one* recipe earlier in this chapter. We'll have a `markov` application with three subcommands: `markov generate`, `markov summarize`, and the combined application, `markov gensumm`.

We'll rely on the subcommand design from the *Combining many applications using the Command design pattern* recipe earlier in this chapter. This will provide a handy hierarchy of `Command` subclasses:

- The `Command` class is an abstract superclass.
- The `Generate` subclass performs the chain-generating functions from *Chapter 13* recipe the *Designing scripts for composition* recipe.

- The Summarize subclass performs summarizing functions from *Chapter 13* recipe the *Using logging for control and audit output* recipe.
- A GenSumm subclass can perform combined chain generation and summarization, following the ideas of the *Combining many applications using the **Command** design pattern* recipe.

In order to create a simple command-line application, we'll need appropriate argument parsing. For more on argument parsing, see *Chapter 6*.

This argument parsing will rely on the subcommand-parsing capability of the built-in `argparse` module. We can create a common set of command options that apply to all subcommands. We can also create unique options for each of the distinct subcommands.

## How to do it...

This recipe will start with a consideration of what the CLI commands need to look like. A first release often involves some prototypes or examples to be sure that the commands are truly useful to the user. After learning the user's preferences, we can change how we implement the argument definitions in each of the **Command** subclasses.

1. Define the CLI. This is an exercise in **User Experience (UX)** design. While a great deal of UX design is focused on web and mobile device applications, the core principles are appropriate for CLI applications as well. Earlier, we noted that the root application will be called `markov`. It will have the following three subcommands:

```
markov generate -o detail_file.csv -s samples
markov summarize -o summary_file.md detail_file.csv ...

markov gensumm -g samples
```

The `gensumm` command combines the `generate` and `summarize` commands into a single operation that does both.

2. Define the root Python application. We'll call it `markov.py`. It's common to have a package `__main__.py` file that contains the application. It's often simpler to use an



OS alias to provide the UX name.

3. We'll import the class definitions from the *Combining many applications using the Command design pattern* recipe. This will include the Command superclass and the Generate, Summarize, and GenSumm subclasses. We'll extend the Command class with an additional method, `arguments()`, to set the unique options in the argument parser for this command. This is a class method and is called on the class as a whole, not an instance of the class:

```
class Command:
    @classmethod
    def arguments(
        cls,
        sub_parser: argparse.ArgumentParser
    ) -> None:
        pass

    def __init__(self) -> None:
        pass

    def execute(self, options: argparse.Namespace) -> None:
        pass
```

4. Here are the unique options for the generate subcommand. We won't repeat the entire class definition, only the new `arguments()` method. This creates arguments that are unique to the markov generate subcommand:

```
class Generate(Command):
    @classmethod
    def arguments(
        cls,
        generate_parser: argparse.ArgumentParser
    ) -> None:
        default_seed = os.environ.get("RANDOMSEED", "0")
        generate_parser.add_argument(
            "-s", "--samples", type=int, default=1_000)
        generate_parser.add_argument(
            "-o", "--output", dest="output")
        generate_parser.add_argument(
```

```

        "-r", "--randomize", default=default_seed)
generate_parser.set_defaults(command=cls)

```

5. Here is the new arguments() method of the Summarize subcommand:

```

class Summarize(Command):
    @classmethod
    def arguments(
        cls,
        summarize_parser: argparse.ArgumentParser
    ) -> None:
        summarize_parser.add_argument(
            "-o", "--output", dest="summary_file")
        summarize_parser.add_argument(
            "output_files", nargs="*", type=Path)
        summarize_parser.set_defaults(command=cls)

```

6. Here is the new arguments() method for the composite command, GenSumm:

```

class GenSumm(Command):
    @classmethod
    def arguments(
        cls,
        gensumm_parser: argparse.ArgumentParser
    ) -> None:
        default_seed = os.environ.get("RANDOMSEED", "0")
        gensumm_parser.add_argument(
            "-s", "--samples", type=int, default=1_000)
        gensumm_parser.add_argument(
            "-o", "--output", dest="summary_file.md")
        gensumm_parser.add_argument(
            "-r", "--randomize", default=default_seed)
        gensumm_parser.set_defaults(command=cls)

```

7. Create the overall argument parser. Use this to create a subparser builder. For each subcommand, create a subparser and add arguments that are unique to that command:

```

import argparse

def get_options(
    argv: list[str]
) -> argparse.Namespace:
    parser = argparse.ArgumentParser(prog="Markov")
    subparsers = parser.add_subparsers()
    generate_parser = subparsers.add_parser("generate")
    Generate.arguments(generate_parser)

    summarize_parser = subparsers.add_parser("summarize")
    Summarize.arguments(summarize_parser)
    gensumm_parser = subparsers.add_parser("gensumm")
    GenSumm.arguments(gensumm_parser)

```

8. Parse the command-line values. In most cases, the argument definitions include validation rules. In this case, there's an additional validation check to make sure a command was provided. Here are the final parsing and validating steps:

```

options = parser.parse_args(argv)
if "command" not in options:
    parser.error("No command selected")
return options

```

The overall parser includes three subcommand parsers. One will handle the markov generate command, another handles markov summarize, and the third handles a combined markov gensumm. Each subcommand has slightly different combinations of options.

The command option is set via the `set_defaults()` method. This also provides useful additional information about the command to be executed. In this case, we've provided the class that must be instantiated.

The overall application is defined by the following `main()` function:

```

from typing import cast

def main(argv: list[str] = sys.argv[1:]) -> None:

```

```
options = get_options(argv)
command = cast(type[Command], options.command())
command.execute(options)
```

The resulting object will have an `execute()` method that does the real work of this command.

## How it works...

There are two parts to this recipe:

- Using the **Command** design pattern to define a related set of classes that are polymorphic. For more information on this, see the *Combining many applications using the Command design pattern* recipe.
- Using features of the `argparse` module to handle subcommands.

The `argparse` module feature that's important here is the `add_subparsers()` method of a parser. This method returns an object to build each distinct subcommand parser. We assigned this object to the `subparsers` variable.

We also used the `set_defaults()` method of a parser to add a command argument to each of the subparsers. This argument will be populated by the defaults defined for one of the subparsers. The value assigned by the `set_defaults()` method actually used will show which of the subcommands was invoked.

Consider the following OS command:

```
(cookbook3) % markov generate -s 100 -o x.csv
```

This command will be parsed to create a `Namespace` object that looks like this:

```
Namespace(command=<class '__main__.Generate'>, output='x.csv', samples=100)
```

The `command` attribute in the `Namespace` object is the default value provided as part of the

subcommand definition. The values for `output` and `samples` come from the `-o` and `-g` options.

## There's more...

The `get_options()` function has an explicit list of classes that it incorporates into the overall command. As shown, a number of lines of code are repeated, and this could be optimized. We can provide a data structure that replaces a number of lines of code:

```
def get_options_2(argv: list[str] = sys.argv[1:]) -> argparse.Namespace:
    parser = argparse.ArgumentParser(prog="markov")
    subparsers = parser.add_subparsers()
    sub_commands = [
        ("generate", Generate),
        ("summarize", Summarize),
        ("gensumm", GenSumm),
    ]

    for name, subc in sub_commands:
        cmd_parser = subparsers.add_parser(name)
        subc.arguments(cmd_parser)
    # The parsing and validating remains the same...
```

This variation on the `get_options()` function uses a sequence of two-tuples to provide the command name and the relevant class to implement the command. Iterating through this list ensures that all of the various subclasses of the `Command` class are processed in a perfectly uniform manner.

## See also

- See the *Designing scripts for composition* and *Using logging for control and audit output* recipes in *Chapter 13* for the basics of building applications focused on being composable.
- See the *Combining two applications into one* recipe from earlier in this chapter for the background on the components used in this recipe.
- See the *Using argparse to get command-line input* recipe in *Chapter 6* for more on the

background of argument parsing.

- Other tools for creating CLIs include click, hydra, and invoke.

## Wrapping and combining CLI applications

One common kind of automation involves running several programs, some of which are not Python applications. This commonly arises when integrating multiple tools, which are often applications used to build applications or documents. Since the programs aren't written in Python, it's impossible to refactor each program to create a composite Python application. When using a non-Python application, we can't follow the *Combining two applications into one* recipe shown earlier in this chapter.

Instead of aggregating the Python components, an alternative is to wrap the other programs in Python, creating a composite application. The use case is very similar to the use case for writing a shell script. The difference is that Python is used instead of a shell language. Using Python has some advantages:

- Python has a rich collection of data structures. Most shell languages are limited to strings and arrays of strings.
- Python has several outstanding unit test frameworks. Rigorous unit testing gives us confidence that the combined application will work as expected.

In this recipe, we'll look at how we can run other applications from within Python.

### Getting ready

In the *Designing scripts for composition* recipe in *Chapter 13*, we identified an application that did some processing that led to the creation of a rather complex result. For the purposes of this recipe, we'll assume that the application is not written in Python.

We'd like to run this program several thousand times, but we don't want to copy and paste the necessary commands into a script. Also, because the shell is difficult to test and has so few data structures, we'd like to avoid using the shell.

For this recipe, we'll work with an application as if it were a native binary application,

written in Rust, Go, or Pascal. There are two ways to explore this:

- The `markov_gen.pas` file in this book's Git repository can be used to build a working, native binary application. The Free Pascal Compiler project (<https://www.free-pascal.org>) has compilers for a large number of platforms.
- Another common situation is the need to execute a Jupyter notebook using the `jupyter execute` command. We can't directly import a notebook, but must execute it via a separate command.

Another alternative that can help with exploring these design alternatives is to make a Python application behave like a binary executable by adding a *shebang* line as the first line in the file. In many cases, the following can be used as the first line of a Python script:

```
#!/usr/bin/env python
```

For macOS and Linux, use the following to change the mode of the file to executable:

```
% chmod +x your_application_file.py
```

Working with a native binary application means that we can't import a Python module that comprises the application. Instead, the application is run as a separate OS process. This limits interaction to command-line argument values and OS environment variables.

To run a native binary application, we use the `subprocess` module. There are two common design patterns for running another program from within Python:

- The other program doesn't produce any output, or we don't want to gather the output in our Python program. The first situation is typical of OS utilities that return a status code when they succeed or fail. The second situation is typical of programs that update files and produce logs.
- The other program produces the output; the Python wrapper needs to capture and process it. This may happen when the Python wrapper needs to take extra actions to clean up or retry in the event of failure.

In this recipe, we'll look at the first case: the output isn't something we need to capture. In the *Wrapping a program and checking the output* recipe, we'll look at the second case, where the output will be scrutinized by the Python wrapper program.

In many cases, one benefit of wrapping an existing application with Python is the ability to dramatically rethink the UX. This lets us redesign the CLI to better fit the user's needs.

Let's look at wrapping a program that's normally started with the `src/ch14/markov_gen` command. Here's an example:

```
(cookbook3) % src/ch14/markov_gen -o data/ch14_r04.csv -s 100 -r 42
# file = "data/ch14_r04.csv"
# samples = 100
# randomize = 42
```

The output filename needs to be flexible so that we can run the program hundreds of times. This is often done by interpolating a sequence number into the filename. For example, `f"data/ch14/samples_{n}.csv"` would be used in Python to create unique filenames.

## How to do it...

In this recipe, we'll start by creating a small demonstration application. This is a **spike solution** (<https://wiki.c2.com/?SpikeSolution>). This will be used to be sure we understand how the other application works. Once we have the correct OS command, we can wrap this in a function call to make it easier to use:

1. Import the `argparse` and `subprocess` modules and the `Path` class. We'll also need the `sys` module:

```
import argparse
import subprocess
from pathlib import Path
import sys
```

2. Write the core processing using the `subprocess` module to invoke the target application. This can be tested separately to ensure it can execute the application. In



this case, `subprocess.run()` will execute the given command, and the `check=True` option will raise an exception if the status is non-zero. Here's the spike solution that demonstrates the essential processing:

```

directory, n = Path("/tmp"), 42
filename = directory / f"sample_{n}.csv"
command = [
    "markov_gen",
    "--samples", "10",
    "--output", str(filename),
]
subprocess.run(command, check=True)

```

This minimal spike can be run to make sure things work before proceeding to refactor the spike into something more useful.

3. Wrap the spike solution in a function that reflects the desired behavior. The processing looks like this:

```

def make_files(directory: Path, files: int = 100) -> None:
    for n in range(files):
        filename = directory / f"sample_{n}.csv"
        command = [
            "markov_gen",
            "--samples", "10",
            "--output", str(filename),
        ]
        subprocess.run(command, check=True)

```

4. Write a function to parse the command-line options. In this case, there are two positional parameters: a directory and a number of chain samples to generate. The function looks like this:

```

def get_options(argv: list[str]) -> argparse.Namespace:
    parser = argparse.ArgumentParser()
    parser.add_argument("directory", type=Path)
    parser.add_argument("samples", type=int)
    options = parser.parse_args(argv)
    return options

```

5. Write a main function to do the parsing and processing:

```
def main(argv: list[str] = sys.argv[1:]) -> None:
    options = get_options(argv)
    make_files(options.directory, options.samples)
```

We now have a function that's testable using any of the Python unit-testing frameworks. This can give us real confidence that we have a reliable application built around an existing non-Python application.

## How it works...

The subprocess module is how Python runs other programs. The `run()` function does a number of things for us.

In a POSIX (such as Linux or macOS) context, the steps are similar to the following sequence:

1. Prepare the `stdin`, `stdout`, and `stderr` file descriptors for the child process.
2. Invoke a function like the `os.execve()` function to start the child process.
3. Wait for the child process to finish and collect the final status.

An OS shell, such as `bash`, conceals these details from application developers and users. The `subprocess.run()` function, similarly, hides the details of creating and waiting for a child process.

Using the subprocess module to run a separate executable allows Python to integrate a wide variety of software components into a unified whole. Using Python offers a much richer collection of data structures than the shell, proper exception handling instead of checking the final status code, and a way to unit test.

## There's more...

We'll add a simple clean-up feature to this script. The idea is that all of the output files should be created as an atomic operation.

In order to clean up, we'll need to wrap the core processing in a `try:` block. We'll write

a second function, `make_files_clean()`, that uses the original `make_files()` function to include a clean-up feature. A new overall function, `make_files_clean()`, would look like this:

```
def make_files_clean(directory: Path, files: int = 100) -> None:
    """Create sample data files, with cleanup after a failure."""
    try:
        make_files(directory, files)
    except subprocess.CalledProcessError as ex:
        # Remove any files.
        for partial in directory.glob("sample_*.csv"):
            partial.unlink()
        raise
```

The exception-handling block does two things. First, it removes any incomplete files from the current working directory. Second, it re-raises the original exception so that the failure will propagate to the client application.

Any test cases for this application would need to make use of mock objects. See the *Mocking external resources* recipe in *Chapter 15*.

## See also

- This kind of automation is often combined with other Python processing. See the *Designing scripts for composition* recipe in *Chapter 13*.
- The goal is often to create a composite application; see the *Managing arguments and configuration in composite applications* recipe earlier in this chapter.
- For a variation on this recipe, see the *Wrapping a program and checking the output* recipe, which is up next in this chapter.

## Wrapping a program and checking the output

One common kind of automation involves wrapping a program. The advantage of a Python wrapper is the ability to perform detailed aggregation and analysis of the output files.

A Python program might transform, filter, or summarize the output from a subprocess.

In this recipe, we'll see how to run other applications from within Python, and collect and process the output.

## Getting ready

In the *Designing scripts for composition* recipe in *Chapter 13*, we identified an application that did some processing, leading to the creation of a rather complex result. We'd like to run this program several hundred times, but we don't want to copy and paste the necessary commands into a script. Also, because the shell is difficult to test and has so few data structures, we'd like to avoid using the shell.

For this recipe, we'll work with a native binary application written in some compiled language like Ada, Fortran, or Pascal. This means that we can't simply import the Python module that comprises the application. Instead, we'll have to execute this application by running a separate OS process with the `subprocess` module. There are two common use cases for running another binary program from within Python:

- Either there isn't any output, or we don't want to process the output file in our Python program.
- We need to capture and possibly analyze the output to retrieve information or ascertain the level of success. We might need to transform, filter, or summarize the log output.

In this recipe, we'll look at the second case: the output must be captured and summarized. In the *Wrapping and combining CLI applications* recipe in this chapter, we looked at the first case, where the output is simply ignored.

Here's an example of running the `markov_gen` application:

```
(cookbook3) % RANDOMSEED=42 src/ch14/markov_gen --samples 5 --output t.csv
# file = "t.csv"
# samples = 5
# randomize = 42
```

There were three lines of output written to the OS standard output file, all starting with

#. These show the file being created, the number of samples, and the random number generator seed being used. This output is confirmation the data was correctly created.

We want to capture the details of these lines from this application and summarize them. The total of all the generated samples should match the number of samples summarized, confirming that all of the data was processed.

## How to do it...

We'll start by creating a **spike solution** (<https://wiki.c2.com/?SpikeSolution>) to confirm the command and arguments needed to run another application. We'll transform the spike solution into a function that captures output for further analysis.

1. Import the `argparse` and `subprocess` modules and the `Path` class. We'll also need the `sys` module and the `Any` type hint:

```
import argparse
from collections.abc import Iterable, Iterator
from pathlib import Path
import subprocess
import sys
from typing import Any
```

2. Write the core processing, using the `subprocess` module to invoke the target application. Here's a spike solution that demonstrates the essential processing:

```
directory, n = Path("/tmp"), 42
filename = directory / f"sample_{n}.toml"
temp_path = directory / "stdout.txt"
command = [
    "src/ch14/markov_gen",
    "--samples", "10",
    "--output", str(filename),
]
with temp_path.open("w") as temp_file:
    process = subprocess.run(
        command,
        stdout=temp_file, check=True, text=True
    )
```

```
output_text = temp_path.read_text()
```

This spike does two things: it builds a complicated command line for a subprocess and it also collects the output from the subprocess. A temporary file allows the subprocess module to run a process that creates a very large file.

The idea is to create a script with this minimal spike and be sure things work before proceeding to refactor the spike into something more useful.

3. Refactor the spike to create a function that runs a command and collects the output.

Here's a `command_output()` function:

```
def command_output(
    temporary: Path, command: list[str]
) -> str:
    temp_path = temporary / "stdout"
    with temp_path.open("w") as temp_file:
        subprocess.run(
            command,
            stdout=temp_file, check=True, text=True
        )
    output_text = temp_path.read_text()
    temp_path.unlink()
    return output_text
```

4. Refactor the rest of the spike into a function to generate the commands. It makes sense for this function to be a generator so it can create a collection of similar commands.

```
def command_iter(options: argparse.Namespace) -> Iterable[list[str]]:
    for n in range(options.iterations):
        filename = options.directory / f"sample_{n}.csv"
        command = [
            "src/ch14/markov_gen",
            "--samples", str(options.samples),
            "--output", str(filename),
            "--randomize", str(n+1),
        ]
```

**yield** command

5. Define a function to parse the expected output from a command. We'll decompose the parsing into a sequence of generators that create regular expression Match objects, extract the matched groups, and build a final dictionary reflecting the content. The function can look like this:

```
def parse_output(result: str) -> dict[str, Any]:
    matches = (
        re.match(r"^#\s*([\s=]+\s*=\s*(.*?)\s*$", line)
        for line in result.splitlines()
    )
    match_groups = (
        match.groups()
        for match in matches
        if match
    )
    summary = {
        name: value
        for name, value in match_groups
    }
    return summary
```

6. Here's the high-level function to extract useful information from the command output. The generator function looks like this:

```
import tempfile

def summary_iter(options: argparse.Namespace) -> Iterator[dict[str, Any]]:
    commands = command_iter(options)
    with tempfile.TemporaryDirectory() as tempdir:
        results = (
            command_output(Path(tempdir), cmd)
            for cmd in commands
        )
        for text in results:
            yield parse_output(text)
```

This function will use a stack of generator expressions. For more background, see the *Using stacked generator expressions* recipe in *Chapter 9*. Since these are all generator expressions, each individual result is processed separately. This can allow large files to be digested as small summaries one at a time.

7. Write a function to parse the command-line options. In this case, the target directory is a positional parameter, and the number of samples in each file and the number of files to generate are options. The function looks like this:

```
def get_options(argv: list[str]) -> argparse.Namespace:
    parser = argparse.ArgumentParser()
    parser.add_argument("directory", type=Path)
    parser.add_argument("-s", "--samples", type=int, default=1_000)
    parser.add_argument("-i", "--iterations", type=int, default=10)
    options = parser.parse_args(argv)
    return options
```

8. Combine the parsing and execution into a main function:

```
def main(argv: list[str] = sys.argv[1:]) -> None:
    options = get_options(argv)
    parsed_results = list(summary_iter(options))
    print(f"Built {len(parsed_results)} files")
    # print(parsed_results)
    total = sum(
        int(rslt['samples']) for rslt in parsed_results
    )
    print(f"Total {total} samples")
```

Now we can run this new application and have it execute the underlying application and also gather the output, producing a helpful summary. We've built this using Python instead of a **bash** (or other shell) script. We can make use of Python's data structures and unit testing.

## How it works...

The subprocess module is how Python programs run other programs available on a given computer. For more background, see the *Wrapping and combining CLI applications* recipe



in this chapter.

The subprocess module gives us access to one of the most important parts of the operating system: launching a subprocess and collecting the output. The underlying OS can direct output to the console, or files, or through “pipes” to another process. The default behavior when launching a subprocess is to inherit the parent process definitions for the three standard files, stdin, stdout, and stderr. In this recipe, we’ve replaced the default stdout assignment with a file that permits us to gather (and analyze) output that would have gone to the console.

## There’s more...

Once we’ve wrapped the markov\_gen binary application within a Python application, we have a number of alternatives available to us for improving the output.

Because we’ve wrapped the underlying application, we don’t need to change this code to change the results it produces. We can modify our wrapper program, leaving the original data generator intact.

We can refactor the main() function to replace the print() functions with processing to create a more useful format. A possible rewrite would emit a CSV file with the detailed generator information:

```
import csv

def main_2(argv: list[str] = sys.argv[1:]) -> None:
    options = get_options(argv)

    total_counter = 0
    wtr = csv.DictWriter(sys.stdout, ["file", "samples", "randomize"])
    wtr.writeheader()
    for summary in summary_iter(options):
        wtr.writerow(summary)
        total_counter += int(summary["samples"])
    wtr.writerow({"file": "TOTAL", "samples": total_counter})
```

The list of files and numbers of samples can be used to partition the data for model training

purposes.

We are able to build useful features separately by creating layers of features. Leaving the underlying application untouched can help us to perform regression tests to be sure the core statistical validity has not been harmed by adding new features.

## See also

- See the *Wrapping and combining CLI applications* recipe from earlier in this chapter for another approach to this recipe.
- This kind of automation is often combined with other Python processing. See the *Designing scripts for composition* recipe in *Chapter 13*.
- The goal is often to create a composite application; see the *Managing arguments and configuration in composite applications* recipe from earlier in this chapter.
- Many practical applications will work with more complex output formats. For information on processing complex line formats, see the *String parsing with regular expressions* recipe in *Chapter 1* and the *Reading complex formats using regular expressions* recipe in *Chapter 11*. Much of *Chapter 11*, relates to the details of parsing input files.
- For more information on interprocess communication, see *The Linux Documentation Project: Interprocess Communication Mechanisms*.

## Join our community Discord space

Join our Python Discord workspace to discuss and find out more about the book: <https://packt.link/dHrHU>.



# 15

## Testing

Testing is central to creating working software. Here's the canonical statement describing the importance of testing:

*"Any program feature without an automated test simply doesn't exist". (Kent Beck, *Extreme Programming Explained: Embrace Change*)*

We can distinguish several kinds of testing:

**Unit testing:** This applies to independent *units* of software: functions, classes, or modules. The unit is tested in isolation to confirm that it works correctly.

**Integration testing** : This combines units to be sure they integrate properly.

**System testing** : This tests an entire application or a system of interrelated applications to be sure that the suite of software components works properly. This is also called **end-to-end testing** or **functional testing**. This is often used for **acceptance testing** to confirm that software is fit for use.

**Performance testing** : This ensures that a unit, subsystem, or whole system meets

performance objectives (also often known as **load testing**). In some cases, performance testing includes the study of resources such as memory, threads, or file descriptors. The goal is to be sure that software makes appropriate use of system resources. This is sometimes called **benchmarking**, when the goal is to measure resource usage instead of ensuring that the usage is below some threshold.

These are some of the more common types. In this chapter, we'll focus on unit testing since it is foundational to creating trust that software works reliably. Other forms of testing rest on the foundation of reasonably complete unit tests.

It's sometimes helpful to summarize a test scenario following the Gherkin language. In this test specification language, each scenario is described by *GIVEN-WHEN-THEN* steps. Here's an example:

```
Scenario: Binomial coefficient typical case.  
  
Given n = 52  
And k = 5  
When The binomial coefficient is computed with c = binom(n, k)  
Then the result, c, is 2,598,960
```

This approach to writing tests describes the given starting state or arrangement, an action to perform, and one or more assertions about the resulting state after the action. This is sometimes called the “arrange-act-assert” pattern.

In this chapter, we'll look at the following recipes:

- *Using docstrings for testing*
- *Testing functions that raise exceptions*
- *Handling common doctest issues*
- *Unit testing with the unittest module*
- *Combining unittest and doctest tests*
- *Unit testing with the pytest module*

- *Combining pytest and doctest tests*
- *Testing things that involve dates or times*
- *Testing things that involve randomness*
- *Mocking external resources*

We'll start by including tests within the docstring of a module, class, or function. This makes the test case act as both documentation of the design intent and a verifiable confirmation that it really does work as advertised.

## Using docstrings for testing

Good Python includes docstrings inside every module, class, function, and method. Many tools can create useful, informative documentation from docstrings. Refer back to the *Writing clear documentation strings with RST markup* recipe in *Chapter 3* for an example of how to create docstrings.

One important element of a docstring is a concrete example. The examples provided in docstrings can become unit-test cases that are exercised by Python's **doctest** tool.

In this recipe, we'll look at ways to turn examples into proper automated test cases.

### Getting ready

We'll look at a small function definition as well as a class definition. Each of these will contain docstrings that include examples that can be used as automated tests.

We'll use a function to compute the binomial coefficient of two numbers. It shows the number of combinations of  $n$  things taken in groups of size  $k$ . For example, how many ways a 52-card deck can be dealt into 5-card hands is computed like this:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

This can be implemented by a Python function that looks like this:

```
from math import factorial

def binom_draft(n: int, k: int) -> int:
    return factorial(n) // (factorial(k) * factorial(n - k))
```

Functions, generally, have no internal state, making a function like this relatively easy to test. This will be one of the examples used for showing the unit testing tools available.

We'll also look at a class that uses lazy calculation of the mean and median of a collection of numbers. Objects often have internal state, defined by the various `self.` attributes. State changes are often difficult. This is similar to the classes shown in *Chapter 7*. The *Designing classes with lots of processing* and *Using properties for lazy attributes* recipes both have classes similar to this.

Here is an outline of the `Summary` class, with some implementation details omitted:

```
from collections import Counter

class Summary:
    def __init__(self) -> None:
        self.counts: Counter[int] = collections.Counter()

    def __str__(self) -> str:
        ...

    def add(self, value: int) -> None:
        self.counts[value] += 1

    @property
    def mean(self) -> float:
        ...

    @property
    def median(self) -> float:
        ...
```

The `add()` method changes the internal state of a `Summary` object. Because of this state change to the `self.counts` attribute, we'll need to provide more sophisticated examples to

show how an instance of the `Summary` class behaves.

## How to do it...

We'll show two variations in this recipe. The first variation can be applied to functions like the `binom()` function where there is no object with a mutable state. The second is more appropriate for stateful operations, such as the `Summary` class. We'll look at them together because they're very similar, even though they apply to different kinds of applications.

### Writing examples for functions

This recipe starts by creating the function's docstring, and then adds an example of how the function works:

1. Start the docstring with a summary:

```
def binom(n: int, k: int) -> int:
    """
    Computes the binomial coefficient.
    This shows how many combinations exist of
    *n* things taken in groups of size *k*.
```

2. Include the parameter definitions and the return value definition:

```
:param n: size of the universe
:param k: size of each subset
:returns: the number of combinations
```

3. Mock up an example of using the function at Python's `>>>` prompt:

```
>>> binom(52, 5)
2598960
```

4. Close the docstring with the appropriate quotation marks:

```
"""
```



## Writing examples for stateful objects

This recipe also starts with writing a docstring. The docstring will show several steps using the stateful object to show the object's internal state changes:

1. Write a class-level docstring with a summary. It can help to leave some blank lines in front of the doctest example:

```
class Summary:
    """
    Computes summary statistics.
```

2. Extend the class-level docstring with a concrete example of how the class works. In this case, we'll show how the `add()` method sets the state of the object. We'll also show how to interrogate the state of the object:

```
>>> s = Summary()
>>> s.add(8)
>>> s.add(9)
>>> s.add(9)
>>> round(s.mean, 2)
8.67
>>> s.median
9
>>> print(str(s))
mean = 8.67
median = 9
```

3. Finish with the triple quotes to end the docstring for this class:

```
"""
```

Because this example uses floating-point values, we've rounded the result of the mean in the docstring example. Floating-point values might not have the exact same text representation on all platforms and an exact test for equality may fail unexpectedly.

## Running the tests

When we run the **doctest** program, we'll generally get a silent response because the test passed.

The interaction looks like this:

```
(cookbook3) % python -m doctest recipe_01.py
```

When the tests pass, there is no output. We can add a `-v` command-line option to see an enumeration of the tests run. This can be helpful to confirm that all of the tests in the module were found.

What happens when something doesn't work? We'll modify a test case to have a wrong answer and force a failure. When we run the **doctest** program – using a broken test case – we'll see output like this:

```
(cookbook3) % python -m doctest recipe_01.py
*****
File "/Users/slott/Documents/Writing/Python/Python Cookbook
3e/src/ch15/recipe_01.py", line 29, in recipe_01.Summary
Failed example:
    s.median
Expected:
    10
Got:
    9
*****
1 items had failures:
  1 of  7 in recipe_01.Summary
***Test Failed*** 1 failures.
```

This shows where the error is. It shows an expected value from the test example, and the actual answer that failed to match the expected answer. Ordinarily – without the `-v` option – silence means all tests were passed successfully.

## How it works...

The `doctest` module includes a main program – as well as several functions – that scan a Python file for examples. The scanning operation looks for blocks of text that have a characteristic pattern of the Python REPL: a `>>>` prompt with code, followed by lines that show the response from the code, followed by a blank line to end the example output. Clearly, these must be formatted to precisely match the Python REPL output to be found.

The `doctest` parser creates a small test case object from the prompt line and the block of response text. There are three common cases:

- **No expected response text:** We saw this pattern when we defined the tests for the `add()` method of the `Summary` class.
- **A single line of response text:** This was exemplified by the `binom()` function and the `mean()` method of the `Summary` class.
- **Multiple lines of response:** Responses are bounded by either the next `>>>` prompt or a blank line. This was exemplified by the `str()` example of the `Summary` class.

Unless special annotations are used, the output text must precisely match the expectation text. In general, every space counts.

This testing protocol imposes some software design constraints. Functions and classes must be designed to work from the `>>>` prompt. Because it can become awkward to create very complex objects as part of a docstring example, class designs must be kept simple enough to be demonstrated at the interactive prompt. These constraints often have the benefit of keeping a design understandable.

The simplicity of the final comparison with the expected result can create some complications. In the example, we rounded the value of the mean to two decimal places. This is because the display of floating-point values may vary slightly from platform to platform.

## There's more...

One of the important considerations in test design is identifying edge cases. An **edge case** generally focuses on the limits for which a calculation is designed.

There are, for example, two edge cases for the binomial function:

$$\binom{n}{0} = \binom{n}{n} = 1$$

We can add these to the examples to be sure that our implementation is sound. This leads to a docstring that looks like the following:

```
"""
    Computes the binomial coefficient.
    This shows how many combinations exist of
    *n* things taken in groups of size *k*.

    :param n: size of the universe
    :param k: size of each subset
    :returns: the number of combinations

    >>> binom(52, 5)
    2598960
    >>> binom(52, 0)
    1
    >>> binom(52, 52)
    1

    """
```



To keep the examples straight in the source code files, we've changed the name of this function to `binom2`. This hack lets us keep both examples in a single Python module.

In some cases, we might need to test values that are outside the valid range of values. These cases raise exceptions, which means they aren't really ideal for putting into the docstring. The examples can clutter the explanation with details of things that should never normally

happen. Fortunately, we have a place to put additional examples.

In addition to reading docstrings, the tool also looks for test cases in a global variable named `__test__`. This variable must refer to a mapping. The keys to the mapping will be test case names, and the values of the mapping must be doctest examples. Generally, each value will need to be a triple-quoted string.

Because the examples in the `__test__` variable are not inside the docstrings, they don't show up when using the built-in `help()` function. Nor do they show up when using other tools to create documentation from source code. This might be a place to put examples of failures or complex exceptions.

We might add something like this:

```
__test__ = {
    "GIVEN_binom_WHEN_0_0_THEN_1": """
        >>> binom(0, 0)
        1
        """,
    "GIVEN_binom_WHEN_52_52_THEN_1": """
        >>> binom(52, 52)
        1
        """,
}
```

We can use this for tests that don't need to be as visible as the docstring examples.

## See also

- In the *Testing functions that raise exceptions* and *Handling common doctest issues* recipes later in this chapter, we'll look at two additional doctest techniques.
- For more background to the concept of stateless functions, see *Chapter 3* and *Chapter 9*.

## Testing functions that raise exceptions

Python permits docstrings inside packages, modules, classes, functions, and methods. A good docstring should contain an example of how the feature is used. The example may need to include common exceptions, too. There's one complicating factor, however, to including exceptions.

When an exception is raised, the traceback messages created by Python are not completely predictable. The message may include object ID values that are impossible to predict or module line numbers that may vary slightly depending on the context in which the test is executed. The general matching rules for `doctest` compare expected and actual results precisely. In this recipe, we'll look at additional techniques to add flexibility.

### Getting ready

We'll look at a small function definition as well as a class definition. Each of these will contain docstrings that include examples that can be used as formal tests.

We'll use the function from the *Using docstrings for testing* recipe, shown earlier in this chapter, that computes the binomial coefficient of two numbers. It shows the number of combinations of  $n$  things taken in groups of  $k$ . For example, it shows how many ways a 52-card deck can be dealt into 5-card hands.

This function does a simple calculation and returns a value; it lacks any internal state, making each request independent. We'd like to include some additional test cases in the `__test__` variable to show what this does when given values outside the expected ranges.

### How to do it...

We start by running the `binom` function we defined previously. This output provides a handy template to show the expected output:

1. Run the function manually at the interactive Python prompt to collect the actual exception details. Copy and paste these results.
2. Create a global `__test__` variable at the end of the module. One approach is to build

the mapping from all global variables with names that start with `test_`:

```

    recipe_02
2 items passed all tests:
  2 tests in recipe_02.__test__.test_GIVEN_n_5_k_52_THEN_ValueError
  3 tests in recipe_02.binom
5 tests in 3 items.

```

3. Define each test case as a global variable with a block of text containing the doctest example. This can include additional notes about the scenario. These variables must be set before the final creation of the `__test__` mapping.
4. Paste in the interactive session session output.

It will start like this:

```

test_GIVEN_n_5_k_52_THEN_ValueError_1 = """
GIVEN n=5, k=52 WHEN binom(n, k) THEN exception
>>> binom(52, -5)
Traceback (most recent call last):
  File
"/Users/slott/miniconda3/envs/cookbook3/lib/python3.12/doctest.py",
line 1357, in __run
    exec(compile(example.source, filename, "single",
File "<doctest
recipe_02.__test__.test_GIVEN_n_5_k_52_THEN_ValueError[0]>",
line 1, in <module>
    binom(52, -5)
File "/Users/slott/Documents/Writing/Python/Python Cookbook
3e/src/ch15/recipe_02.py", line 29, in binom
    return factorial(n) // (factorial(k) * factorial(n - k))
           ^^^^^^^^^^^^^^^^^
ValueError: factorial() not defined for negative values
"""

```

5. Replace the traceback details with `. . .`. Leave the initial line and the final exception in place. Add a directive to doctest, by putting `# doctest: +ELLIPSIS` after the line to be executed. It will look like this:

```
test_GIVEN_n_5_k_52_THEN_ValueError_2 = """
    GIVEN n=5, k=52 WHEN binom(n, k) THEN exception
    >>> binom(5, 52)
    Traceback (most recent call last):
    ...
    ValueError: factorial() not defined for negative values
    """
```

We can now use a command like this to test the entire module's features:

```
(cookbook3) % python -m doctest recipe_02.py
```

Because each test is a separate global variable, we can easily add test scenarios. All of the names starting with `test_` will become part of the final `__test__` mapping that's used by the `doctest` tool.

## How it works...

Because eliding traceback details is so common, the **doctest** tool recognizes the ellipsis (`...`) in the context of a traceback. The ellipsis is also available in other contexts as one of many directives to modify the testing behavior. The directives are included as special comments with the line of code that performs the test action. They can also be provided as general instructions on the command line.

We have two additional ways to handle tests that include an exception:

- We can use a `# doctest: +IGNORE_EXCEPTION_DETAIL` directive on the line of code that will raise the exception. This lets us provide a full traceback error message. The details of the traceback are ignored, and only the final exception line is matched against the expected value. This makes it possible to copy an actual error and paste it into the documentation.
- We can use a `# doctest: +ELLIPSIS` directive and replace parts of the traceback message with `...`. This directive is redundant for traceback messages.

The use of an explicit directive can help to make it clear what the intent is.



## There's more...

There are two more directives that are often useful:

- `+NORMALIZE_WHITESPACE`: Using this directive allows some flexibility in the whitespace for the expected value.
- `+SKIP`: The test is skipped.

There are a few more directives, but they're rarely needed.

## See also

- See the *Using docstrings for testing* recipe earlier in this chapter. This recipe shows the basics of doctest.
- See the *Handling common doctest issues* recipe next in this chapter. This shows other special cases that require doctest directives.

## Handling common doctest issues

A docstring that contains an example is part of good Python programming. The way the doctest tool uses literal matching of the expected text output against the actual text can make testing complicated for Python objects that do not have a consistent text representation.

For example, object hash values are randomized. This often results in the order of elements in a set collection being unpredictable. We have several choices for creating test case example output:

- Write examples that can tolerate randomization. One technique is by sorting the elements of a set into a defined order.
- Stipulate a specific value for the `PYTHONHASHSEED` environment variable.

There are several other considerations beyond simple variability in the location of keys or items in a set. Here are some other concerns:

- The `id()` and `repr()` functions may expose an internal object ID. No guarantees can be made about these values.
- Floating-point values may vary across platforms.
- The current date, time, and local timezone cannot meaningfully be used in a test case.
- Random numbers using the default seed are difficult to predict.
- OS resources may not exist, or may not be in the proper state.

It's important to note that `doctest` examples require an *exact* match with the text. This means our test cases must avoid unpredictable results stemming from hash randomization or floating-point implementation details.

## Getting ready

We'll look at three separate versions of this recipe. The first will include a function where the output includes the contents of a set. Because the order of items in a set can vary, this isn't as easy to test as we'd like. Here's the function definition:

```
from string import ascii_letters

def unique_letters(text: str) -> set[str]:
    letters = set(text.lower())
    non_letters = letters - set(ascii_letters)
    return letters - non_letters
```

Testing the `unique_letters()` function is difficult because the order of items within a set is unpredictable.

The second example will be a class that doesn't define a unique `__repr__()` definition. The default definition of the `__repr__()` method will expose an internal object ID. Since these vary, the test results will vary. Here's the class definition:

```

class Point:
    def __init__(self, lat: float, lon: float) -> None:
        self.lat = lat
        self.lon = lon

    @property
    def text(self) -> str:
        ns_hemisphere = "S" if self.lat < 0 else "N"
        ew_hemisphere = "W" if self.lon < 0 else "E"
        lat_deg, lat_ms = divmod(abs(self.lat), 1.0)
        lon_deg, lon_ms = divmod(abs(self.lon), 1.0)
        return (
            f"{lat_deg:02.0f}°{lat_ms*60:4.3f}'{ns_hemisphere}"
            f" {lon_deg:03.0f}°{lon_ms*60:4.3f}'{ew_hemisphere}"
        )

```

For the third example, we'll look at a real-valued function so that we can work with floating-point values:

$$\phi(n) = \frac{1}{2} \left[ 1 + \operatorname{erf} \frac{n}{\sqrt{2}} \right]$$

This function is the cumulative probability density function for standard z-scores. See the *Creating a partial function* recipe in *Chapter 9*, for more information on the idea of normalized scores.

Here's the Python implementation:

```

from math import sqrt, pi, exp, erf

def phi(n: float) -> float:
    return (1 + erf(n / sqrt(2))) / 2

def frequency(n: float) -> float:
    return phi(n) - phi(-n)

```

The `phi()` and `frequency()` functions involve some rather complicated numeric processing. The unit tests have to reflect the floating-point precision issues.

## How to do it...

We'll look at set ordering and object representation in three mini-recipes. We'll start with set ordering, then look at object IDs, and finally, floating-point values.

### Writing doctest examples with unpredictable set ordering

1. Write a draft of the test that seems to capture the essence:

```
>>> phrase = "The quick brown fox..."
>>> unique_letters(phrase)
{'b', 'c', 'e', 'f', 'h', 'i', 'k', 'n', 'o', 'q', 'r', 't', 'u', 'w',
 'x'}
```

This test will work when the hash values for these strings happen to fall into this specific order.

2. One possible fix is to sort the results to impose an order.

Another alternative is to compare the output to a set object. The two choices look like this:

```
>>> phrase = "The quick brown fox..."
>>> sorted(unique_letters(phrase))
['b', 'c', 'e', 'f', 'h', 'i', 'k', 'n', 'o', 'q', 'r', 't', 'u', 'w',
 'x']
>>> (unique_letters(phrase) ==
...   {'b', 'c', 'e', 'f', 'h', 'i', 'k', 'n', 'o', 'q', 'r', 't',
...   'u', 'w', 'x'})
... )
True
```

A third choice is to set the PYTHONHASHSEED environment variable to force known orderings. We'll look at this alternative below.

### Writing doctest examples with object IDs

Ideally, our applications won't display object IDs. These are essentially impossible to predict. Here's what we can do:

1. Define a *happy path* doctest scenario to show that the class performs its essential methods correctly. In this case, we'll create a `Point` instance and use the `text` property to see a representation of the point:

```
>>> Point(36.8439, -76.2936).text
'36°50.634'N 076°17.616'W'
```

2. When we define a test that displays the object's representation string, the test will include results that include the unpredictable object ID. The doctest might look like the following:

```
>>> Point(36.8439, -76.2936)
<recipe_03.Point object at 0x107910610>
```

We need to change the test by using a `# doctest: +ELLIPSIS` directive. This means changing the `>>> Point(36.8439, -76.2936)` line in the test, and using an ellipsis on the exception displayed in the expected output to look like this:

```
>>> Point(36.8439, -76.2936) # doctest: +ELLIPSIS
<recipe_03.Point object at ...>
```

This kind of test suggest a design improvement. It's often best to define `__repr__()`. Another choice is to avoid tests where `__repr__()` may be used.

## Writing doctest examples for floating-point values

We have two choices when working with float values. We can round the values to a certain number of decimal places. An alternative is to use the `math.isclose()` function. We'll show both:

1. Import the necessary libraries and define the `phi()` and `frequency()` functions as shown previously.
2. For each example, include an explicit use of `round()`:

```
>>> round(phi(0), 3)
0.5
>>> round(phi(-1), 3)
0.159
>>> round(phi(+1), 3)
0.841
```

3. An alternative is to use the `isclose()` function from the `math` module:

```
>>> from math import isclose
>>> isclose(phi(0), 0.5)
True
>>> isclose(phi(1), 0.8413, rel_tol=.0001)
True
>>> isclose(phi(2), 0.9772, rel_tol=1e-4)
True
```

Because float values can't be compared exactly, it's best to display values that have been rounded to an appropriate number of decimal places. It's sometimes nicer for readers of examples to use `round()` because it may be slightly easier to visualize how the function works, compared to the `isclose()` alternative.

## How it works...

Because of hash randomization, the hash keys used for sets are unpredictable. This is an important security feature, used to defeat a subtle denial-of-service attack. For details, see url: <http://www.ocert.org/advisories/ocert-2011-003.html>.

Since Python 3.7, dictionary keys are guaranteed to be kept in insertion order. This means that an algorithm that builds a dictionary will provide a consistent sequence of key values. The same ordering guarantee is not made for sets. Interestingly, sets of integers tend to have a consistent ordering because of the way hash values are computed for numbers. Sets of other types of objects, however, will not show consistent ordering of items.

When confronted with unpredictable results like set ordering or internal object identification revealed by the `__repr__()` method, we have a testability issue. We can either change

the software to be more testable, or we can change the test to tolerate some unpredictability.

Most floating-point implementations are reasonably consistent. However, there are few formal guarantees about the last few bits of any given floating-point number. Rather than trusting that all of the bits have exactly the right value, it's often a good practice to round the value to a precision consistent with other values in the problem domain.

Being tolerant of unpredictability can be taken too far, allowing the test to tolerate bugs. For more in-depth testing of mathematical functions, the hypothesis package provides ways to define a domain of robust test cases.

## There's more...

We can run the tests with the PYTHONHASHSEED environment variable set. In Linux (and macOS X) we can do this in a single command-line statement:

```
(cookbook3) % PYTHONHASHSEED=42 python3 -m doctest recipe_03.py
```

This will provide a fixed, reproducible hash randomization while running doctest. We can also use PYTHONHASHSEED=0 to disable hash randomization.

The **tox** tool has a `--hashseed=x` option to allow setting a consistent hash seed to an integer value prior to running tests.

## See also

- The *Testing things that involve dates or times* recipe, in particular, the `now()` method of `datetime` requires some care.
- The *Testing things that involve randomness* recipe shows how to test processing that involves using the `random` module.
- We'll look at how to work with external resources in the *Mocking external resources* recipe later in this chapter.

## Unit testing with the unittest module

The `unittest` module allows us to step beyond the examples used by `doctest`. Each test case can have one more scenario built as a subclass of the `TestCase` class. These use result checks that are more sophisticated than the literal text matching used by the `doctest` tool.

The `unittest` module also allows us to package tests outside docstrings. This can be helpful for tests for edge cases that might be too detailed to be helpful documentation. Often, `doctest` cases focus on the **happy path** – the most common use cases, where everything works as expected. We can use the `unittest` module to more easily define test cases that diverge from the happy path.

This recipe will show how we can use the `unittest` module to create more sophisticated tests.

### Getting ready

It's sometimes helpful to summarize a test following ideas behind the Gherkin language. In this test specification language, each scenario is described by *GIVEN-WHEN-THEN* steps. For this case, we have a scenario like this:

```
Scenario: Summary object can add values and compute statistics.
```

```
Given a Summary object
```

```
And numbers in the range 0 to 1000 (inclusive) shuffled randomly
```

```
When all numbers are added to the Summary object
```

```
Then the mean is 500
```

```
And the median is 500
```

The `TestCase` class doesn't precisely follow this three-part given-when-then (or arrange-act-assert) structure. A `TestCase` class generally has two parts:

- A `setUp()` method must implement the *Given* steps of the test case.
- A `runTest()` method must handle the *Then* steps to confirm the results using a number of assertion methods to confirm the actual results match the expected results.



The *When* steps can be in either method. The choice of where to implement the *When* steps is often tied to the question of reuse. For example, a class or function may have a number of methods to take different actions or make a number of state changes. In this case, it makes sense to pair each *When* step with distinct *Then* steps to confirm correct operation. The `runTest()` method can implement both *When* and *Then* steps. A number of subclasses can share the common `setUp()` method.

As another example, a class hierarchy may offer a number of alternative implementations for the same algorithm. In this case, the *Then* step confirmation of correct behavior is in the `runTest()` method. Each alternative implementation has a distinct subclass with a unique `setUp()` method for the *Given* and *When* steps.

An optional `tearDown()` method is available for those tests that need to perform some cleanup of left-over resources. This is outside the test's essential scenario specification.

We'll create some tests for a class that is designed to compute some basic descriptive statistics. The `unittest` test cases let us define sample data that's larger than anything we'd ever choose to enter as `doctest` examples. We can easily use thousands of data points rather than two or three as part of evaluating performance.

The bulk of the code that we're going to test was shown in the *Using docstrings for testing* recipe earlier in this chapter.

Because we're not looking at the implementation details, we can think of this as opaque-box testing; the implementation details are not known to the tester.

We'd like to be sure that when we use thousands of samples, the class performs correctly. We'd also like to ensure that it works quickly; we'll use this as part of an overall performance test, as well as a unit test.

## How to do it...

We'll need to create a separate module and a subclass of `TestCase` in that module. Tools like **pytest** can discover test modules if their names begin with `test_`, giving us a naming convention for these additional modules. Here's how we can create tests separate from

the module's code:

1. Create a file with a name related to the module under test. If the module was named `summary.py`, then a good name for a test module would be `test_summary.py`. Using the `test_` prefix makes it easier for tools like **pytest** to find the test.
2. We'll use the `unittest` module for creating test classes. We'll also be using the `random` module to scramble the input data. We'll also import the module under test:

```
import unittest
import random

from recipe_01 import Summary
```

3. Create a subclass of `TestCase`. Provide this class with a name that shows the intent of the test. We've chosen a name with a summary of the three steps:

```
class
    GIVEN_Summary_WHEN_1k_samples_THEN_mean_median(unittest.TestCase):
```

4. Define a `setUp()` method in this class that handles the *Given* step of the test. We've created a collection of 1,001 samples shuffled into a random order:

```
def setUp(self) -> None:
    self.summary = Summary()
    self.data = list(range(1001))
    random.shuffle(self.data)
```

5. Define a `runTest()` method that handles the *When* step of the test:

```
def runTest(self) -> None:
    for sample in self.data:
        self.summary.add(sample)
```

6. Add assertions to the `runTest()` method to implement the *Then* steps of the test:

```
self.assertEqual(500, self.summary.mean)
self.assertEqual(500, self.summary.median)
```

If our test module is called `recipe_04.py`, we can use the following command to find `TestCase` classes in the `recipe_04` module and run them:

```
(cookbook3) % python -m unittest recipe_04.py
```

If all of the assertions pass, then the test suite will pass and the test run will be successful overall.

## How it works...

The `TestCase` class is used to define one test case. The class can have a `setUp()` method to create the unit and possibly the request. The class must have at least a `runTest()` method to make a request of the unit and check the response.

A single test often isn't sufficient. If we created three separate test classes in the `recipe_04.py` module, then we would see output that looks like this:

```
(cookbook3) % python -m unittest recipe_04.py
```

```
...
```

```
-----  
Ran 3 tests in 0.003s
```

```
OK
```

As each test is passed, a `.` is displayed. This shows that the test suite is making progress. The summary shows the number of tests run and the time. If there are failures or exceptions, the counts shown at the end will reflect these.

Finally, there's a summary line. In this case, it consists of `OK`, showing that all the tests passed.

If we include a test that fails, we'll see the following output when we use the `-v` option to

get verbose output:

```
(cookbook3) % python -m unittest -v recipe_04.py
runTest (recipe_04.GIVEN_Summary_WHEN_1k_samples_THEN_mean_median.runTest) ... ok
test_mean (recipe_04.GIVEN_Summary_WHEN_1k_samples_THEN_mean_median_2.test_mean) ... FAIL
test_median (recipe_04.GIVEN_Summary_WHEN_1k_samples_THEN_mean_median_2.test_median) ... ok
test_mode (recipe_04.GIVEN_Summary_WHEN_1k_samples_THEN_mode.test_mode) ... ok

=====
FAIL: test_mean (recipe_04.GIVEN_Summary_WHEN_1k_samples_THEN_mean_median_2.test_mean)
-----
Traceback (most recent call last):
  File "/Users/slott/Documents/Writing/Python/Python Cookbook 3e/src/ch15/recipe_04.py",
    line 122, in test_mean
    self.assertEqual(501, self.summary.mean)
AssertionError: 501 != 500.0

-----
Ran 4 tests in 0.004s

FAILED (failures=1)
```

There's a final summary of FAILED. This includes (`failures=1`) to show how many tests failed.

## There's more...

In these examples, we have two assertions for the two *Then* steps inside the `runTest()` method. If one fails, the test stops as a failure, and the other step is not exercised.

This is a weakness in the design of this test. If the first assertion fails, we may not get all of the diagnostic information we might want. We should avoid having a sequence of otherwise independent assertions in the `runTest()` method.

When we want more diagnostic details, we have two general choices:

- Use multiple test methods instead of a single `runTest()`. We can create multiple methods with names that start with `test_`. The default implementation of the test

loader will execute the `setUp()` method prior to each separate `test_` method when there is no overall `runTest()` method. This is often the simplest way to group a number of related tests together.

- Use multiple subclasses of the `TestCase` subclass, each with a separate *Then* step implementation. When the `setUp()` is inherited, this will be shared by each subclass.

Following the first alternative, the test class would look like this:

```
class GIVEN_Summary_WHEN_1k_samples_THEN_mean_median_2(unittest.TestCase):
    def setUp(self) -> None:
        self.summary = Summary()
        self.data = list(range(1001))
        random.shuffle(self.data)

        for sample in self.data:
            self.summary.add(sample)

    def test_mean(self) -> None:
        self.assertEqual(500, self.summary.mean)

    def test_median(self) -> None:
        self.assertEqual(500, self.summary.median)
```

We've refactored the `setUp()` method to include the *Given* and *When* steps of the test. The two independent *Then* steps are refactored into their own separate `test_mean()` and `test_median()` methods. These two methods are used instead of the `runTest()` method.

Since each test is run separately, we'll see separate error reports for problems with computing the mean or with computing the median.

The `TestCase` class defines numerous assertions that can be used as part of the *Then* steps. We encourage careful study of the `unittest` section of the *Python Standard Library* documentation to see all of the variations available.

In all but the smallest projects, it's common practice to sequester the test files into a separate directory, often called `tests`. When this is done, we can rely on the discovery application that's part of the `unittest` framework.

The `unittest` loader can search each module in a given directory for all classes that are derived from the `TestCase` class. This collection of classes within the larger collection of modules becomes the complete `TestSuite`.

We can do this with the `discover` command of the `unittest` package:

```
(cookbook3) % (cd src; python -m unittest discover -s ch15)
.....
-----
Ran 15 tests in 0.008s

OK
```

This will locate all test cases in all test modules in the `tests` directory of a project.

## See also

- We'll combine `unittest` and `doctest` in the *Combining `pytest` and `doctest` tests* recipe next in this chapter. We'll look at mocking external objects in the *Mocking external resources* recipe later in this chapter.
- The *Unit testing with the `pytest` module* recipe later in this chapter covers the same test case from the perspective of the `pytest` tool.

## Combining `unittest` and `doctest` tests

In some cases, we'll want to combine tests written for the `unittest` and `doctest` tools. For examples of using the `doctest` tool, see the *Using `docstrings` for testing* recipe earlier in this chapter. For examples of using the `unittest` tool, see the *Unit testing with the `unittest` module* recipe earlier in this chapter.

The `doctest` examples are an essential element of the documentation strings on modules, classes, methods, and functions. The `unittest` cases will often be in a separate `tests` directory in files with names that match the pattern `test_*.py`. An important part of creating trustworthy software is running as wide a variety of tests as possible.

In this recipe, we'll look at ways to combine a variety of tests into one tidy package.

## Getting ready

We'll refer back to the example from the *Using docstrings for testing* recipe, shown earlier in this chapter. This recipe created tests for a class, `Summary`, that does some statistical calculations. In that recipe, we included examples in the docstrings.

In the *Unit testing with the unittest module* recipe earlier in this chapter, we wrote some `TestCase` classes to provide additional tests for this class.

As context, we'll assume there's a project folder structure that looks like the following directory tree:

```
project-name/  
  src/  
    summary.py  
  tests/  
    test_summary.py  
  README  
  pyproject.toml  
  requirements.txt  
  tox.ini
```

This means tests are in both the `src/summary.py` module and in the `tests/test_summary.py` file.

We need to combine all of the tests into a single, comprehensive test suite.

The recipe examples use `recipe_01.py` instead of some cooler name such as `summary.py`. Ideally, a module should have a memorable, meaningful name. The book content is quite large, and the names are designed to match the overall chapter and recipe outline.

## How to do it...

To combine `unittest` and `doctest` test cases, we'll start with an existing test module, and add a `load_tests()` function to merge the relevant `doctest` with the existing `unittest` test cases. A function named `load_tests()` must be provided. This name is required so the `unittest` loader will use it:

1. To use doctest tests, import the doctest module. To write TestCase classes, import the unittest module. We'll also need the random module so we can control the random seeds in use:

```
import unittest
import doctest
import random
```

2. Import the modules containing doctest examples:

```
import recipe_01
```

3. To implement the load\_tests protocol, define a load\_tests() function in the test module. We'll combine the standard tests automatically discovered by unittest with the additional tests found by the doctest module:

```
def load_tests(
    loader: unittest.TestLoader, standard_tests: unittest.TestSuite,
    pattern: str
) -> unittest.TestSuite:
    dt = doctest.DocTestSuite(recipe_01)
    standard_tests.addTests(dt)
    return standard_tests
```

The loader argument value to the load\_tests() function is the test case loader currently being used; this is generally ignored. The standard\_tests argument value will be all of the tests loaded by default. Generally, this is the suite of all subclasses of TestCase. The function updates this object with the additional tests. The pattern value is the value provided to the loader to locate tests; this is also ignored.

When we run this from the OS command prompt, we see the following:

```
(cookbook3) % python -m unittest -v recipe_05.py
test_mean
(recipe_05.GIVEN_Summary_WHEN_1k_samples_THEN_mean_median.test_mean) ... ok
```



```
test_median
(recipe_05.GIVEN_Summary_WHEN_1k_samples_THEN_mean_median.test_median) ...
ok
Summary (recipe_01)
Doctest: recipe_01.Summary ... ok
Twc (recipe_01)
Doctest: recipe_01.Twc ... ok
GIVEN_binom_WHEN_0_0_THEN_1 (recipe_01.__test__)
Doctest: recipe_01.__test__.GIVEN_binom_WHEN_0_0_THEN_1 ... ok
GIVEN_binom_WHEN_52_52_THEN_1 (recipe_01.__test__)
Doctest: recipe_01.__test__.GIVEN_binom_WHEN_52_52_THEN_1 ... ok
binom (recipe_01)
Doctest: recipe_01.binom ... ok
binom2 (recipe_01)
Doctest: recipe_01.binom2 ... ok

-----
Ran 8 tests in 0.006s

OK
```

This shows us that the unittest test cases were included as well as doctest test cases.

## How it works...

The `unittest.main()` application uses a test loader to find all of the relevant test cases. The loader is designed to find all classes that extend `TestCase`. It will also look for a `load_tests()` function. This function can provide a suite of additional tests. It can also do non-default searches for tests when that's needed.

Generally, we can import a module with `docstrings` and use a `DocTestSuite` to build a test suite from the imported module. We can, of course, import other modules or even scan the `README` file for more examples to test. The goal is to make sure every example in both the code and the documentation actually works.

## There's more...

In some cases, a module may be quite complicated; this can lead to multiple test modules. The test modules may have names such as `tests/test_module_feature_X.py` to show

that there are tests for separate features of a very complex module. The volume of code for test cases can be quite large, and keeping the features separate can be helpful.

In other cases, we might have a test module that has tests for several different but closely related small modules. A single test module may employ inheritance techniques to cover all the modules in a package.

When combining many smaller modules, there may be multiple suites built in the `load_tests()` function. The body might look like this:

```
import doctest
import unittest

import recipe_01 as ch15_r01
import recipe_02 as ch15_r02
import recipe_03 as ch15_r03
import recipe_05 as ch15_r04

def load_tests(
    loader: unittest.TestLoader, standard_tests: unittest.TestSuite,
    pattern: str
) -> unittest.TestSuite:
    for module in (ch15_r01, ch15_r02, ch15_r03, ch15_r04):
        dt = doctest.DocTestSuite(module)
        standard_tests.addTests(dt)
    return standard_tests
```

This will incorporate doctest examples from multiple modules into a single, comprehensive test suite.

## See also

- For examples of doctest, see the *Using docstrings for testing* recipe earlier in the chapter. For examples of unittest, see the *Unit testing with the unittest module* recipe earlier in this chapter.

## Unit testing with the `pytest` module

The `pytest` tool allows us to step beyond the examples used by `doctest` in docstrings. Instead of using a subclass of `TestCase`, the `pytest` tool lets us use function definitions. The `pytest` approach uses Python's built-in `assert` statement, leaving the test case looking somewhat simpler.

The `pytest` tool is not part of Python; it needs to be installed separately. Use a command like this:

```
(cookbook3) % python -m pip install pytest
```

In this recipe, we'll look at how we can use `pytest` to simplify our test cases.

### Getting ready

The Gherkin language can help to structure a test. For this recipe, we have a scenario like this:

```
Scenario: Summary object can add values and compute statistics.
```

```
Given a Summary object
```

```
And numbers in the range 0 to 1000 (inclusive) shuffled randomly
```

```
When all numbers are added to the Summary object
```

```
Then the mean is 500
```

```
And the median is 500
```

A `pytest` test function doesn't precisely follow the Gherkin three-part structure. A test function generally has two parts:

- If necessary, fixtures are defined to establish the *Given* steps. Fixtures are designed for reuse as well as composition. A fixture can also tear down resources after a test has finished.
- The body of the function will usually handle the *When* steps to exercise the object being tested and the *Then* steps to confirm the results.

These boundaries are not fixed. A fixture might, for example, create an object and also take action, executing both the *Given* and *When* steps. This permits multiple test functions to apply several independent *Then* steps.

We'll create some tests for a class that is designed to compute some basic descriptive statistics. The bulk of the code was shown in the *Using docstrings for testing* recipe.

This is an outline of the class, provided as a reminder of what the method names are:

```
class Summary:
    def __init__(self) -> None: ...
    def __str__(self) -> str: ...
    def add(self, value: int) -> None: ...
    @property
    def mean(self) -> float: ...
    @property
    def median(self) -> float: ...
    @property
    def count(self) -> int: ...
    @property
    def mode(self) -> list[tuple[int, int]]: ...
```

We want to duplicate testing shown in the *Unit testing with the unittest module* recipe. We'll use the `pytest` features to do this.

## How to do it...

It's often best to start with a separate test file, perhaps even a separate `tests` directory:

1. Create a test file with a name similar to the module under test. If the module file was named `summary.py`, then a good name for a test module would be `test_summary.py`. Using the `test_` prefix makes the test easier to find.
2. We'll use the `pytest` module for creating test classes. We'll also be using the `random` module to scramble the input data. Also, we need to import the module under test:

```
import random
import pytest
from recipe_01 import Summary
```

3. Implement the *Given* step as a fixture. This is marked with the `@pytest.fixture` decorator. It creates a function that can return a useful object, data for creating an object, or a mocked object:

```
@pytest.fixture()
def flat_data() -> list[int]:
    data = list(range(1001))
    random.shuffle(data)
    return data
```

4. Implement the *When* and *Then* steps as a test function with a name visible to pytest. This means the function name must begin with `test_`:

```
def test_flat(flat_data: list[int]) -> None:
```

When a parameter name in a test function definition is the name of a fixture function, the fixture function is evaluated automatically. The results of the fixture function are provided at runtime. This means the shuffled set of 1,000 values will be provided as an argument value for the `flat_data` parameter.

5. Implement a *When* step to perform an operation on an object:

```
summary = Summary()
for sample in flat_data:
    summary.add(sample)
```

6. Implement the *Then* steps to validate the outcome:

```
assert summary.mean == 500
assert summary.median == 500
```

If our test module is called `test_summary.py`, we can often execute the tests found in it

with a command like the following:

```
(cookbook3) % python -m pytest test_summary.py
```

This will invoke the main application that's part of the `pytest` package. It will search the given file for functions with names starting with `test_` and execute those test functions.

## How it works...

We're using several parts of the `pytest` package:

- The `@fixture` decorator can be used to create reusable test fixtures with objects in known states, ready for further processing.
- The `pytest` application to do several things:
  - Discover tests. By default, it searches a directory named `tests` for module names starting with `test_`. Within those, it looks for functions with names starting with `test_`. It also finds `unittest.TestCase` classes.
  - Run all of the tests, evaluating the fixtures as needed.
  - Display a summary of the results.

When we run the `pytest` command, we'll see output that looks like this:

```
(cookbook3) % python -m pytest recipe_06.py
===== test session starts =====
platform darwin -- Python 3.12.0, pytest-7.4.3, pluggy-1.3.0
rootdir: /Users/slott/Documents/Writing/Python/Python Cookbook 3e
configfile: pytest.ini
plugins: anyio-4.0.0
collected 3 items

recipe_06.py ... [100%]

===== 3 passed in 0.02s =====
```

As each test is passed, a `.` is displayed. This shows that the test suite is making progress.

The summary shows the number of tests run and the time. If there are failures or exceptions, the counts on the last line will reflect this.

If we change one test slightly to be sure that it fails, we'll see the following output:

```
(cookbook3) % python -m pytest recipe_06.py
===== test session starts =====
platform darwin -- Python 3.12.0, pytest-7.4.3, pluggy-1.3.0
rootdir: /Users/slott/Documents/Writing/Python/Python Cookbook 3e
configfile: pytest.ini
plugins: anyio-4.0.0
collected 3 items

recipe_06.py F..                                     [100%]

===== FAILURES =====
_____ test_flat _____

flat_data = [883, 104, 898, 113, 519, 94, ...]

    def test_flat(flat_data: list[int]) -> None:
        summary = Summary()
        for sample in flat_data:
            summary.add(sample)
>       assert summary.mean == 501
E       assert 500.0 == 501
E         + where 500.0 = <recipe_01.Summary object at 0x10fdbc350>.mean

recipe_06.py:57: AssertionError
===== short test summary info =====
FAILED recipe_06.py::test_flat - assert 500.0 == 501
===== 1 failed, 2 passed in 0.17s =====
```

This shows a summary of passing and failing tests and the details of each failure.

## There's more...

In this example, we have two *Then* steps inside the `test_flat()` function. These are implemented as two `assert` statements. If the first one fails, the test stops as a failure, and the following step will be skipped. This means we might not see all the diagnostic information we might need.

A better design is to use multiple test functions. All of the functions can share a common fixture. In this case, we can create a second fixture that depends on the `flat_data` fixture and builds a `Summary` object to be used by a number of tests:

```
@pytest.fixture()
def summary_object(flat_data: list[int]) -> Summary:
    summary = Summary()
    for sample in flat_data:
        summary.add(sample)
    return summary

def test_mean(summary_object: Summary) -> None:
    assert summary_object.mean == 500

def test_median(summary_object: Summary) -> None:
    assert summary_object.median == 500
```

Since each of these test functions are run separately, we'll see separate error reports for problems when computing the mean and the median, or possibly when computing both.

## See also

- The *Unit testing with the unittest module* recipe in this chapter covers the same test case from the perspective of the `unittest` module.

## Combining pytest and doctest tests

In most cases, we'll have a combination of `pytest` and `doctest` test cases. For examples of using the `doctest` tool, see the *Using docstrings for testing* recipe. For examples of using the `pytest` tool, see the *Unit testing with the pytest module* recipe.

Frequently, documentation will contain `doctest`. We need to be sure all examples – in `docstrings` and documentation – work correctly. In this recipe, we'll combine these `doctest` examples and the `pytest` test cases into one tidy package.



## Getting ready

We'll refer back to the example from the *Using docstrings for testing* recipe. This recipe created tests for a class, `Summary`, that does some statistical calculations. In that recipe, we included examples in the docstrings.

In the *Unit testing with the pytest module* recipe, we wrote some test functions to provide additional tests for this class. These tests were put into a separate module, with a name starting with `test_`, specifically, `test_summary.py`.

Following the *Combining unittest and doctest tests* recipe, we'll also assume there's a project folder structure that looks like the following directory tree:

```
project-name/  
  src/  
    summary.py  
  tests/  
    test_summary.py  
  README  
  pyproject.toml  
  requirements.txt  
  tox.ini
```

The `tests` directory should contain all the module files with tests. We've chosen the directory named `tests` and a module named `test_*.py` so that they fit well with the automated test discovery features of the `pytest` tool.

The recipe examples use `recipe_07` instead of a cooler name such as `summary`. As a general practice, a module should have a memorable, meaningful name. The book's content is quite large, and the names are designed to match the overall chapter and recipe outline.

## How to do it...

It turns out that we don't need to write any Python code to combine the tests. The **pytest** module will locate test functions. It can also be used to locate doctest cases:

1. Create a shell command to run the test suite in the `recipe_07.py` file, as well as to examine the `recipe_01.py` module for the additional doctest cases:

```
% pytest recipe_07.py --doctest-modules recipe_01.py
```

When we run this from the OS command prompt, we see the following:

```
(cookbook3) % pytest recipe_07.py --doctest-modules recipe_01.py
===== test session starts =====
platform darwin -- Python 3.12.0, pytest-7.4.3, pluggy-1.3.0
rootdir: /Users/slott/Documents/Writing/Python/Python Cookbook 3e
configfile: pytest.ini
plugins: anyio-4.0.0
collected 7 items

recipe_07.py ..                               [ 28%]
recipe_01.py .....                            [100%]

===== 7 passed in 0.06s =====
```

The `pytest` command worked with both files. The dots after `recipe_07.py` show that two test cases were found in this file. This was 28% of the test suite. The dots after `recipe_01.py` show that five test cases more were found; this was the remaining 72% of the suite.

This shows us that the `pytest` test cases were included as well as `doctest` test cases. What's helpful is that we don't have to adjust anything in either of the test suites to execute all of the available test cases.

## How it works...

The `pytest` application has a variety of ways to search for test cases. The default is to search the given paths for all functions with names that start with `test_` in a given module. It will also search for all subclasses of `TestCase`. If we provide a directory, it will search it for all modules with names that begin with `test_`. Often, we'll collect our test files in a directory named `tests` because this is the default directory that will be searched.

The `--doctest-modules` command-line option is used to mark modules that contain `doctest` examples. These examples are also added to the test suite as test cases.

This level of sophistication in finding and executing a variety of types of tests makes

**pytest** a very powerful tool. It makes it easy to create tests in a variety of forms to create confidence that our software will work as intended.

## There's more...

Adding the `-v` option provides a more detailed view of the tests found by the `pytest` tool. Here's how the additional details are displayed:

```
(cookbook3) % python -m pytest -v recipe_07.py --doctest-modules
recipe_01.py
===== test session starts =====
platform darwin -- Python 3.12.0, pytest-7.4.3, pluggy-1.3.0 --
/Users/slott/miniconda3/envs/cookbook3/bin/python
cachedir: .pytest_cache
rootdir: /Users/slott/Documents/Writing/Python/Python Cookbook 3e
configfile: pytest.ini
plugins: anyio-4.0.0
collected 7 items

recipe_07.py::recipe_07.__test__.test_example_class PASSED [ 14%]
recipe_07.py::test_flat PASSED [ 28%]
recipe_01.py::recipe_01.Summary PASSED [ 42%]
recipe_01.py::recipe_01.__test__.GIVEN_binom_WHEN_0_0_THEN_1 PASSED [ 57%]
recipe_01.py::recipe_01.__test__.GIVEN_binom_WHEN_52_52_THEN_1 PASSED [ 71%]
recipe_01.py::recipe_01.binom PASSED [ 85%]
recipe_01.py::recipe_01.binom2 PASSED [100%]

===== 7 passed in 0.05s =====
```

Each individual test is identified, providing us with a detailed explanation of the test processing. This can help confirm that all of the expected doctest examples were properly located in the module under test.

## See also

- For examples of doctest, see the *Using docstrings for testing* recipe earlier in this chapter.
- The *Unit testing with the pytest module* recipe earlier in this chapter has the `pytest` test cases used for this recipe.

- For examples of the unittest version of these tests, see the *Unit testing with the unittest module* recipe earlier in this chapter.

## Testing things that involve dates or times

Many applications rely on functions like `datetime.datetime.now()` or `time.time()` to create a timestamp. When we use one of these functions with a unit test, the results are essentially impossible to predict. This is an interesting dependency injection problem here: our application depends on a class that we would like to replace only when we're testing.

One option is to design our application to avoid functions like `now()`. Instead of using this method directly, we can create a factory function that emits timestamps. For test purposes, this function can be replaced with one that produces known results.

The alternative is called *monkey-patching* – injecting a new object at test time. This can reduce the design complexity; it tends to increase the test complexity.

In this recipe, we'll write tests with `datetime` objects. We'll need to create mock objects for `datetime` instances to create repeatable test values. We'll use the **pytest** package features for monkey-patching.

### Getting ready

We'll work with a small function that creates a CSV file. This file's name will include the date and time in the format of `YYYYMMDDHHMMSS` as a long string of digits. This kind of file-naming convention might be used by a long-running server application. The name helps match a file and related log events. It can help to trace the work being done by the server.

The application uses this function to create these files:

```
import datetime
import json
from pathlib import Path
from typing import Any
```

```
def save_data(base: Path, some_payload: Any) -> None:
    now_date = datetime.datetime.now(tz=datetime.timezone.utc)
    now_text = now_date.strftime("extract_%Y%m%d%H%M%S")
    file_path = (base / now_text).with_suffix(".json")
    with file_path.open("w") as target_file:
        json.dump(some_payload, target_file, indent=2)
```

This function has the use of `now()`, which produces a distinct value each time this is run. Since this value is difficult to predict, it makes test assertions difficult to write.

To create a reproducible test output, we can create a mock version of the `datetime` module. We can then *monkey patch* the test context to use this mock object instead of the actual `datetime` module. Within the mocked module, we can create a mock class with a mock `now()` method to provide a fixed, easy-to-test response.

For this case, we have a scenario like this:

```
Scenario: save_date function writes JSON data to a date-stamped file.

Given a base directory Path
And a payload object {"primes": [2, 3, 5, 7, 11, 13, 17, 19]}
And a known date and time of 2017-9-10 11:12:13 UTC
When save_data(base, payload) function is executed
Then the output file of "extract_20170910111213.json" is found in the base
directory
And the output file has a properly serialized version of the payload
And the datetime.datetime.now() function was called once to get the date and
time
```

This can be implemented as a test case using the **pytest** constructs.

## How to do it...

This recipe will create and patch mock objects to create a test fixture:

1. We'll need to import a number of modules required by the module we're testing:

```
import datetime
import json
from pathlib import Path
```

2. We'll also need the core tools for creating mock objects and test fixtures. Also, we'll need the module we're going to test:

```
from unittest.mock import Mock
import pytest

import recipe_08
```

3. We must create an object that will behave like the `datetime` module for the purposes of the test scenario. This mocked module must contain a name that appears to be a class, also named `datetime`. The class must appear to contain a method, `now()`, which returns a known object rather than a date that changes each time the test is run. We'll create a fixture, and the fixture will return this mock object with a small set of attributes and behaviors defined:

```
@pytest.fixture()
def mock_datetime() -> Mock:
    return Mock(
        name="mock datetime",
        datetime=Mock(
            name="mock datetime.datetime",
            now=Mock(return_value=datetime.datetime(2017, 9, 10, 11,
12, 13)),
        ),
        timezone=Mock(name="mock datetime.timezone",
            utc=Mock(name="UTC")),
    )
```

The `Mock` object is a *namespace*, a feature that packages, modules, and classes all share. In this example, each attribute name is another `Mock` object. The most deeply-buried object has a `return_value` attribute to make it behave like a function.

4. We also need a way to isolate the behavior of the filesystem into test directories. The

tmp\_path fixture is built in to pytest and provides temporary directories into which test files can be written safely.

5. We can now define a test function that will use the mock\_datetime fixture and the tmp\_path fixture. It will use the monkeypatch fixture to adjust the context of the module under test:

```
def test_save_data(
    mock_datetime: Mock, tmp_path: Path, monkeypatch:
    pytest.MonkeyPatch
) -> None:
```

6. We can use the monkeypatch fixture to replace an attribute of the recipe\_08 module. The datetime attribute value will be replaced with the Mock object created by the mock\_datetime fixture:

```
monkeypatch.setattr(recipe_08, "datetime", mock_datetime)
```

Between the fixture definitions and this patch, we've created a *Given* step that defines the test arrangement.

7. We can now exercise the save\_data() function in a controlled test environment. This is the *When* step that exercises the code under test:

```
data = {"primes": [2, 3, 5, 7, 11, 13, 17, 19]}
recipe_08.save_data(tmp_path, data)
```

8. Since the date and time are fixed by the Mock object, the output file has a known, predictable name. We can read and validate the expected data in the file. Further, we can interrogate the Mock object to be sure it was called exactly once with no argument values. This is a *Then* step to confirm the expected results:

```
expected_path = tmp_path / "extract_20170910111213.json"
with expected_path.open() as result_file:
    result_data = json.load(result_file)
assert data == result_data
```

```
mock_datetime.datetime.now.assert_called_once_with(tz=mock_datetime.  
                                                    timezone.utc)
```

This test confirms the application's `save_data()` function will create the expected file with the proper content.

## How it works...

The `unittest.mock` module has a wonderfully sophisticated class definition, the `Mock` class. A `Mock` object can behave like other Python objects, while offering a limited subset of behaviors. In this example, we've created three different kinds of `Mock` objects.

The `Mock(wraps="datetime", ...)` object mocks a complete module. It will behave, to the extent needed by this test scenario, like the standard library `datetime` module. Within this object, we created a mock class definition but didn't assign it to any variable.

The `Mock(now=...)` object behaves like a mock class definition inside the mock module. We've created a single `now` attribute value, which will behave like a static function.

The `Mock(return_value=...)` object behaves like an ordinary function or method. We provide the return value required for this test.

In addition to returning the given value, a `Mock` object records the history of calls. This means an assertion can check those calls. The `call()` function from the `Mock` module provides a way to describe the arguments that are expected in the function call.

## There's more...

In this example, we created a mock for the `datetime` module that had a very narrow feature set for this test. The module contained a mocked class named `datetime`. This class has a single attribute, a mocked function, `now()`.

Instead of the `return_value` attribute, we can use the `side_effect` attribute to raise an exception instead of returning a value. We can use this to spot code that's not using the `now()` method properly, but using the deprecated `utcnow()` or the `today()` methods.



We can extend this pattern and mock more than one attribute to behave like a function. Here's an example that mocks several functions:

```
@pytest.fixture()
def mock_datetime_now() -> Mock:
    return Mock(
        name="mock datetime",
        datetime=Mock(
            name="mock datetime.datetime",
            utcnow=Mock(side_effect=AssertionError("Convert to now()")),
            today=Mock(side_effect=AssertionError("Convert to now()")),
            now=Mock(return_value=datetime.datetime(2017, 7, 4, 4, 2, 3)),
        ),
```

Two of the mocked methods, `utcnow()` and `today()`, each define a side effect that will raise an exception. This allows us confirm legacy code has been converted to make proper use of the `now()` method.

## See also

- The *Unit testing with the unittest module* recipe earlier in this chapter has more information about the basic use of the `unittest` module.

## Testing things that involve randomness

Many applications rely on the `random` module to create random values or put values into a random order. In many statistical tests, repeated random shuffling or random selection is done. When we want to test one of these algorithms, any intermediate results or details of the processing are essentially impossible to predict.

We have two choices for trying to make the `random` module predictable enough to write detailed unit tests:

- Use the `random` module with a known seed value.
- Use a `Mock` object to replace the `random` module with a `Mock` object to produce predictable values.

In this recipe, we'll look at ways to unit test algorithms that involve randomness.

## Getting ready

Given a sample dataset, we can compute a statistical measure such as a mean or median. A common next step is to determine the likely values of these statistical measures for some overall population. This can be done by a technique called **bootstrapping**.

The idea is to resample the initial set of data repeatedly. Each of the resamples provides a different estimate of the statistical measures for the population.

In order to be sure that a resampling algorithm is implemented correctly, it helps to eliminate randomness from the processing. We can resample a carefully planned set of data with a non-randomized version of the `random.choice()` function. If this works properly, then we have confidence that the randomized version will also work.

Here's our candidate resampling function:

```
from collections.abc import Iterator
import random

def resample(population: list[int], N: int) -> Iterator[int]:
    for i in range(N):
        sample = random.choice(population)
        yield sample
```

For our example, we'll compute alternative values of the mean based on resampling. The overall resampling procedure looks like this:

```
from collections import Counter
import statistics

def mean_distribution(population: list[int], N: int) -> Counter[float]:
    means: Counter[float] = Counter()
    for _ in range(1000):
        subset = list(resample(population, N))
        measure = round(statistics.mean(subset), 1)
        means[measure] += 1
```

**return** means

This evaluates the `resample()` function to create a number of subsets. Each subset's mean populates the means collection. The histogram created by this `mean_distribution()` function will provide a helpful estimate for population variance.

Here's what the output looks like:

```
>>> random.seed(42)
>>> population = [8.04, 6.95, 7.58, 8.81, 8.33, 9.96, 7.24, 4.26, 10.84,
4.82, 5.68]

>>> mean_distribution(population, 4).most_common(5)
[(7.8, 51), (7.2, 45), (7.5, 44), (7.1, 41), (7.7, 40)]
```

This shows us that the most likely value for the mean of the overall population could be between 7.1 and 7.8. There's more to this kind of analysis than we're showing here. Our focus is limited to the narrow question of testing the `resample()` function.

The test for resampling involves a scenario like the following:

**Scenario:** Resample example

**Given** a random number generator where `choice()` always return the sequence `[23, 29, 31, 37, 41, 43, 47, 53]`

**When** we evaluate the expression `resample(any 8 values, 8)`

**Then** the expected results are `[23, 29, 31, 37, 41, 43, 47, 53]`

**And** the `choice()` function was called 8 times

## How to do it...

We'll define a mock object that can be used instead of the `random.choice()` function. With this fixture in place, the results are fixed and predictable:

1. We'll need the core tools for creating mock objects and test fixtures. Also, we'll need the module we're going to test:

```
from unittest.mock import Mock
import pytest
import recipe_09
```

2. We'll need an object that will behave like the `choice()` function. We'll create a fixture built on another fixture:

```
@pytest.fixture()
def expected_resample_data() -> list[int]:
    return [23, 29, 31, 37, 41, 43, 47, 53]

@pytest.fixture()
def mock_random_choice(expected_resample_data: list[int]) -> Mock:
    mock_choice = Mock(name="mock_random.choice",
                        side_effect=expected_resample_data)
    return mock_choice
```

The `expected_resample_data` fixture provides a specific list of values that will provide expected results. Using this fixture, the `mock_random_choice` choice fixture returns the expected values in response to the `choice()` function.

3. We can now define a test function that will use the `mock_random_choice` fixture, which creates a mock object, and the `monkeypatch` fixture, which lets us adjust the context of the module under test:

```
def test_resample(
    mock_random_choice: Mock,
    expected_resample_data: list[int],
    monkeypatch: pytest.MonkeyPatch,
) -> None:
```

4. We can use the `monkeypatch` fixture to replace the `choice` attribute of the `random` module with the `Mock` object created by the `mock_random_choice` fixture:

```
monkeypatch.setattr(recipe_09.random, "choice",
                    mock_random_choice) # type: ignore [attr-defined]
```

Between the fixture definitions and this patch, we've created a *Given* step that defines the test arrangement.

5. We can now exercise the `resample()` function in a controlled test environment. This is the *When* step that exercises the code under test:

```
data = [2, 3, 5, 7, 11, 13, 17, 19]
resample_data = list(recipe_09.resample(data, 8))
```

6. Since the random choices are fixed by the `Mock` object, the result is fixed. We can confirm that the data created by the `mock_random_choice` fixture was used for re-sampling. We can also confirm that the mocked choice function was properly called with the input data:

```
assert resample_data == expected_resample_data
assert mock_random_choice.mock_calls == 8 * [call(data)]
```

This test helps to confirm that our `resample()` function will create the output based on the given input and the `random.choice()` function.

## How it works...

When we create a `Mock` object, we must provide the methods and attributes to define the behavior of the object being mocked. When we create an instance of `Mock` that provides the `side_effect` argument value, we're creating a callable object. The callable object will return the next value from the `side_effect` sequence each time the `Mock` object is called. This gives us a handy way to mock iterators.

If any value in `side_effect` is an exception, this is raised.

We can also see the call history using the `mock_calls` attribute of a `Mock` object. This lets us confirm that the callable was provided proper argument values.

## There's more...

The `resample()` function has an interesting pattern to it. When we take a step back from the details, we see this:

```
def resample_pattern(X: Any, Y: Any) -> Iterator[Any]:
    for _ in range(Y):
        yield another_function(X)
```

The `X` argument value is simply passed through to another function without any processing. For testing purposes, it doesn't matter what the value of `X` is. What we're testing is that the parameter's value in the `resample()` function is provided to the `another_function()` function, untouched.

The `mock` library provides an object called `sentinel` that can be used to create an opaque argument value in these circumstances. When we refer to an attribute of the `sentinel` object, this reference creates a distinct object. We might use `sentinel.POPULATION` as a kind of mock for a collection of values. The exact collection doesn't matter since it's simply passed as an argument to another function (called `random.choice()` in the actual code).

Here's how this use of a `sentinel` object can change this test:

```
from unittest.mock import Mock, call, sentinel

@pytest.fixture()
def mock_choice_s() -> Mock:
    mock_choice = Mock(name="mock random.choice()",
                       return_value=sentinel.CHOICE)
    return mock_choice

def test_resample_2(
    mock_choice_s: Mock, monkeypatch: pytest.MonkeyPatch
) -> None:
    monkeypatch.setattr(
        recipe_09.random, "choice", mock_choice_s # type: ignore
        [attr-defined]
    )
    resample_data = list(recipe_09.resample(sentinel.POPULATION, 8))
```

```
assert resample_data == [sentinel.CHOICE] * 8
```

The output from the mocked `choice()` function is a recognizable sentinel object. Similarly, the parameter to the `resample()` function is a different sentinel object. We expect this to be called 8 times, because the `N` parameter is set to 8 in the test case.

When an object should pass through a function untouched, we can write test assertions to confirm this expected behavior. If the code we're testing uses the population object improperly, the test can fail when the result is not the untouched sentinel object.



The 1.7.1 release of the **mypy** tool struggles with the imports in the `recipe_09` module. We used a `# type: ignore [attr-defined]` comment to suppress a confusing mypy message.

This test gives us confidence that the population of values is provided, untouched, to the `random.choice()` function and the `N` parameter value defines the size of the returned set of items from the population.

## See also

- The *Building sets – literals, adding, comprehensions, and operators* recipe in *Chapter 4*, and the *Creating dictionaries – inserting and updating* recipe in *Chapter 5*, the *Using cmd to create command-line applications* recipe in *Chapter 6* show how to seed the random number generator to create a predictable sequence of values.
- In *Chapter 7*, there are several other recipes that show an alternative approach, for example, *Using a class to encapsulate data and processing*, *Designing classes with lots of processing*, *Optimizing small objects with \_\_slots\_\_*, and *Using properties for lazy attributes*.
- Also, in *Chapter 8*, see the *Choosing between inheritance and composition – the "is-a" question*, *Separating concerns via multiple inheritance*, *Leveraging Python's duck typing*, and *Creating a class that has orderable objects* recipes.

## Mocking external resources

In earlier recipes in this chapter, namely, *Testing things that involve dates or times* and *Testing things that involve randomness*, we wrote tests for involving resources with states that we could predict and mock. In one case, we created a mock `datetime` module that had a fixed response for the current time. In the other case, we created a mock function from the `random` module.

A Python application can use the `os`, `subprocess`, and `pathlib` modules to make significant changes to the internal states of a running computer. We'd like to be able to test these external requests in a safe environment, using mocked objects, and avoid the horror of corrupting a working system with a misconfigured test. Another example is database access, which requires mock objects to respond to create, retrieve, update, and delete requests.

In this recipe, we'll look at ways to create more sophisticated mock objects. These will allow the safe testing of changes to precious OS resources like files and directories.

### Getting ready

We'll revisit an application that makes a number of OS changes. In *Chapter 11*, the *Replacing a file while preserving the previous version* recipe showed how to write a new file and then rename it so that the previous copy was always preserved.

A thorough set of test cases would present a variety of failure modes. Having tests for several different kinds of errors can help provide confidence that the function behaves properly.

The essential design was a definition of a class of objects, `Quotient`, and a `save_data()` function to write one of those objects to a file. Here's an overview of the code:

```
from pathlib import Path
import csv
from dataclasses import dataclass, asdict, fields

@dataclass
class Quotient:
```



```
    numerator: int
    denominator: int

def save_data(output_path: Path, data: Quotient) -> None:
```

```
    ... # Details omitted
```

Consider what happens when there's a failure in the middle of the `save_data()` function. Outcomes include a file that's partially rewritten, and useless to other applications. To prevent this, the recipe presented a `safe_write()` function that included several steps to create a temporary file and then rename that file to be the desired output file. Essentially, the function looked like this:

```
def safe_write(output_path: Path, data: Quotient) -> None:
```

```
    ... # Details omitted
```

The `safe_write()` function is shown in detail in *Chapter 11*. This is designed to handle a number of scenarios:

1. Everything works – sometimes called the “happy path” – and the file is created properly.
2. The `save_data()` function raises an exception. The corrupted file is removed and the original left in place.
3. Failure occurs elsewhere in the processing of the `safe_write()` processing. There are three scenarios where a `Path` method raises an exception.

Each of the above scenarios can be translated into Gherkin to help clarify precisely what it means; for example:

**Scenario:** `save_data()` function is broken.

**Given** some faulty set of data, `"faulty_data"`, that causes a failure in the `save_data()` function

**And** an existing file, `"important_data.csv"`

**When** `safe_write("important_data.csv", faulty_data)` is processed

**Then** `safe_write` raises an exception

**And** the existing file, `"important_data.csv"` is untouched

Detailing each of the five scenarios helps us define Mock objects to provide the various kinds of external resource behaviors we need. Each scenario suggests a distinct fixture to reflect the distinct failure mode.

## How to do it...

We'll use a variety of testing techniques. The `pytest` package offers the `tmp_path` fixture, which can be used to create isolated files and directories. In addition to an isolated directory, we'll also want to use a Mock to stand in for the parts of the application that we're not testing:

1. Identify all of the fixtures required for the various scenarios. For the happy path, where the mocking is minimal, the `tmp_path` fixture is all we need. For scenario two, where the `save_data()` function is broken, this function should be mocked. For the remaining three scenarios, mock objects can be defined that will replace methods of Path objects.
2. This test will use a number of features from the `pytest` and `unittest.mock` modules. It will be creating Path objects and test functions defined in the `recipe_10` module:

```
from pathlib import Path
from typing import Any

from unittest.mock import Mock, sentinel
import pytest

import recipe_10
```

- Write a test fixture to create the original file, which should not be disturbed unless everything works correctly. We'll use a `sentinel` object to provide some text that is unique and recognizable as part of this test scenario:

```
@pytest.fixture()
def original_file(tmp_path: Path) -> Path:
    precious_file = tmp_path / "important_data.csv"
    precious_file.write_text(hex(id(sentinel.ORIGINAL_DATA)),
                             encoding="utf-8")
    return precious_file
```

- Write a mock to replace the `save_data()` function. This will create mock data used to validate that the `safe_write()` function works. In this, too, we'll use a `sentinel` object to create a unique string that is recognizable later in the test:

```
def save_data_good(path: Path, content: recipe_10.Quotient) -> None:
    path.write_text(hex(id(sentinel.GOOD_DATA)), encoding="utf-8")
```

- Write the *happy path* scenario. The `save_data_good()` function can be given as the `side_effect` of a `Mock` object and used in place of the original `save_data()` function. Using a `Mock` means the call history will be tracked. This helps to confirm that the overall `safe_write()` function being tested really does use the `save_data()` function to create the expected resulting file:

```
def test_safe_write_happy(original_file: Path, monkeypatch:
    pytest.MonkeyPatch) -> None:
    mock_save_data = Mock(side_effect=save_data_good)
    monkeypatch.setattr(recipe_10, "save_data", mock_save_data)
    data = recipe_10.Quotient(355, 113)
    recipe_10.safe_write(Path(original_file), data)
    actual = original_file.read_text(encoding="utf-8")

    assert actual == hex(id(sentinel.GOOD_DATA))
```

- Write a mock for scenario two, in which the `save_data()` function fails to work correctly. The mock can rely on a `save_data_failure()` function to write recognizably corrupt data, and then also raise an unexpected exception:

```
def save_data_failure(path: Path, content: recipe_10.Quotient) ->
None:
    path.write_text(hex(id(sentinel.CORRUPT_DATA)), encoding="utf-8")
    raise RuntimeError("mock exception")
```

7. Write the test case for scenario two, using the `save_data_failure()` function as the `side_effect` of a `Mock` object:

```
def test_safe_write_scenario_2(
    original_file: Path, monkeypatch: pytest.MonkeyPatch
) -> None:
    mock_save_data = Mock(side_effect=save_data_failure)
    monkeypatch.setattr(recipe_10, "save_data", mock_save_data)
    data = recipe_10.Quotient(355, 113)
    with pytest.raises(RuntimeError) as ex:
        recipe_10.safe_write(Path(original_file), data)
    actual = original_file.read_text(encoding="utf-8")
    assert actual == hex(id(sentinel.ORIGINAL_DATA))
```

The `save_data_failure()` function wrote corrupt data, but the `safe_write()` function preserved the original file.

This recipe produced two test scenarios that confirm the `safe_write()` function will work. We'll turn to the remaining three scenarios in the *There's more...* section later in this recipe.

## How it works...

When testing software that makes OS, network, or database requests, it's imperative to include cases where the external resource fails to operate as expected. The principal tools for doing this are `Mock` objects and the `monkeypatch` fixture. A test can replace Python library functions with `Mock` objects that raise exceptions instead of working correctly.

For the happy path scenario, we replaced the `save_data()` function with a `Mock` object that wrote some recognizable data. Because we're using the `tmp_path` fixture, the file was written into a safe, temporary directory, where it could be examined to confirm that new, good data replaced the original data.

For the first of the failure scenarios, we used the monkeypatch fixture to replace the `save_data()` function with a function that both wrote corrupt data and also raised an exception as if an OS problem occurred. This is one way to simulate a broad spectrum of application failures that involve some kind of persistent filesystem artifact. In simpler cases, where there is no artifact, a `Mock` object with an exception class as the value of the `side_effect` parameter is all that's required to simulate a failure.

These test scenarios also made use of unique sentinel objects. Evaluating the value of the `hex(id(x))` provides a distinct string value that's difficult to predict.

## There's more...

The remaining scenarios are very similar; they can all share the following test function:

```
def test_safe_write_scenarios(
    original_file: Path,
    mock_pathlib_path: Mock,
    monkeypatch: pytest.MonkeyPatch
) -> None:
    mock_save_data = Mock(side_effect=save_data_good)
    monkeypatch.setattr(recipe_10, "save_data", mock_save_data)
    data = recipe_10.Quotient(355, 113)
    with pytest.raises(RuntimeError) as exc_info:
        recipe_10.safe_write(mock_pathlib_path, data)
    actual = original_file.read_text(encoding="utf-8")
    assert actual == hex(id(sentinel.ORIGINAL_DATA))
    mock_save_data.assert_called_once()
    mock_pathlib_path.with_suffix.mock_calls == [
        call("suffix.new"), call("suffix.old")
    ]
    # Scenario-specific details...
```

This function uses the `save_data_good()` function as the side effect when the mocked `save_data()` function is invoked. The given `save_data_good()` function will be executed and will write a known good test file. Each of these scenarios involve exceptions from `Path` operations after the good file was created.

We've omitted showing any scenario-specific details. The key feature of this test is pre-

servicing the original good data in spite of exceptions.

To support multiple exception scenarios, we want to use three different versions of the `mock_pathlib_path` mock object.

We can use a parameterized fixture to spell out these three alternative configurations of the mock objects. First, we'll package the choices as three separate dictionaries that provide the `side_effect` values:

```
scenario_3 = {
    "original": None, "new": None, "old": RuntimeError("3")}

scenario_4 = {
    "original": RuntimeError("4"), "new": None, "old": None}

scenario_5 = {
    "original": None, "new": RuntimeError("5"), "old": None}
```

We've used `RuntimeError` as the exception to raise, triggering alternative execution paths. In some cases, it may be necessary to use a `IOError` exception. In this case, any exception would be fine.

Given these three dictionary objects, we can plug the values into a fixture via the `request.params` option provided by `pytest`:

```
@pytest.fixture(
    params=[scenario_3, scenario_4, scenario_5],
)
def mock_pathlib_path(request: pytest.FixtureRequest) -> Mock:
    mock_mapping = request.param
    new_path = Mock(rename=Mock(side_effect=mock_mapping["new"]))
    old_path = Mock(unlink=Mock(side_effect=mock_mapping["old"]))
    output_path = Mock(
        name="mock_output_path",
        suffix="suffix",
        with_suffix=Mock(side_effect=[new_path, old_path]),
        rename=Mock(side_effect=mock_mapping["original"]),
    )
    return output_path
```

Because this fixture has three parameter values, any test using this fixture will be run three times, once with each of the values. This lets us reuse the `test_safe_write_scenarios()` test case to be sure it works with a variety of system failures.

We've created a variety of mock objects to inject failures throughout the processing in a complex function. Using parameterized fixture helps to define consistent mock objects for these tests.

There's yet another scenario that involves a successful operation followed by a failing operation on the same file. This doesn't fit the above pattern and requires another test case with a slightly more sophisticated set of mock objects. We leave this as an exercise for you.

## See also

- The *Testing things that involve dates or times* and *Testing things that involve randomness* recipes earlier in this chapter show techniques for dealing with unpredictable data.
- Elements of this can be tested with the `doctest` module. See the *Using docstrings for testing* recipe earlier in this chapter for examples. It's also important to combine these tests with any doctests. See the *Combining pytest and doctest tests* recipe earlier in this chapter for more information on how to do this.

## Join our community Discord space

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

<https://packt.link/dHrHU>







# 16

## Dependencies and Virtual Environments

Python runs in an environment defined by the OS. There are some slight differences between Windows, macOS, and most Linux environments. We'll set aside micro-controller environments, since the ability to tailor those environments is quite a bit more involved. We'll try to minimize the OS differences to focus on the common aspects that are universally available.

There are several common aspects within run-time environments. We can divide these into two groups:

**Persistent** Aspects of the environment that change slowly.

- The Python run-time in use. This includes a binary application and often includes a number of external libraries.
- The standard libraries available. These are accessed via the importer, and are generally available via the `import` statement. They are generally found by their path, relative

to the Python binary.

- The other libraries installed as *site packages*. These are also accessed by the importer. These libraries are also found by their path, relative to the Python binary.
- Libraries found by other mechanisms available in the `sites` package. Most notably, the `PYTHONPATH` environment variable.

**Transient** Aspects of the environment can change each time the Python run-time is started.

- The environment variables defined by the shell in use. These are available through the `os` module.
- The current working directory and user information, defined by the OS. This is available through the `os`, `os.path`, and `pathlib` modules.
- The command line used to start Python. This is available through several attributes of the `sys` module, including `sys.argv`, `sys.stdout`, `sys.stdin`, and `sys.stderr`.

The persistent environment is managed via OS-level commands, outside of our application programs. Changes to the persistent aspects of the environment are generally examined once when Python starts. This means an application we write can't easily install a package and then use that package.

The persistent environment has two viewpoints:

**The Actual Environment** : A single site is handled by the system administrator and requires elevated privileges. For example, the Python run-time is often in a path owned by the root user and made visible through a common, system-wide value of the `PATH` environment variable.

**Virtual Environments** : Any number of virtual environments are localized by individual users and require no special privileges. Multiple Python run-times and their associated site packages can be owned by a single user.

Once upon a time — in the long-past olden days, when computer capabilities were tiny

— a single actual environment was all that could be managed. Adding and changing the collection of installed packages required cooperation among Python users. An administrator with elevated privileges implemented any changes.

Now that computers are vastly more capable, each individual user can easily have multiple virtual environments. Indeed, we often build and test modules with numerous virtual environments reflecting different releases of the Python run-time. Each individual is able to manage their own virtual environments.

When working cooperatively, it becomes important to share the details of virtual environments so that multiple users can recreate a common virtual environment. The effort of sharing a single actual environment is shifted to each user having to prepare and manage their own virtual environment.

Environment management seems to parallel Ginsberg's Theorem and the Laws of Thermodynamics:



- The overall environment management workload is neither created nor destroyed.
- Any change to an environment requires work.
- Nothing is free from change (unless it's utterly isolated from all external considerations).

While most Linux distributions come with Python pre-installed, there's no compelling reason to use this version of Python for any purpose. It's generally much easier to install a personal version of Python and manage virtual environments with that personal version. Having an individual Python installation permits ready updates to new releases without waiting for a Linux distribution to catch up with the state of the art.

There are two broad classes of tools involved in managing environments:

- OS-specific tools required to install the Python binary. This varies by OS and can be challenging to new developers. We'll avoid the complications involved in these tools

and refer readers to the <https://www.python.org/downloads/> page.

- Python-based tools, like **PIP**, used to install Python libraries. Since these tools depend on Python, the commands are universal for all OSs. This chapter will focus on these tools.

In this chapter, we'll look at the following recipes for managing virtual environments:

- *Creating environments using the built-in venv*
- *Installing packages with a requirements.txt file*
- *Creating a pyproject.toml file*
- *Using pip-tools to manage the requirements.txt file*
- *Using Anaconda and the conda tool*
- *Using the poetry tool*
- *Coping with changes in dependencies*

We'll start with creating virtual environments using the built-in tools.

## Creating environments using the built-in venv

Once Python is installed, creating virtual environments unique to each project can be done with the internal venv module.

There are two principle use cases for a virtual environment:

- Manage the Python version. We might have distinct virtual environments for Python 3.12 and Python 3.13. In some cases, we may need to manage multiple minor releases of Python 3.13.
- Manage the mix of site-specific packages required by our project. Rather than trying to update the single actual environment, we can create new virtual environments as new releases of packages become available.

These two use cases overlap a great deal. Each release of Python will have distinct versions of the standard library packages and may have distinct versions of external site-specific packages.

The most important part of using a virtual environment is making sure that it has been activated. A number of scenarios will change the internal state of the browser, deactivating the virtual environment. Closing a terminal window and rebooting the computer are two of the most common ways to deactivate an environment.

Changing terminal windows or opening a new terminal window may start a shell environment in which the virtual environment is not active. This is easily remedied by activating the environment before starting to use it.

## Getting ready

It's important to note that Python must be installed. Python may not be present, and whatever Python version is part of the OS should not be used for development or experimentation. For macOS and Windows, it's common to install a pre-built binary. This may involve downloading a disk image and running an installer or downloading an installer application and running it.

For Linux, it's common to build Python from source for the given distribution. An alternative is to use an administrative tool like **rpm**, **pkg**, **yum**, or **aptitude** to install a pre-built Python for the specific distribution.

Most Python releases will include the `pip` and `venv` packages. Microcontroller Python and WASM-based Python are often difficult to update using desktop tools; they're outside the scope of this book.

## How to do it...

First, we'll look at creating a virtual environment that can be used to install packages and resolve imports. Once the environment has been created, we'll look at activating and deactivating it. The environment must be active in order to properly install and use packages.

It's important to avoid putting a virtual environment under configuration control. Instead, the configuration details required to recreate the environment are put under configuration control.

When using tools like **Git**, a `.gitignore` file can be used to ignore any virtual environment details of a project. An alternative approach is to separate virtual environment definitions from specific project directories.

## Create a virtual environment

1. First, create the project directory. For very small projects, no additional files are needed. For most projects, `src`, `tests`, and `docs` directories are often helpful for organizing the project code, the test code, and the documentation.
2. Choose between a “concealed” or a visible file. In Linux and macOS, files with names that start with `.` are generally not shown by most commands. Since the virtual environment is not a directory we'll ever work with, it's often simplest to use the name `.venv`.

In some cases, we want the directory to be visible. Then, the `venv` name would be the best choice.

3. The following command will create a virtual environment:

```
% python -m venv .venv
```

The virtual environment will be in the `.venv` directory within the project directory.

After this is done, the virtual environment must be activated. Any time a new terminal window is opened, the environment in that window needs to be activated.

## Activate and deactivate an environment

Activating a virtual environment requires an OS-specific command. The Python *Standard Library* documentation provides all of the variant commands. We'll show the two most common variants:

- For Linux and macOS, using **bash** or **zsh**, enter the following command to activate a virtual environment:

```
% source .venv/bin/activate
```

- For Windows, enter the following command to activate a virtual environment:

```
> .venv\Scripts\activate
```

Once the virtual environment has been activated, a number of environment variables will change. Most notably, the `PATH` environment variable will include the virtual environment's `bin` directory. This will, for example, make the `deactivate` command available. Additionally, the prompt will change to include the virtual environment's name. It might look like the following:

```
C:\Users\Administrator>.venv\Scripts\activate  
(.venv) C:\Users\Administrator>
```

On the first line, the default prompt shows the directory. On the second line, the prompt has `(.venv)` as a prefix to show that the virtual environment is now active.

Once the virtual environment has been activated, all further use of the `pip` command to install packages will be directed to the active environment. Any Python application that's run will search the active environment to install packages.

To deactivate an environment, use the following command:

```
% deactivate
```

The `activate` command created this new command as part of the virtual environment, so it's universally available for all OSs.



## How it works...

For most OSs, there are a few key environment variables that define a virtual environment. The `PATH` environment variable generally provides locations for finding the Python executable. In a Windows environment, this will also make the launcher, the `py` command, available.

The locations of the remaining Python elements are all relative to the executable. In particular, the standard library is an adjacent path, and this library has the `sites` package that handles all of the other details of locating installed packages.

The details of the virtual environment are defined by three directories and a configuration file.

The configuration file, `pyvenv.cfg`, provides a few important settings. The three directories are `bin`, `include`, and `lib`. (For Windows, these names are `Scripts`, `Include`, and `Lib`). The `bin` directory has script files that perform activation of the virtual environment. Setting the `PATH` environment variable makes these scripts available. This includes the `deactivate` command. Additionally, the `bin` directory contains a `pip` executable command and a link to the proper python binary.

## There's more...

There are a number of options in the `venv` command. Of these, two seem to be particularly useful:

- The `--without-pip` option skips the installation of a venv-specific copy of PIP. It seems better to use `python -m pip` than to rely on the virtual environment installation.
- The `--prompt` option can set a nicer environment name than `.venv`.

We'll often use a command like the following to activate an environment:

```
% python -m venv --prompt ch17 --without-pip .venv
```

This will ensure the prompt becomes (ch17) instead of the vague and potentially confusing (.venv).

## See also

- Once a virtual environment is created, we can add external libraries. See *Installing packages with a requirements.txt file* for advice on managing dependencies.

## Installing packages with a requirements.txt file

One of the significant strengths of Python is the vast ecosystem of packages available in libraries like the Python Package Index (PyPI) at <https://pypi.org>. It's easy to use the PIP tool to add libraries to an environment.

In some cases, this is — perhaps — too easy. All of the dependencies, starting with the libraries on which the Python run-time is built, are in a constant state of flux. Each has a distinct tempo for updates. In some cases, there is limited cooperation among the vast number of people involved.

To manage the constant change, it's important for people developing applications to track dependencies carefully. We suggest decomposing dependencies into three levels of specificity:

- **Generic, name-only dependencies:** For example, an application might need Beautiful Soup.
- **Filtered:** As the Beautiful Soup project evolves, there may be versions with known bugs, or that are missing essential features. We might want to narrow the dependency to omit or exclude a specific version, or require a version that is  $\geq 4.0$ .
- **Pinned (or Locked):** When it is time to build (and test) a specific virtual environment, it is essential to have a detailed list of the exact version numbers used for testing.

When we're first exploring data or a problem domain or candidate solutions, we may download a great many packages into a development environment. As a project matures, the virtual environment contents will shift. In some cases, we'll learn we don't need a

package; the unused packages will be ignored and should be removed. In other cases, the mix of packages will expand as new options are explored. Throughout this, the pinned version numbers may change to track acceptable versions of packages on which our project depends.

## Getting ready

It works out well to record generic dependencies in places like a `pyproject.toml` file. (We'll look at this in the *Creating a `pyproject.toml` file* recipe.)

The specific, pinned dependencies can be separated into a collection of requirements files. There are a number of dependency use cases, leading to a collection of closely related files.

The format for a requirements file is defined as part of the PIP documentation. See the **Requirements File Format** page of <https://packaging.python.org>.

## How to do it...

1. Gather the general requirements. It's best to look at the `import` statements to discern what packages are direct dependencies. We might find that a project uses `pydantic`, `beautifulsoup4`, `jupyterlab`, `matplotlib`, `pytest`, and `memray`.
2. Open a file named `requirements.txt` in the top-level directory of the project.
3. Each line of the file will have a requirements specifier with four pieces of information:
  - The package name. Note that *typo-squatting* is a prevalent problem with open source; be sure to find the correct, current repository for a package, not a similar-looking name.
  - Any extras needed. If present, these are enclosed in `[ ]`. For example, `rich [jupyter]` might be used when using the `rich` package for text styling with Jupyter Lab.
  - A version specifier. This has a comparison (`==`, `>=`, etc.), and a version as a dotted sequence of numbers. For example, `pillow>=10.2.0` selects any version

of the `pillow` package at or after version 10.2.0, avoiding a known vulnerability with version 10.1.0.

- If necessary, any further environment constraints separated by a `;`. For example, `sys_platform == 'win32'` might be used to provide a platform-specific requirement.

While complex conditions can be created, they're not often needed. It's best to avoid writing version information unless a specific bug fix, missing feature, or compatibility problem surfaces.

The full set of rules for this file is in the PEP 508 document.

Version specifiers are defined in the Python Packaging Guide. See the **Version specifiers** page of <https://packaging.python.org>.

For example, here is the list of dependencies:

```
pydantic
beautifulsoup4
types-beautifulsoup4
jupyterlab
matplotlib
pytest
memray
```

4. Activate the project's virtual environment (if it's not already activated):

```
% source .venv/bin/activate
```

5. Run the following command to install the latest versions of the named packages:

```
(ch17) % python -m pip install -r requirements.txt
```

The **PIP** application will find matching versions of the various packages and install them. Because some of these packages have complex layers of dependencies, the installation can be rather time-consuming the first time it's attempted.

This list of seven packages expands to about 111 packages in total that must be installed.

For many projects, this is all that's required to build a useful environment definition. In many cases, this base definition needs to have more specific version information provided. This is a separate recipe; see *Using pip-tools to manage the requirements.txt file*.

## How it works...

The **PIP** application uses the `-r` option to parse a file with required packages. Within this file, we can have simple lists of packages, and complex rules for locating the proper version of a package. We can even have other `-r` options to incorporate other files of requirements. Using multiple files can help organize very complex projects.

When we name a package **PIP**, it will examine the target's metadata to locate packages on which it depends. These transitive dependencies must be installed before the target package is installed. This means an internal lattice structure showing all of the dependencies must be built. This can involve downloading multiple copies of a package, as version constraints are resolved into a single, final list of packages to install.

While it's easy to use **PIP** to manually install a single package, this leads to confusion about what a project needs and what's currently installed in the virtual environment. Avoiding this requires a disciplined approach of always doing these two things when exploring a new package:

- Add the package to the `requirements.txt` file.
- Run `python -m pip install -r requirements.txt` to add packages to the current virtual environment.

When removing packages from the `requirements.txt` file, we can generally proceed by deleting the virtual environment and creating an entirely new one. This leads to the following sequence of commands being used:

```
% (ch17) deactivate
% python -m venv --clear --prompt ch17 .venv
% source .venv/bin/activate
% (ch17) python -m pip install -r requirements.txt
```

Because **PIP** maintains a cache of downloaded files, this environment will be rebuilt relatively quickly. The use of `requirements.txt` ensures the environment is built in a repeatable fashion.

## There's more...

It's very common to install components manually and uncover conflicts. For example, a colleague clones a repository and cannot run the unit test suite because the `requirements.txt` file is incomplete.

Another case is an audit of development environments. As new people join a team, they may install new releases of a package named in the `requirements.txt` file. To be confident everyone has the same version, it helps to freeze the version information for the packages in a virtual environment.

For both use cases, the `python -m pip freeze` command can be used. This will report *all* of the installed packages and the versions that were used. The output from this is in the same format as a `requirements` file.

We can use a command like the following:

```
% source .venv/bin/activate
% (ch17) python -m pip freeze >audit_sfl.txt
```

These output files can be compared to locate differences and repair environments that are not consistent with expectations.

Additionally, the output from the `pip freeze` subcommand can be used to replace a generic `requirements.txt` file with a file that specifically pins each and every package in use. While this is very easy, it's not terribly flexible because it provides specific versions. There are

better ways to build a `requirements.txt` file using `pip-tools`. We'll look at this in *Using pip-tools to manage the requirements.txt file*.

## See also

- See the *Creating environments using the built-in venv* recipe to see how to create a virtual environment.
- See the *Using pip-tools to manage the requirements.txt file* recipe for a way to manage dependencies.

## Creating a `pyproject.toml` file

In addition to a virtual environment and a clear list of dependencies, a project also benefits from an overall summary, in the form of a `pyproject.toml` file.

A `pyproject.toml` file is required by some Python tools and is helpful to have in general. It provides a central summary of the technical details of the project.

With the adoption of PEP 621, this file has become the expected place for metadata about a project. It replaces the older `setup.py` module.

This recipe is based on the Sample Project project in the packaging authority Git repository at <https://github.com/pypa>. The recipe is also based on the **Packaging Python Projects**, page, one of the Packaging Authority tutorials. See <https://packaging.python.org>.

## Getting ready

We'll assume the project is not a trivial single-file module but something larger. This means there will be a directory structure somewhat like the following:

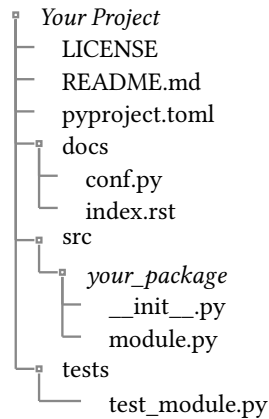


Figure 16.1: Project files

We’ve shown a common structure that applies broadly to many projects. The top-level name, *Your Project*, is a name that works for your collection of projects.

The name *your\_package* inside the `src` directory is the name by which the package will be known when it is imported. This does not have to precisely match the overall project name, but it should have a clear relationship. As an example, the Beautiful Soup project has a PYPI entry with the name `beautifulsoup4`, but the imported package is named `bs4` in your Python’s local site packages. The connection is clear.

We’ve shown the `README.md` file with an extension that indicates it’s written in Markdown notation. Common alternatives are `README.rst` and `README`.

The `LICENSE` file can be a difficult choice. See <https://spdx.org/licenses/> for a comprehensive list of open-source licenses. See **GNU License List** at <https://www.gnu.org> for advice on various open source licenses.

The content of the `docs` directory is often built using tools like Sphinx. We’ll address documentation in *Chapter 17*.



## How to do it...

1. Make sure the `README.md` has a summary of how to install and use the project. This is subject to change as the project evolves.

There are six essential questions: “who?”, “what?”, “why?”, “when?”, “where?”, and “how?” that can help write a short paragraph to describe the project. The C4 model offers additional help on how to describe software. See [C4 Model](#).

2. Determine which build system will be used. Choices include **setuptools**, **hatch**, and **poetry**. Parts of content of the `pyproject.toml` file will be unique to the build system.

For this recipe, we’ll use **setuptools** as the build tool.

3. There are numerous templates available for a `pyproject.toml` file. The PYPA sample project example is comprehensive, and perhaps a bit daunting. There are two tables in the TOML that are required: `[project]` and `[build-system]`. The rest can be ignored when getting started.

Here’s a short template for the `[project]` table:

```
[project]
name = "project_name"
version = "2024.1.0"
description = "A useful description."
requires-python = ">=3.12"
authors = [
  {email = "your.email@example.com", name = "Your Name"}
]
dependencies = [
  your dependencies
]
readme = "README.md"
license = {file = "LICENSE"}
```

The first six items need to have values replaced with facts about your project. The last two items, `readme` and `license`, don’t often change because they’re references to files in the project directory.

The name must be a valid identifier for a project. They are defined by PEP-508. They are names made of letters, digits, and the special characters -, \_, and .. Interestingly, they can't include spaces or end with a punctuation mark. `ch17-recipe3` is acceptable, but `ch17_` is invalid.

The dependencies must be a list of the direct requirements that must be installed in order for this project to work. These are the same kinds of dependency specifications provided in a `requirements.txt` file. See *Installing packages with a requirements.txt file* for more information.

Here's a template for the `[build-system]` table. This uses the small, widely available `setuptools` tool:

```
[build-system]
build-backend = "setuptools.build_meta"
requires = [
    "setuptools",
]
```

4. It can help to open this file with `tomllib` to confirm it's formatted properly. This can be done interactively in the Python console as follows:

```
>>> from pathlib import Path
>>> import tomllib
>>> doc = Path("pyproject.toml").read_text()
>>> tomllib.loads(doc)
```

If the file is invalid in some way, this will raise a `tomllib.TOMLDecodeError` exception. The exception will provide the line and column for the syntax error, or it will say “at end of document” when a structure isn't terminated properly.

## How it works...

A number of tools make use of the `pyproject.toml` contents. There is a complicated relationship between **PIP** and the build tool named in the `pyproject.toml` file. For this recipe, we're using **setuptools**.

The following diagram summarizes some of the steps involved in downloading and installing a library:

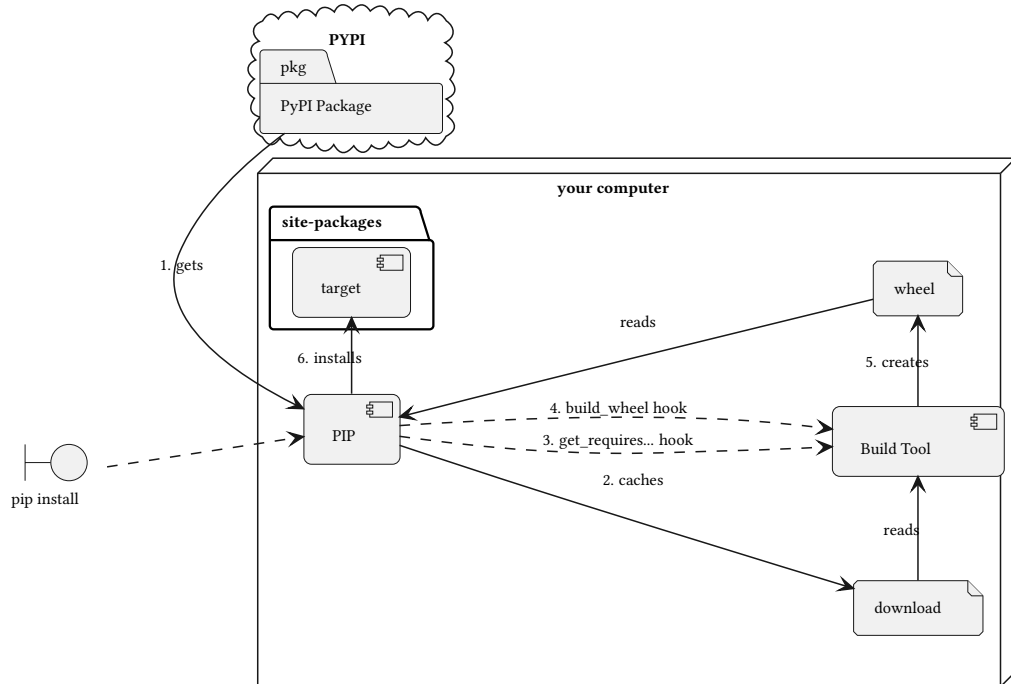


Figure 16.2: How PIP and the build tool collaborate

This summary diagram is neither an exhaustive nor definitive look at how packages are installed. For more information, see PEP-517.

The processing begins with the `pip install` command, shown with a boundary icon. The **PIP** operation proceeds through the numbered steps:

1. **PIP** starts by getting the compressed archive from a package index like PYPI.
2. The archive is cached on the local computer for future use.
3. **PIP** uses the `get_requires_for_build_wheel` build-tool hook to gather requirements. The build tool gets dependency information from the `pyproject.toml` file and provides this to **PIP**. The **PIP** tool will download these additional projects. These

projects have their own requirements. The graph of requirements is resolved to identify all of the required installations.

4. In some cases, a new wheel-format file is required. In other cases, the project provides a wheel-formatted file. The **PIP** tool can use the `build_wheel` build-tool hook to combine the downloaded files into an installable form.

Some distributions include the source files, and may include data files or scripts that aren't trivially copied to the `site-packages` directory.

5. **PIP** then installs the wheel in the virtual environment's appropriate `site-packages` directory.

Possible build tools to build a package include `setuptools`, `build`, `hatch`, and `poetry`. All of these build tools can be used by **PIP**. They all make use of `pyproject.toml`.

## There's more...

In addition to the dependencies required for the project to work, additional dependencies are often based on other things we may do with the project. Common additional use cases are running tests, developing new features, and fixing bugs.

Tools for these additional use cases are optional dependencies. They are generally listed in a separate table, with sub-tables for each use case. For example, we might add the following table with two sub-tables to list the tools used for testing, and additional tools for more general development:

```
[project.optional-dependencies]
test = [
    "tox",
    "ruff",
    "mypy",
    "pytest",
    "pytest-cov"
]
dev = [
    "pip-tools",
```

```
    "sphinx"  
  ]
```

These additional lists permit someone to install the test suite to confirm the downloaded project passes all of its test cases. They also permit someone to download appropriate tools for maintaining the documentation and the detailed lists of dependencies.

Note that in these examples, none of the dependencies are named with a specific, pinned version. This is because we're going to use `pip-tools` to build a `requirements.txt` file from the information available in the `pyproject.toml` file. See *Using pip-tools to manage the requirements.txt file*.

Tools like **flit** and **twine** are often used to upload to a repository like PYPI. For enterprise developers, there may be an enterprise Python repository. These tools make use of additional tables in the `pyproject.toml` file.

The **flit** tool, for example, uses additional `[tool.flit.sdist]` and `[tool.flit.external-data]` tables to provide information required to perform an upload.

## See also

- See <https://python-semantic-release.readthedocs.io/en/latest/> for the Python Semantic Release tool that can modify version names based on Git commit messages.
- See *Using TOML for configuration files* in *Chapter 13* for more information on TOML files.
- See *Using pip-tools to manage the requirements.txt file* for our approach to refining the list of requirements into a line of pinned version numbers.
- The Hypermodern Python project has a template usable with the Cookie-Cutter tool to build a directory structure. See <https://github.com/cjolowicz>. This template relies on Poetry for managing dependencies and virtual environments.

- Chapter 17 contains the *The bare minimum: a README.rst file* recipe to address the README file in more depth.

## Using pip-tools to manage the requirements.txt file

Above, we noted that a project's dependencies have three levels of specificity:

**Generic** : Name-only dependencies

**Filtered** : With a very general constraint like `>= 4.0`

**Pinned** : With a specific version like `== 4.12.2`

How do we align these levels? One easy way is with the **pip-tools** package. This package includes the **pip-compile** tool, which will digest requirements, resolve dependencies, and create a derivative `requirements.txt` file with pinned version numbers.

A companion tool, **pip-sync**, can be used to ensure that the active virtual environment matches the `requirements.txt` file. This can be considerably faster than dropping and recreating a virtual environment.

### Getting ready

PIP-tools must be downloaded and installed. Generally, this is done with the following terminal command:

```
(ch17) % python -m pip install pip-tools
```

This assumes the virtual environment is active; in the example, it's named `ch17`. Using the `python -m pip` command ensures that we will use the `pip` command that goes with the currently active virtual environment.

The **pip-compile** tool will locate requirements in a `pyproject.toml` or `requirements.in` file. From this information, it builds a detailed `requirements.txt` file that can be used with `pip` or `pip-sync` to create the virtual environment.

## How to do it...

1. Make sure the dependencies are in the `pyproject.toml` file. In some cases, an old `requirements.txt` file may have been used to get started. It's a good idea to confirm that the information is in the `pyproject.toml` file because the `requirements.txt` file will be replaced.
2. The first time you do this, it helps to delete any old `requirements.txt` file not created by `pip-compile`.
3. To build the core `requirements.txt` file, run the `pip-compile` command:

```
(ch17) % pip-compile
```

This will locate the dependencies in the `pyproject.toml` file. It will then locate all of the transitive requirements and build a set of requirements with conflicts resolved.

It will both write a `requirements.txt` file and also display this file on the console.

4. To build a `requirements-test.txt` file, run the `pip-compile` command with the `--extra` option:

```
(ch17) % pip-compile --extra test -o requirements-test.txt
```

This creates a file with the optional dependencies from the `test = [...]` section of the `[project.optional-dependencies]` table.

5. To build a comprehensive `requirements-dev.txt` file that contains all of the extras, run the `pip-compile` command with the `--all-extras` option:

```
(ch17) % pip-compile --all-extras -o requirements-dev.txt
```

This creates a file with all of the optional dependencies in the `[project.optional-dependencies]` table.

6. When needed, use the `pip-sync` command to rebuild the current virtual environment to match changes to one of the `requirements.txt` files.

It's common practice to use this with the **tox** tool. In the `commands_pre` section of a test environment description, use `pip-sync requirements.txt` to make sure the test virtual environment is synchronized with the package versions in the `requirements.txt` file.

## How it works...

The `pip-compile` tool will look for information in three places:

- The `pyproject.toml` file.
- A `requirements.in` file, if present. This isn't needed, since the same information is in the `pyproject.toml` file.
- Any previously created `requirements.txt` file.

Using both the `pyproject.toml` and any previously created `requirements.txt` file allows the tool to properly reflect incremental changes. This means it can minimize the work required to analyze projects that haven't changed much. When starting a new project, it's sometimes helpful to delete the `requirements.txt` file entirely after making significant changes.

## There's more...

When making changes, there are two options that can help rebuild the `requirements.txt` details:

- The `--rebuild` option will clear caches and redo the analysis of dependencies.
- The `--upgrade some-package` option will look for upgrades for the named `some-package` package only. This prevents analysis of other packages that should be left alone. Multiple `--upgrade` options can be supplied to track multiple changes.

These two commands let us manage incremental change, upgrade `requirements.txt`, rebuild the virtual environment, and test with new versions of packages. This ensures the description of the environment matches the actual environment. We can share the project with confidence.



There are times when packages have conflicting requirements. Assume our project depends on project A and project T. It turns out project A also requires project T. Problems can arise when our project requires  $T \geq 10.11$  that's distinct from the version required by the A project, for example,  $T < 10.9$ . This can be challenging to resolve.

We can hope our project's limitation of  $T \geq 10.11$  is too specific; we can weaken the constraint and find a compatible version. In other cases, the requirements stated by project A might be too specific, and we need to consider making a change to the other project's code. Ideally, this is a proper issue and pull request, but it may require forking the project to offer distinct constraints. The worst case requires re-engineering our project to change the nature of the dependencies or — perhaps — stop using project A.

In some cases, there are obscure errors in `pyproject.toml` and the `pip-compile` tool reports an infuriatingly opaque error message:

```
(ch17) % pip-compile
Backend subprocess exited when trying to invoke get_requires_for_build_wheel
Failed to parse /Users/slott/Documents/Writing/Python/Python Cookbook
3e/src/ch17/pyproject.toml
```

This is a problem with the formatting of the `pyproject.toml` file.

One way to uncover the problem is to attempt to do an “editable” installation of the project in the current working directory. This will use the `pip install` command with the `-e .` option to use the current directory as the project to install.

It looks like this:

```
(ch17) % pip install -e .
```

This will report the specific error found in the `pyproject.toml` file. We can then repair the error and run `pip-compile` again.

## See also

- For more background on projects as a whole, see the [PYPY Sample Project](#).

- For information on licenses, see SPDX.
- The *Installing packages with a requirements.txt file* recipe describes using a file to drive **PIP** installations.
- See PEP-517 for more information on how a build system works.

## Using Anaconda and the conda tool

There are some limitations on the kinds of packages the **PIP** tool can install. The most notable limitation involves projects that involve extension modules written in a compiled language like Rust or C. The variations among platforms — including hardware and OS — can make it difficult to distribute all the required variants of the package’s binary files.

In a Linux environment, where compilers like GNU CC are readily available, a package with an extension module can include source code. The **PIP** tool can use the compiler to build the necessary binaries.

For macOS and Windows, additional tools are required to create binaries. Free compilers are not as readily available as they are in a Linux environment, presenting a potential problem.

Conda solves the problems with binaries by making a wide selection of pre-built binaries available in their repository. It also makes sure a compiler is available on the target platform for the cases where a pre-built binary isn’t available.

The **conda** tool is a virtual environment manager and package installer. It fulfills the same use cases as **PIP**, combined with **venv** and `pip-tools`. This includes builds of packages that include binaries, often used for high-performance numeric applications. The command-line interface for **conda** is the same on all platforms, permitting simpler, more consistent documentation.

The Anaconda package index is curated by the Anaconda company. Check out their website, <https://anaconda.com>, for prices and fees. The packages provided have been integrated and tested. This testing takes time, and the official Anaconda distribution can lag behind

what's available in PYPI. Further, it's a subset of what's available on PYPI because it tends to focus on data analytics and data science.

A separate package index, conda-forge (<https://conda-forge.org>) is community based. This channel contains packages that more closely mirror what's in PYPI. In many cases, we'll install packages from this channel because we want something new, or we want something outside the curated subset available from Anaconda.

## Getting ready

There are two ways to get the **conda** tool:

- Download and install the full Anaconda distribution. This is a large download, anywhere from 900 MB for Windows to over 1,000 MB for more Linux distributions.
- Download and install miniconda and use it to install only the desired packages. This is a much smaller download, generally about 100 MB.

For the full Anaconda install, see <https://www.anaconda.com/download>. There are two varieties of the installer:

**Graphical** : These installers use the OS interactive tools to support some configuration.

**Command-line** : These installers are sophisticated shell archives that run in a terminal window. They provide the same options for installation as the graphical installer. There's more typing and less pointing and clicking.

For a miniconda install, see <https://docs.conda.io/projects/miniconda/en/latest/index.html>. Each OS has slightly different kinds of installers:

**Windows** : The installer is an executable program that uses the Windows installer.

**macOS** : There are PKG images that can be downloaded and double-clicked to use a macOS UI. There are *also* command-line images that can be executed from the terminal window.

**Linux** : These are shell-archive files that are started from a terminal window.

While there are a lot of choices here, we recommend using a command-line installer for

Miniconda.

See the Miniconda page for recommended shell commands to use the **curl** program to fetch the image and then perform the installation.

Once the **conda** tool has been installed, it can be used to create and populate virtual environments. Note that the **conda** tool creates a base virtual environment. When **conda** is installed, the (base) environment should be shown as part of terminal window prompts. This serves as a visual cue that no other environment has been activated. It may help to exit and restart all terminal windows to be sure **conda** is working.

## How to do it...

It's essential that the conda tool is installed. See the *Getting ready* section of this recipe for advice on installing conda.

1. Use the `conda create` command to create a new virtual environment:

```
% conda create -n cookbook3 python=3.12
```

Note the commands are the same on all operating systems.

The virtual environment's files are kept outside the project directory. For macOS, there will be a `~/miniconda3/envs` directory that has all of the virtual environment files.

2. Use the `conda activate` command to activate this new virtual environment:

```
(base) % conda activate cookbook3
```

3. Use the `conda install` command to install a list of packages in the virtual environment. Conda has its own conflict resolver that's separate from the one used by the **PIP** tool or **pip-compile**. While we can use the `requirements.txt` file, we don't really need all of those details. It's often easier to provide the package name information as shown in this command:

```
(cookbook3) % conda install pydantic beautifulsoup4 jupyterlab  
matplotlib pytest memray
```

4. To create a shareable definition of the current virtual environment, use the `conda env export` command:

```
(cookbook3) % conda env export >environment.yml
```

This uses the shell redirect feature to save the exported information into a YAML-formatted file that lists all of the requirements. This file can be used by `conda env create` to recreate this environment.

## How it works...

The virtual environment created by **conda** has the proper `PATH` environment variable set to point to a specific Python binary. The standard library packages and site-specific packages are located in nearby directories.

This parallels the virtual environments created by the built-in `venv` module. It follows the rules of PEP-405, which defines the rules for virtual environments.

In order to work consistently, the `conda` command must be visible. This means a base `conda` installation must **also** be named in the system `PATH` environment variable. This is a crucial step in using **conda**. The Windows installer has the option to either update the system path or to create special command windows in which the necessary path setting has been made. Similarly, the macOS installer requires an extra step to make the `conda` command available to the **zsh** shell.

The Anaconda repositories may have pre-built binaries, which can be downloaded and used by the `conda` tool. In cases where binaries aren't available, the `conda` tool will download the source and build the binaries as needed.

## There's more...

One of the most common use cases is upgrading to the latest releases of packages. This is done with the `conda update` command:

```
(cookbook3) % conda update pydantic
```

This will look for the version of the package available in the various channels being searched. It will compare the available version with what's currently installed in the active virtual environment.

To collaborate politely with tools like `tox` for testing, it helps to use the `pip freeze` command to create a `requirements.txt` file. By default, `tox` uses `pip` to build virtual environments. The **PIP** tool will not overwrite packages installed by **conda**, allowing them to coexist peacefully.

Another choice is to use the `tox-conda` plug-in to allow the `tox` tool to use `conda` to create and manage virtual environments. See the `tox-conda` repository at <https://github.com/tox-dev/tox-conda>.

Not all libraries and packages are part of the Anaconda-supported, curated library. In many cases, we'll need to step outside Anaconda and use the community `conda-forge` channel in addition to the Anaconda channel.

We'll often need to use a command like the following to use the `conda-forge` channel:

```
(cookbook3) % conda install --channel=conda-forge tox
```

We can also use `pip` to add packages to a `conda` environment. It's rarely needed, but it does work nicely.

## See also

- See *Creating environments using the built-in venv* for more information on virtual environments.

- See <https://www.anaconda.com/download> for the full Anaconda installation.
- See <https://docs.conda.io/projects/miniconda/en/latest/index.html> for the Miniconda installation.

## Using the poetry tool

The combination of the `venv`, `pip`, and `pip-tools` packages allows us to create virtual environments and populate them with packages from the PYPI package index.

The **poetry** tool is a virtual environment manager and package installer combined into a single tool. It fulfills the same use cases as **PIP**, combined with **venv** and `pip-tools`. It also fulfills the same use cases as **conda**. The CLI is the same on all platforms, permitting simpler, more consistent documentation for developers using **Poetry** to manage environments.

There are some minor differences in the way Poetry enables a virtual environment. Rather than tweaking the current shell's environment variables, it launches a sub-shell. The sub-shell has the required virtual environment settings.

## Getting ready

Note that the **Poetry** tool must be installed in its own virtual environment, separate from any project managed by Poetry. This is best done by using the Poetry installer. This involves OS-specific commands to download and execute the installer. The installer is written in Python, which makes the task somewhat more consistent across OSs.

See <https://python-poetry.org/docs> for details.

There are two steps:

- Download from <https://install.python-poetry.org>.
- Execute the downloaded Python script.

The recommended commands vary slightly between operating systems:

- **macOS, Linux, and Windows Subsystem for Linux:** The `curl` command is generally available for doing the download. This command can be used:

```
% curl -sSL https://install.python-poetry.org | python3 -
```

After this, the follow-on step will update the system PATH environment variable. The output from the installation will provide the location to use. These two examples are for macOS, where the file is in `~/ .local/bin`.

Edit the `~/ .zshrc` file to add the following line:

```
export PATH="~/ .local/bin:$PATH"
```

As an alternative, it is possible to define an alias for the location of the poetry command. This is often `~/ .local/bin/poetry`.

- **Windows Powershell:** The `Invoke-WebRequest` Powershell command performs the download. The Python launcher, `py`, runs the appropriate version of Python:

```
PS C:\> (Invoke-WebRequest -Uri https://install.python-poetry.org  
-UseBasicParsing).Content | py -
```

The script for poetry is placed in the `AppData\Roaming\Python\Scripts` sub-directory. Either add this to the PATH environment variable or use the path explicitly, for example: `AppData\Roaming\Python\Scripts\poetry --version`.

Changing the current working directory with `chdir` means explicitly referring to your home directory's `AppData` sub-directory.

Once the **poetry** tool has been installed, it can be used to create and populate virtual environments.

## How to do it...

1. Use the `poetry new` command to create a new project directory. This will not only create a virtual environment; it will also create a directory structure and create a `pyproject.toml` file:



```
% poetry new recipe_05
```

The virtual environment's files are kept outside the project directory. For macOS, there will be a `~/Library/Caches/pypoetry` directory that has all of the virtual environment files.

Note that poetry tries to cooperate with other virtual environment tools. This means you can use a `venv activate` command to set the environment variables.

2. Rather than activate the environment within a shell's environment, it's often easier to start a sub-shell with the appropriate environment settings.

Use the `poetry shell` command to start a shell that has the virtual environment activated:

```
% poetry shell
```

Use the shell's `exit` command to terminate this sub-shell and return to the previous environment.

3. Use the `poetry add` command to add packages to the environment. This will both update the `pyproject.toml` file and install the packages:

```
%(recipe-05-py3.11) poetry add pydantic beautifulsoup4 jupyterlab  
matplotlib pytest memray
```

This also creates a `poetry.lock` file that defines the exact versions of each dependency.

## How it works...

The virtual environment created by **poetry** has the proper `PATH` environment variable set to point to a specific Python binary. The standard library packages and site-specific packages are located in nearby directories. Poetry properly leverages the information in

the `pyproject.toml` file, reducing the number of additional files required to define the working environment.

In order to work consistently, the `poetry` command must be visible. This means either adding the poetry location to the system `PATH` environment variable or using an alias. This is a crucial step in using **poetry**.

The alternative to an alias is what is shown in the recipe, using `~/local/bin/poetry` explicitly. This is less than optimal, but it makes the relationship between the current working virtual environment and the poetry command more clear.

## There's more...

One of the most common use cases for an environment and tool like Poetry is upgrading to the latest releases of packages. This is done with the `poetry update` command. A specific list of packages can be provided. With no parameters, all packages are examined.

Here's an example:

```
% (recipe-05-py3.11) poetry update pydantic
```

This will look for the version of the `pydantic` package available in the various channels being searched and compare that version with what's currently installed in the active virtual environment. This will also update the `poetry.lock` file after installing the updates.

To collaborate politely with tools like `tox` for testing, some additional options are required in the `tox.ini` file. One easy-to-use approach is to skip the default install procedure that `tox` uses and use `poetry` commands to run commands in a poetry-managed environment.

Here's a suggestion of how to use `poetry` with `tox`:

```
[tox]
isolated_build = true

[testenv]
skip_install = true
```

```
allowlist_externals = poetry
commands_pre =
    poetry install
commands =
    poetry run pytest tests/ --import-mode importlib
```

Using `poetry run` means the command will be executed in the virtual environment. This makes it possible to use `tox` to define multiple environments, and rely on Poetry to assemble the various environments for testing purposes.

## See also

- See <https://python-poetry.org/docs> for details on Poetry.
- See <https://tox.wiki/en/4.15.1/> for details on Tox.

## Coping with changes in dependencies

As we noted in the *Installing packages with a requirements.txt file*, all packages on which an application is built are in a constant state of flux. Each project has a distinct tempo for updates. To manage the constant change, it's important for people developing applications to track dependencies carefully.

A common complaint about Python is sometimes summarized as dependency hell. This summarizes the work required to track and test with new dependencies, some of which may be in conflict. This work to manage change is essential; it's the minimum required to maintain a viable product. Instead of adding features, it preserves functionality in a world of constant change.

There are two common cases where upgrades turn into more than simply installing and testing with upgraded packages:

- Changes that break our application in some way
- Incompatibilities among packages our application depends on

In the first case, our software fails to work. In the second case, we can't even build a virtual

environment in which to test. This second case is often the most frustrating.

## Getting ready

We'll consider a hypothetical project, the `applepie` application. This application has several dependencies:

- A single module from the `apple` project's package, named `apple.granny_smith`
- Some classes from the `pie_filling` project
- Several classes from the `pastry_crust` project
- An oven implementation

The general dependencies are named in the `pyproject.toml` file as a list of projects. We can imagine the detailed `requirements.txt` (or `poetry.lock`) file looks like the following:

```
apple == 2.7.18
pie_filling = 3.1.4
pastry_crust >= 4.2
oven == 0.9.1a
```

Using this framework for an application, we'll look at what changes we need to make when we see changes in the dependencies. One change will remove needed functionality; the other change will be an incompatibility between releases of the `pie_filling` and `pastry_crust` projects. We'll need to make appropriate changes in our application based on these changes occurring in the larger Python ecosystem.

## How to do it...

We'll decompose this into two sub-recipes: one for a dependency that leads to a test failure, and the second for dependencies that are incompatible. We need to adjust our project so it continues to work in the presence of ongoing change.

### A change caused a test failure

To continue our example, the `oven` tool has had a significant change to the API. The new release, `oven` version 1.0, doesn't have the same interface version 0.9.1a had. The

consequence is a failure in our code:

1. Clarify what the failure is and what caused it. Ask “why?” enough times to identify the root cause. There are several aspects of the failure that may need to be explored.

Be sure to understand the top layer of the problem: how the failure manifested itself. Ideally, a unit test failed. Another good avenue for detection is to use a tool like **mypy** or **ruff** to identify a potential for failure. Less helpful is an acceptance or system test that failed even though unit tests all passed. Perhaps the worst case is failure after deployment, in the hands of a customer.

Also, be sure to understand what changed. It’s common to introduce a number of version upgrades all at once. It may be necessary to reverse those changes and then upgrade each required package one at a time to identify the source of the failure.

2. Failures after a release often lead to problem reports in issue-tracking tools. Update any issue-tracking software with the root cause analysis.

Failures during testing should also lead to an internal report of a problem. The repair may require extensive rework, and it is generally helpful to track the reason for the rework.

3. Choose among the four kinds of fixes that are possible:
  - (a) Your code needs to be fixed. The change to oven version 1.0 is a clear improvement.
  - (b) The change to oven introduced a bug, and you need to report the problem to the maintainers of oven or fix their code.
  - (c) Revise the dependencies to pin oven version 0.9.1a in `pyproject.toml` to prevent upgrades.
  - (d) Looking closely at the oven project, it may be clear it is no longer a good fit with this project, and it needs to be replaced.

These are not exclusive choices. In some cases, multiple paths will be followed.

There may be a pervasive change to our project required to accommodate the changes to oven 1.0. This may be an opportunity to refactor our code to more carefully isolate this dependency to simplify making changes in the future.

When a project seems to have a bug, we have two choices: we can report the issue and hope it's fixed, or we can clone the repository, make a fix, and submit a pull request to pull our changes into the next release. The benefit of open source is the reduced cost to begin a project. Ongoing maintenance, however, is an eternal feature of a landscape that thrives on innovation. While we benefit from other's work in open source, we also need to contribute by proposing fixes.

In some cases, we will pin a specific version while we decide what to do about a dependency. We may pin an old version while choosing among alternatives and rewriting our project to replace the old oven with the new `convection_cooker`.

In effect, code that breaks due to an upgrade is a bug fix. It may be a bug fix for a project we require. More often, the fix is applied in our project to make use of a change in other projects. Managing changes to our application is the price we pay for innovation in the broad ecosystem of Python packages.

### **A changed dependency is incompatible with another dependency**

In this case, the `pastry_crust` version 4.3 uses `sugar` version 2.0. Sadly, the `pie_filling` version 3.1.4 uses the older `sugar` version 1.8.

1. Identify the root cause of the conflict, to the extent possible. Trying to discern why the `pie_filling` project team has not upgraded to `sugar` version 2.0 may be very difficult. A common observation is a lack of activity in the `pie_filling` project; but without knowing the principal contributors well, it's difficult to ask why they aren't making changes.

Be perfectly clear in identifying what changed. It's common to introduce a number of version upgrades all at once. It may be necessary to reverse those changes and then upgrade each required package one at a time to identify the source of the failure.

These conflicts are not in a direct dependency, but in an indirect dependency.

We value the idea of encapsulation and abstraction right up until we observe conflicting requirements that are encapsulated by a project. When these conflicts appear, the obscurity of encapsulation becomes a burden.

2. Document the conflict as an issue in issue-tracking software. Be sure to provide links to the conflicting projects and their issue trackers. The resolution may involve extended side-bar conversations with other projects to understand the nature of the conflicting requirements. It helps to keep notes on these conversations.
3. Choose among the four kinds of fixes that are possible:
  - (a) It's unlikely any small change to your code will resolve the problem. The change required is replacing the `pie_filling` requirement with something else and making sweeping changes.
  - (b) It's possible that changes to the `pie_filling` project may correct the problem. This may involve a great deal of work on someone else's project.
  - (c) Revise the dependencies to pin `pastry_crust` version 4.2 in `pyproject.toml` to prevent upgrades.
  - (d) Looking closely at the `pie_filling` project, it may be clear it is no longer a good fit for this project and needs to be replaced. This is a sweeping change to the project.

These choices are not exclusive. In some cases, multiple paths will be followed. Perhaps the most popular choice is pinning the version that prevents the compatibility problem.

The rework to the `pie_filling` project can involve two activities: we can report the issue and hope they fix it, or we can clone their repository, make a fix, and submit a pull request to pull our changes into their next release. This kind of ongoing maintenance of open source software created by others is an eternal feature of a landscape that thrives on innovation.

Incompatibilities among the required supporting projects is an architectural problem. It's rarely solved quickly. An important lesson learned from this kind of problem is that all architectural decisions need to be revocable: any choice needs to have an alternative, and the software needs to be written so that either alternative can be exercised.

### **How it works...**

The essential step here is doing root cause analysis: asking “why?” something fails to pass tests when upgrades are attempted.

For example, our `applepie` project's dependencies may pin `oven` version 0.9.1a because version 1.0 introduced a failure. The pinned version may be appropriate for a few hours until the `oven` project fixes a bug, or it could remain in place for a much longer period of time. Our project may go through several releases before the problem with `oven` 1.0 is finally fixed by release 1.1.

It requires some discipline to review the requirements and make sure any pinned versions still need to be pinned.

### **There's more...**

One source of frustration with dependency hell is the lack of time budgeted for finding and fixing dependency problems. In an enterprise context, this is an acute problem because project sponsors and managers are often narrowly focused on budget and the time required to implement new features. Time to resolve dependency issues is rarely part of the budget because these problems are so difficult to anticipate.

A terrible situation can arise where fixing dependency problems is counted against a team's velocity metric. This can happen when there are no “story points” assigned to upgrading the dependencies and rerunning the test suite. In this case, the organization has created a perverse incentive to pin versions forever, without following the progress of other projects.

It's imperative to have a periodic task to review each and every requirement. This task involves seeing what changes have been made and what may have been deprecated since the last review. This may lead to modifying project version constraints. For example,



we may be able to relax the requirement from a strict `oven==0.9.1a` to a more lenient `oven!=1.0`.



A periodic task to review all requirements is an essential ingredient in managing change and innovation.

Look for updates as well as deprecations.

Allocate time to run tests with new versions and report the bugs discovered.

## See also

- See *Using pip-tools to manage the requirements.txt file* for a very good dependency resolver.
- See *Using Anaconda and the conda tool* for an approach to using the conda repository of curated, compatible software releases.

## Join our community Discord space

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

<https://packt.link/dHrHU>



# 17

## Documentation and Style

We've combined two topics into a single chapter. They're often looked at as “extras” in a project's life-cycle. The deliverable code is often considered to be the most important thing.

Some developers will try to argue that test cases and documentation aren't the code with which the user interacts, and therefore, these additional pieces aren't as important as the code.

This is false.

While it is true the users don't interact *directly* with test cases, the presence of test cases is what gives people the confidence to use the code. Without the test cases, there's no evidence the code does anything useful at all.

Documentation provides essential information that can – eventually – be extracted from the code. A project with a prominent docs folder is considerably more trustworthy than a project that lacks documentation.

Code “style” is a relatively minor point. However, it’s also part of the static assessment of code, including type hint analysis, quality metrics, and more specific “lint” checks. There are many software implementation practices that can be confusing, or rely on poorly documented language or library features. These are the “fuzzy edges” of the software. A lint tool acts like a lint trap in an electric clothes dryer, catching highly flammable lint so it doesn’t clog the vents, preventing a fire. Removing software fuzz can prevent bugs. In some cases, it may only reduce the possibility of problems.

We’ll consider linting and formatting to be quality assurance steps as important as test cases and static type checking.

In this chapter, we’ll look at the following recipes for creating useful documentation:

- *The bare minimum: a README.rst file*
- *Installing Sphinx and creating documentation*
- *Using Sphinx autodoc to create the API reference*
- *Identifying other CI/CD tools in pyproject.toml*
- *Using tox to run comprehensive quality checks*

## The bare minimum: a README.rst file

In *Chapter 16*, the *Creating a pyproject.toml file* recipe described how to create a `pyproject.toml` file with a reference to a README file.

For the purposes of that recipe, we suggested the file is a summary of how to install and use the project.

We also noted there are six essential questions: “who?”, “what?”, “why?”, “when?”, “where?”, and “how?” that can help in writing a short lede paragraph to describe the project.

There are two common challenges when writing a README file:

- Writing too much

- Writing too little

A good package will include a separate docs folder with detailed documentation. The README file is only an introduction and a roadmap through the project's various files and folders. In many cases, where concrete examples are called for, it's important to very judiciously repeat information provided elsewhere in the documentation to avoid contradictions.

A project without a README is visibly deficient. Locating good examples can help provide guidance on what is needed. Some developers feel the code should somehow speak for itself, and serve as documentation. The code, unfortunately, only really answers the “how?” question effectively. Questions about who the users are and how the software should be deployed require statements that must exist outside the software.

In this recipe, we'll dive into what makes a useful README file.

## Getting ready

A preliminary step is to choose the markup that will be used for the README file. There are three common choices:

- Plain text
- Markdown
- ReStructured Text (RST)

The advantage of plain text is the simplicity of avoiding additional formatting elements. The disadvantage is the lack of typographic hints like font changes to provide important context.

The Markdown markup has the advantage of having a small set of elements. These overlap with a number of common practices in writing natural-language text in a file. Showing indented text in a distinct font for example, and treating a paragraph starting with a punctuation mark and a space as a bulleted list item.

Using RST provides a comprehensive set of elements that covers a wide variety of typo-

graphic details. This is the preferred markup language for Python's internal documentation projects. In some cases, the docs folder may be built with RST, but the README file may be in plain text.

The choice is free of long-term consequences, since this file is essentially isolated from the rest of the project's documentation. When in doubt, it can help to toss a three-sided coin to make the choice. The file is not large, and making changes is relatively easy.

## How to do it...

1. Write an **introduction**, or lede, with information about who would use this package, why they would use it, and what it does. In some cases, it may be helpful to state when and where an application is used; this may be needed to clarify client-vs.-server hosting or admin-vs.-user roles. Keep this short; details will follow. This is sometimes called the "elevator pitch" because you can state it during an elevator ride in an office building.
2. Summarize the important **features** of the software. This is often a bulleted list. It may include screen grabs to show a user interface, if that's an important feature. It's important to summarize and not overwrite all the details here. The details should be in the separate docs folder.
3. Detail any **requirements** or dependencies. This may include hardware and operating system if that's important. It must include any Python version constraints. This may repeat the dependencies in the `pyproject.toml` in the case where a library or package is a plug-in or extension to another package or module.
4. Provide **installation** instructions. Often this is the `python -m pip` command required to download and install the package. If there are optional features, these will be summarized here, also.
5. Provide an introduction to **usage** or **operation**. This is not the user guide, but it is what most people will see first, and the usage section should provide a tidy, clear, working example.

There are two distinct approaches to writing this, depending on what the software is:

- For modules and packages that will be imported, a doctest example is ideal. The README can then be tested to confirm the example really is correct and works as expected.
- For applications, the usage may include step-by-step instructions for a common use case, possibly with screen-grab images.

For some simple applications, this may be the entire user guide. Generally, it's only going to show a single, simple use case.

6. Provide the type of **license** and a link.
7. Provide a section on **contributing** to the project. This may be a link to a separate contributor guide document, or it may be a short description of how to make changes and submit a pull request.

In some cases, information about the integration, testing, and deployment may be helpful here. For complicated applications, the build process may involve steps that aren't obvious.

This should also include information about documenting issues and making feature requests.

8. It's also polite to include **credits** or **acknowledgments** for the work of other contributors. This may include information about backers and sponsors.

## How it works...

The key ingredient of a README file is concrete examples of commands and features that actually work. It shows what the software is, how to install it, how to use it, and how to maintain it.

Examining READMEs from popular repositories reveals some common features. There's a Make a README web site that can help create a file in case additional guidance is required.

While there will be additional documentation elsewhere, the README is the first thing most people read. In some cases, it's also the last thing they read. Therefore, it must be clear what the software is and how it will be used.

## There's more...

A common feature of README files is badges showing the general health of the project. There are several sources for these graphical summaries.

The <https://shields.io> site provides a number of static and dynamic badges. A dynamic badge can interrogate services like PyPI or GitHub to post the current status.

In Markdown, something like the following might be used to build a badge.

```
! [release] (https://img.shields.io/pypi/v/<project>.svg)
```

This would show a small graphic badge with pypi on the left, and the current PyPI release number on the right side.



Figure 17.1: Badge Example

The badge can also be a link, and can provide more detailed information.

```
!! [release] (https://img.shields.io/pypi/v/<name>.svg)]  
(https://pypi.org/project/<name>)
```

## See also

- The C4 model offers additional help on how to describe software. See <https://c4model.com>.
- See the <https://github.com/matiassingers/awesome-readme> project on GitHub for good examples.

- Cookie-cutter templates are available with a search for “cookiecutter” repositories: GitHub search. This is quite a large list with 1000’s of cookie-cutter templates:  
<https://github.com/search?q=cookiecutter&type=Repositories&type=repositories>.

## Installing Sphinx and creating documentation

A README file is a summary of the software touching on a few key points. Proper documentation often parallels the important topics of the README, but in more depth.

Important adjuncts to the essential “how-to” guides include two important topics:

- What the software does. This is often a detailed description of the observable features.
- How the software works, showing the implementation concepts.

The C4 model suggests four tiers of abstraction in the description:

1. The context in which an application is used.
2. The containers in which the software runs.
3. Component diagrams showing the architecture of the software.
4. Code diagrams showing the implementation details.

This organization offers the necessary focus for documentation.

We’ll write in RST or Markdown format. Tools like Sphinx then build output documents in a variety of target formats.

We often want to provide an API document with the implementation details, extracted from the docstrings present in our modules, classes, methods, and functions. In *Chapter 2*, the *Including descriptions and documentation* recipe described how to add docstrings to various Python structures. The Sphinx tool autodoc extension extracts the docstrings to produce detailed API documentation.

Further, the Sphinx tool makes it easy to decompose the documentation source into smaller files that are easier to edit and manage.



## Getting ready

We'll need to download and install the Sphinx tool. Generally, this is done with the following terminal command:

```
(cookbook3) % python -m pip install sphinx
```

Using the `python -m pip` command ensures that we will use the `pip` command that goes with the currently active virtual environment.

There are several built-in themes, plus numerous third-party themes. See <https://sphinx-themes.org> for dozens of additional themes.

## How to do it...

1. Make sure the project directory has at least the following sub-directories:
  - The source. This may use the package's name or it may be called `src`.
  - The tests, often called `tests`.
  - The documentation, often called `docs`.
2. Change the working directory to the `docs` directory with a `cd` or `chdir` command.

From there, run the `sphinx-quickstart` command.

```
(cookbook3) recipe_02 % cd docs
(cookbook3) docs % sphinx-quickstart
Welcome to the Sphinx 7.2.6 quickstart utility.
```

```
Please enter values for the following settings (just press Enter to
accept a default value, if one is given in brackets).
```

This will embark on an interactive conversation to gather details about your project and seed your `docs` folder with the files required to run Sphinx.

The result will be several directories and files for the documentation:

- `conf.py` has the project configuration.

- `index.rst` is the root document.
  - Makefile can be used in all other environments to build the documentation.
  - A `make.bat` for use in a Windows environment may also be present.
3. Edit the `index.rst` file to write an initial summary. This might be copied from the README file.
  4. Run the `make html` command to build the initial documentation. This is a shell command, run in the terminal window. Make sure the current working directory is the docs directory.

## How it works...

The Sphinx tool starts processing by reading the root document. The `root_doc` configuration parameter names `index`. The `source_suffix` configuration parameter sets the suffix for this file to `.rst`.

Generally, this file will name the other files in the documentation. The `.. toctree::` directive is used to specify the other files in the documentation.

Let's say we need to write several sections for installation, usage, maintenance, design, and API reference. The `index.rst` will have this as the primary content.

```
The recipe_02 project
=====

The recipe_02 project is an example of Sphinx documentation.

.. toctree::
   :maxdepth: 2
   :caption: Contents:

   installation
   usage
   maintenance
   design
   api
```

The file created by the `sphinx-quickstart` tool will create a preamble in front of the above example. We've omitted them from this example, since there's no good reason to change them.

The `toctree` directive has two parameters, `:maxdepth: 2` and `:caption: Contents:`. These tailor the behavior of the directive's output.

Note that content inside a directive must be indented consistently. Often the initial file will have an indent of three spaces. Some editors work with a default indent of four spaces, so some changes to settings can be helpful.

Each of the names in the `toctree` body refers to a file with the configured suffix, in our case, `.rst`.

The `installation.rst`, `usage.rst`, `maintenance.rst`, `design.rst`, and `api.rst` documents must start with a proper RST title line. The initial content can come from notes or the README. For more help on RST, see *Writing better docstrings with RST markup* in *Chapter 2*.

The content of the `api.rst` document will make use of the `autodoc` extension. We'll look at this in *Using Sphinx autodoc to create the API reference*.

## There's more...

There are many useful extensions for Sphinx. We'll look at including a to-do list.

Enable an extension by adding `sphinx.ext.todo` to the list of extensions in the `conf.py` configuration file:

```
extensions = [  
    'sphinx.ext.autodoc',  
    'sphinx.ext.todo',  
]
```

This will introduce two new directives to the available markup:

- The `.. todo::` directive creates a to-do item.

- The `.. todoclist::` directive will be replaced by the content of all the todo items.

The todo items produce no other output. It's easy to find them; some IDEs will automatically scan the files for the letters todo and keep this as a list of things for a developer to address.

Editing the `conf.py` configuration file to add the following line will enable the `.. todoclist::` directive to include the items in the documentation:

```
todo_include_todos = True
```

With this, the todo items are elevated from personal notes to public items in the documentation.

Sphinx comes with a number of themes that define the styles to use. The default theme is called `alabaster`. Changing to one of the other built-in themes can be done with the `html_theme` setting in the `conf.py` configuration file.

Change to the `sphinxdoc` theme with a setting like the following:

```
html_theme = 'sphinxdoc'
```

Many themes have further customizations possible. Providing an `html_theme_options` dictionary tailors the theme.

## See also

- See Sphinx <https://www.sphinx-doc.org/en/master/> for details on the Sphinx project.
- See <https://sphinx-themes.org> for some additional Sphinx themes.
- See *Using Sphinx autodoc to create the API reference* to see how to build API documentation from code.

## Using Sphinx autodoc to create the API reference

One of the huge strengths of Sphinx is being able to generate the API documentation using the autodoc extension. A series of commands can extract the docstrings from modules, classes, functions, and methods. Options are available to fine-tune exactly what members are included or excluded.

We'll refer back to *Chapter 7, the Extending a built-in collection – a list that does statistics* recipe. In there is this `StatsList` class:

```
class StatsList(list[float]):
    def sum(self) -> float:
        return sum(v for v in self)
    def size(self) -> float:
        return sum(1 for v in self)
    def mean(self) -> float:
        return self.sum() / self.size()
    # etc...
```

Because this inherits methods from the `list` class, there are a large number of available methods. By default, only those methods with docstrings, excluding private methods (those with a leading `_`), will be examined and included in the documentation. We have a number of other choices of which methods to include in the documentation:

- We can name specific methods, and only those will be documented.
- We can ask to include methods without docstrings; the signature will be shown.
- We can ask for private members (those with a leading `_`).
- We can ask for special members (those with double leading `__`).
- We can ask for inherited members to see members from the superclasses.

We'll start by putting docstrings into this class definition. Once that task is finished, we can include the necessary configuration and directives in the documentation directory.

## Getting ready

The first step is to add docstrings to the module, the class, and the methods. In some cases, partial docstrings are in place and the task is to expand them to be more complete.

We might, for example, have already added the following kinds of comments:

```
class StatsList(list[float]):
    """
    A list of float (or int) values that computes some essential statistics.

    >>> x = StatsList([1, 2, 3, 4])
    >>> x.mean()
    2.5
    """

    def sum(self) -> float:
        """
        Sum of items in the list.
        """
        return sum(v for v in self)
```

This started to provide some useful documentation in an API reference. The class docstring has a doctest example to show how it works. The method has a docstring with a summary of what it does.

We need to extend this to add details on parameters, return values, and exceptions raised. This is done with additional syntax called a “field list”.

Each item in a field list has a name and a body. The general syntax is this:

```
:name: body
```

The Sphinx tool defines a large number of field list names that are used to format documentation for a function or method. Here are a few of the most useful ones:

- `:param name:` Description of a parameter
- `:key name:` Description of a keyword parameter

- `:raises exception:` Description of the reason for the exception
- `:var name:` Details for internal variables of a class that are exposed
- `:returns:` The return value from a method or function

These permit the writing of detailed descriptions of methods.

## How to do it...

1. Edit the docstrings to include details.

We might, for example, want to expand on the method definition as follows:

```
def sum(self) -> float:
    """
    Computes the sum of items in the list.

    :returns: sum of the items.
    """
    return sum(v for v in self)
```

For a function as simple as this, the `:returns:` part of the field list seems redundant.

2. Edit the `conf.py` file to add the `'sphinx.ext.autodoc'` string to the list extensions:

```
extensions = [
    'sphinx.ext.autodoc'
]
```

3. Add the `src` directory to `sys.path` in the `conf.py` configuration file:

```
import sys
sys.path.append('../src')
```

This works because the `conf.py` file is a Python module, and can execute **any** Python statements. Adding the `src` directory to the path means Sphinx can import the module.

4. Put the following directive in the `api.rst` document.

```
.. automodule:: stats
   :undoc-members:
```

This will import the module, extract the docstring, and then attempt to create documentation for all of the members, including those that do not – as yet – have docstrings.

The Sphinx quickstart created a `Makefile` to help build the final PDF or HTML file from the source material; see the *Installing Sphinx and creating documentation* recipe for more information. Run the `make html` shell command in the `docs` directory, The build directory will have a static website with the project's documentation.

## How it works...

The examination of docstrings to extract the detailed documentation starts with an elegantly clever feature of the Python language: the documentation string. The rules of RST markup continue a path toward elegant-looking documentation. The resulting Sphinx page looks like this:



## recipe\_03

### Navigation

Contents:

API

- StatsList

### Quick search



## API

Defines classes that compute summary statistics from larger data structures.

- **StatsList**
- Others are possible

```
class stats.StatsList(iterable=(), /)
```

A list of float (or int) values that computes some essential statistics.

```
>>> x = StatsList([1, 2, 3, 4])
>>> x.mean()
2.5
```

**mean()** → float

The mean of values in the list.

**size()** → float

The size of the list. This is often the **len()** but a subclass may define a filter to exclude values.

**stddev()** → float

Standard deviation of the list.

**sum()** → float

Sum of items in the list.

**sum2()** → float

Sum of squares of items in the list.

**variance()** → float

Variance of items in the list.

$$\sigma^2 = \frac{\sum x^2 - \frac{(\sum x)^2}{N}}{N - 1}$$

©2024, S.Lott. | Powered by Sphinx 7.2.6 & Alabaster 0.7.16 | Page source

Figure 17.2: Sphinx Output Example

Note the `variance()` method includes a `.. math::` directive with details on how the computation is performed. This requires some care because the `LaTeXmath` syntax involves a fair number of `\` characters.

There are two ways to deal with `LaTeXmath` in docstrings:

- Use a “raw” string literal and single `\`:

```
r"""
A docstring with :math:`\alpha \times \beta`
"""
```

This means that no other escaped characters can be used. This may prevent using

Unicode characters, for example.

- Use a `\` to escape the special meaning of `\`:

```
"""
A docstring with :math:`\alpha \times \beta`
"""
```

This permits including Unicode escape sequences like `\N{Black Spade Suit}` in the docstring.

In both cases, note that RST uses back-ticks ``` around the content that has a role, like `:math:``.

## There's more...

A cross-reference to another class, module, or method uses `:role:`value`` syntax. The `:role:` portion is the specific kind of reference to help distinguish modules, classes, and functions. The value is a name that has a definition directive somewhere in the documentation.

A cross-reference will generate appropriately formatted text with a hypertext link to the the definition for the name.

Here's an example:

```
Uses the :py:class:`~stats.StatsList` class in the :py:mod:`~stats` module.
```

The `:py:class:`~stats.StatsList`` has the role of `:py:class:` to create a class reference to the `StatsList` class definition. The use of `~` in the name means that only the last level of the name will be shown. The full path is required to generate a correct reference to the class. The `:py:mod:`~stats`` reference is a role of `:py:mod:` and names the `stats` module.

## See also

- See *Installing Sphinx and creating documentation* for more information on Sphinx.

- See *Chapter 7*, the *Extending a built-in collection – a list that does statistics* recipe for the example this is built around.
- See *Chapter 2*, the *Including descriptions and documentation* recipe for more information on docstrings.

## Identifying other CI/CD tools in `pyproject.toml`

The terms **Continuous Integration (CI)** and **Continuous Deployment (CD)** are often used to describe the process of publishing a Python package for use by others. The idea of doing a number of quality checks for integration and deployment is central to good software engineering. Running a test suite is one of many ways to affirm that software is fit for the intended purpose.

Additional tools might include `memray`, which is used to check the use of memory resources. A tool like `ruff` is also an effective linter.

In *Chapter 16*, the *Creating a `pyproject.toml` file* recipe, and also In *Chapter 15*, the *Combining `unittest` and `doctest` tests* recipe, both talk about defining test tools in addition to the dependencies required to install the project.

This suggests there are several layers of requirements (also called dependencies):

- Requirements needed to install the application in the first place. Within this book, this includes projects like `pydantic`, `beautifulsoup4`, `jupyterlab`, and `matplotlib`.
- Optional requirements for special features, plug-ins, or extensions. These aren't required to install the project. They are named in configuration files and applied when the software is used. As an example, the `pydantic` package has an optional validator for email addresses. If your application requires this, it needs to be named as part of the dependency.
- Requirements to run the test suite. For the most part, this has been `pytest` and `mypy`. It hasn't been emphasized, but the unit test cases for the examples in this book all use `tox` for test automation.

- Packages and tools needed for development. This includes tools like `memray` and `sphinx`. A tool like `ruff` or `black` might be part of this set of requirements.

The dependency information is used to install the software properly. It's also used to create development environments for collaboration. The Python ecosystem of packages is in a constant state of flux.

It's imperative to record which versions a package was tested with. This detail makes it possible for tools like **PIP** to download and install the required components in the virtual environment.

## Getting ready

The first step is to create the base `pyproject.toml` file. See the *Creating a `pyproject.toml` file* recipe in *Chapter 16* for another recipe closely related to this. This should have a `dependencies` item in the `[project]` table. It might look like this:

```
[project]
  # details omitted
dependencies = [
  "pydantic",
  "beautifulsoup4",
  "types-beautifulsoup4",
  "jupyterlab",
  "matplotlib"
]
```

When using **Poetry**, this information is in a slightly different format. The information goes in the `[tool.poetry.dependencies]` table. Often, we'll build this by using the `poetry add` command-line tool.

Note that **Poetry** commands offer some additional syntax:

```
% poetry add pydantic@^2.6.0
```

The `^` prefix is a sophisticated rule that permits a larger version number for the minor or patch level. It does not permit any changes to the left-most, major version number, 2,

in this case. This means that any version of Pydantic at or after 2.6.0 will be considered. Versions above 3.x will not be considered.

## How to do it...

1. Add the test dependencies in a table named `[project.optional-dependencies]`.

This will be a list named `test`. It looks like this:

```
[project.optional-dependencies]
test = [
    "tox",
    "pytest",
    "mypy"
]
```

This name of `test` can be used by `pip-compile` to build a detailed `requirements-test.txt` for the test tools.

When using **Poetry**, this optional dependency group is in a different table. We use the `--group` option to specify the group.

The command line would look like this:

```
% poetry add tox@^4.0 --group test
```

2. Add the development dependencies in a table named `[project.optional-dependencies]` Generally, the name `dev` is used. It looks like this:

```
[project.optional-dependencies]
dev = [
    "ruff",
    "pip-tools",
    "memray"
]
```

This name of `dev` can be used by `pip-compile` to build a detailed `requirements-dev.txt` for the entire suite of tools, plus the base dependencies.

When using **Poetry**, the `--group` option specifies the group. An add command might include `--group dev` to add an item to the dev group.

## How it works...

The goal is to provide ranges and patterns in the `pyproject.toml` file, offering flexibility in version identification. Separate `requirements*.txt` files record specific version numbers used for the current release. This generic-specific distinction supports the integration and reuse of complex packages.

## There's more...

When working with tools like `tox`, we can create multiple virtual environments to test our software with variants on the dependencies.

Package installation often uses a `requirements.txt` file with specific version identification. Development efforts, on the other hand, may involve a number of alternative virtual environments.

We can use tools like `pip-compile` to create the mix of packages to permit testing in a number of alternative virtual environments. See *Workflow for layered requirements* at <https://pip-tools.readthedocs.io/en/latest/> for more information.

We'll often create a base `requirements.in` file to define the common requirements across all virtual environments. For more information on this, see *Chapter 16, Dependencies and Virtual Environments*. This is often a simple list of the packages required:

```
# requirements.in
this_package
sphinx
```

This provides the baseline set of packages unique to the project.

We can then create layered `requirements-dev_x.in` files for various test environments. Each of these files will include the base layer `requirements.txt` and an additional set of constraints. The file might look like this:

```
# requirements_dev_x.in
# Anticipation of new release. See ... for details.
-c requirements.txt
some_package>2.6.1
```

We've included a comment that provides information on why this distinct development virtual environment is required. These reasons change frequently, and it's helpful to leave reminders on why a particular environment is helpful.

Within a `tox.ini` file, the `pip-sync` command will build a distinct virtual environment for testing. We'll look at this in the *Using tox to run comprehensive quality checks* recipe.

## See also

- In *Chapter 16*, the *Creating a `pyproject.toml` file* recipe shows a way to start a `pyproject.toml` file.
- See *Using tox to run comprehensive quality checks* for more information on using the `tox` tool to run a test suite.

## Using tox to run comprehensive quality checks

When we start using multiple CI/CD tools, it's essential to make sure all of the tools are used consistently. The virtual environments must also be built consistently.

Traditionally, a tool like `make` was used to rebuild target files when source files were modified. This requires a great deal of care because Python doesn't really fit the compiler-centric model of `make`.

Tools like `tox` and `nox` are far more helpful for running comprehensive sequences of test and CI/CD tools on Python code.

## Getting ready

For careful software development, a variety of tools can be useful:

**Unit testing** : We can use the built-in `doctest` or `unittest` modules. We can also use

tools like `pytest` to find and run a test suite.

**Benchmarking** : Also called performance testing. The `pytest-benchmark` project offers a handy fixture for assuring performance meets expectations.

**Acceptance testing** : A tool like `behave` or the `pytest-bdd` plug-in can help by stating acceptance test cases in Gherkin to make them more readily understood by product owners.

**Type hint checking** : This is often handled by tools like `mypy`, `pyre`, `pyright`, or `pytype`.

**Linting** : While the term “linting” is in common use, this is really better termed “lint blocking”. There are numerous tools, including `ruff`, `pylint`, `flake8`, and `pylama`.

**Style and formatting** : Two popular tools for this are `ruff` and `black`.

**Documentation** : This is often built with `Sphinx`.

This means we’ll need to install the chosen suite of tools. One more tool can be helpful to find them all and bind them into a usable form. The `tox` tool can create and run tests in multiple virtual environments.

We’ll need to download and install the `tox` tool. Generally, this is done with the following terminal command:

```
(cookbook3) % python -m pip install tox
```

Using the `python -m pip` command ensures that we will use the `pip` command that goes with the currently active virtual environment.

## How to do it...

1. There are two ways to provide configuration files for `tox`. We can embed the configuration in the `pyproject.toml`. While this fits the philosophy of the file, the `tox` tool doesn’t handle TOML options. It relies on a string with INI-formatted options in the TOML file.

A better alternative is to create a separate `tox.ini` file. In this file, create an initial



[tox] table with core configuration options. The following are appropriate for many projects:

```
[tox]
description = "Your project name goes here."
min_version = 4.0
```

2. For applications or scripts that don't need to be installed, the following two lines are appropriate to avoid trying to prepare and install a package:

```
skip_sdist = true
no_package = true
```

For packages that will be distributed and installed, add nothing.

3. Create a [testenv] table with general information about test environments. In some cases, a single environment is sufficient. When multiple, distinct environments are required, there will be multiple [testenv] sections.

```
[testenv]
```

4. Inside this [testenv] table, the deps= value lists the test tools that will be used. It might look like this:

```
deps =
    pytest>=7
    pip-tools
    ruff>=0.1.4
    mypy>=1.7
```

The tox tool uses a pip command to build the items listed in the deps section. It can, of course, be used to install **all** of the requirements. Using `-r requirements.txt` will do this.

It's somewhat more efficient to use the pip-sync tool because it can avoid reinstalling any dependencies that are already present in the environment. When using pip-sync,

we do not use `-r requirements.txt` in the `deps=` list.

5. If `pip-sync` is being used to install requirements, this is given as the `commands_pre=` value:

```
commands_pre = pip-sync requirements.txt
```

6. If any unique environment variables are required, they're set by the `setenv=` value:

```
setenv =  
    PYTHONPATH=src/ch03
```

7. Finally, provide the sequence of commands to execute:

```
commands =  
    pytest --doctest-glob='*.txt' src  
    ruff format src  
    ruff check src  
    mypy --strict src
```

8. After closing this file, use the following command to run the test suite:

```
(cookbook3) % tox
```

## How it works...

A great many assumptions and defaults are built into tools like `tox`. This saves us from having to write clever shell scripts or tinker with the assumptions present in a `makefile`. Instead, we can provide a few lines of configuration and a sequence of commands.

Ideally, using `tox` always looks like this:

```
(cookbook3) % tox  
... details omitted  
  
congratulations :) (4.09 seconds)
```

The final **congratulations** is an apt summary.

## There's more...

In many cases, a project has multiple virtual environments. Virtual environments are distinguished using extended names. These will have the general pattern of `[testenv:name]` where `name` is something descriptive.

In the `[tox]` section of the configuration, `env_list` lists the environments to process automatically. Environments not listed can be executed manually by using the `-e` option on the `tox` command.

To test with another version of Python, we add the following to our `tox.ini` file:

```
[testenv:py311]
    base_python = py311
```

This will inherit the details of the master `testenv` settings. An override is applied to change the base Python version to 3.11.

The `py311` name is a handy `tox` shorthand for a longer specification like `python>=3.11`. The tool will search the system-wide `PATH` for candidate Python implementations. To test with multiple Python versions, they all need to be installed in directories named in the `PATH`.

## See also

- See *Identifying other CI/CD tools in `pyproject.toml`*
- See *Chapter 15* for more information on testing.
- See *Chapter 16* for recipes related to virtual environments.
- See <https://tox.wiki/en/latest/> for more information on the `tox` tool.
- See <https://nox.thea.codes/en/stable/> for information on the `nox` tool, which offers similar functionality.

## Join our community Discord space

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

<https://packt.link/dHrHU>







[www.packt.com](http://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

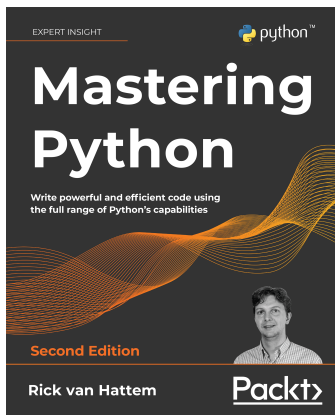
Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [packt.com](http://packt.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [customercare@packtpub.com](mailto:customercare@packtpub.com)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

At [www.packt.com](http://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:



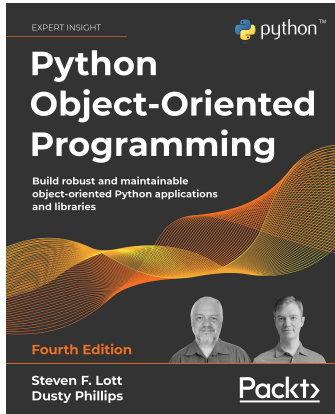
## Mastering Python - Second Edition

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

## Packt is searching for authors like you

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

## Share your thoughts

Now you've finished *Modern Python Cookbook, Third Edition*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.



<https://packt.link/r/1835466389>

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



# Index

## Symbols

() characters 56, 57  
    using, to break long statement  
        54

\* separator  
    used, for forcing keyword-only  
        arguments 114–118

/ separator  
    used, for defining position-only  
        parameters 119–122

/ true division operator 12

// truncated division operator 12

:= walrus operator  
    used, for saving intermediate  
        results 73–75

\_\_exit\_\_() method 317

\_\_init\_\_() method 287, 288

\_\_slots\_\_  
    used, for optimizing small objects  
        292–296

## A

abstract superclass 343

acceptance testing 641

add method 179

    used, for building set 179, 180

aggregation 330, 331

alabaster 753

Anaconda

    Command-line installers 728

    download link 728, 732

    Graphical installers 728

    URL 727

    using 727–730

annotations 432

anonymous tuples 42

Anscombe's Quartet 540

API reference

    creating, with Sphinx autodoc  
        754–759

append operations 428

append() method 152

    used, for building list 152, 153

application

    logging 612–614

    structure 612, 613

aptitude 707

argparse

    used, for obtaining command-line  
        input 243–248

- arguments
  - managing, in composite applications 620–626
- arrange-act-assert pattern 642
- arrays 150
- ASCII bytes
  - creating 29–32
- assignment 211–214
- association 329
- associative array 195
- associative store 202
- autodoc 61
- axes
  - using, directly to create scatter plot 552–556

**B**

- backslash
  - using, to break long statement into logical lines 53, 54
- bag 149, 298
- bang 51
- bash 637, 709
- BBEdit 15
  - download link 48
- Beautiful Soup 35
  - reference link 35, 524
- benchmarking 642, 765
- billboard comments 595
- binary mode 472
- black 765
- block devices 471

- blocks and statements
  - reference link 73
- bootstrapping 687
- break statements
  - used, for avoiding potential problem 77–80
- built-in collection
  - extending 303–307
- built-in type matching functions
  - using 440–442
- built-in venv
  - used, for creating environments 706–708, 710
- bytes
  - decoding 33–36

## C

- C3 algorithm 336, 342
- Cascading Style Sheet (CSS) 525
- cells
  - creating, with Python code 535–540
- changed dependency
  - incompatible, with another dependency 739–741
- character devices 472
- chunking 108
- CI/CD tools
  - identifying, in pyproject.toml 760–763
- class
  - creating, for orderable objects

- 358–363
- designing, with multiple
  - processing 274–279
- used, for encapsulating data and
  - processing 265–268
- using, as namespace for
  - configuration 585–589
- class definitions
  - essential type hints 269–273
  - using 589, 590
- class pattern 446
- class-level static variables 351, 352
- clean-up feature
  - adding, to script 631
- clear documentation strings
  - writing, with RST markup 129–133
- CLI applications
  - combining 627–631
  - wrapping 627–631
- Click 59
  - reference link 239
- closure 418
- cmath package 8
- cmd
  - used, for creating command-line applications 253–256
- cmd module
  - interaction via 240
- Code cell 538
- code first design 436–439
- code rot 612
- collections 145, 195
  - summarizing 399–402
  - transformations, applying to 381–385
  - using 297–301
- color math and programming code
  - examples
    - reference ink 101
- come-out roll 68
- comma-separated values (CSV) 354, 488, 495
- Command design pattern 605, 625
  - used, for combining multiple applications 614–617
- command-line applications
  - creating, with cmd 253–256
- command-line arguments 257
  - processing 248, 249
- command-line input
  - obtaining, with argparse 243–248
  - obtaining, with invoke 249–252
- Command-Line Interface (CLI) 609
- complex data structure
  - serializing 513–516
- complex files
  - rewriting, into easier-to-process format 506, 507
- complex formats
  - reading, with regular expressions

- 500–505
- complex if...elif chains
  - designing 67–72
- complex numbers 8
- complex structures
  - using 353–357
- complex text 240
- complexity of algorithm 300, 301
- complicated strings
  - building, from lists of strings
    - 23–25
    - building, with f-strings 18–22
- composite applications
  - arguments, managing 620–626
  - issues 609
- composite object validation 239
- composites 413
  - creating 617–619
- composition 329–331
  - scripts, designing 591–594
- comprehension 152, 154, 179, 198
  - dictionary, building as 199, 200
- comprehensive quality checks
  - running, with tox 764–767
- compute-intensive processing
  - Eager 275
  - Lazy 275
- concrete paths 475, 480
- Conda 534
  - obtaining 728
  - using 727–730
- conda-forge
  - URL 728
- configuration
  - class, using as namespace
    - 585–589
    - managing, in composite applications
      - 620–626
- configuration files 258
  - finding 569–573
  - handling, methods 568
  - options 570, 571
  - Python, using for 580–584
  - tiers 573, 574
  - TOML, using for 575–579
- container architectures 539
- containers 145, 195
- context managers 324
  - creating 313–317
- contexts
  - creating 313–317
  - managing, using with statement
    - 91–94
- control and audit output
  - logging, used for 596–601
- conversion function 152, 179, 198
- copy of list
  - reversing 173–177
- Craps 68
- CRUD operations 202
- CSV dialect 488
- CSV files
  - dataclasses, using to simplify working with
    - 495–498

- CSV module
  - used, for reading delimited files 488–491
- curl command 732
- currency
  - performing, calculations 3, 4
- D**
- Dask 302
- data
  - ingesting, into notebook 540–544
- data and processing
  - encapsulating, with class 265–268
- data model
  - reference link 297
- data structure
  - selecting 146–150
- data structures and algorithms
  - reference link 150
- data type validation 239
- dataclasses
  - using, for mutable objects 283–287
  - using, to simplify working with CSV files 495–498
- datetime objects
  - used, for writing tests 681–685
- decimal 6, 7
- deep copies of objects
  - creating 215–220
- definition list 133
- del statement 164, 204
- delimited files
  - reading, with CSV module 488–491
- dependencies 760
  - levels of specificity 723
  - modifications, caused test failure 737–739
  - modifications, managing 736, 737, 741
- descriptions
  - including 58, 61
  - including, in script 57
- deserialization 472
- dictionaries
  - building, as comprehension 199, 200
  - building, by setting items 198, 199
  - creating 196–198, 200, 201
  - shrinking 202–204
- dictionary objects
  - building 198
- dictionary-related type hints
  - writing 205–210
- difference() method 186
- directives
  - using 66
- div-mod pair 9
- docstrings 49
  - used, for testing 643–645, 648
  - writing, for library modules 60, 61



- writing, for scripts 58–60
- writing, with RST markup 62–65
- doctest** 51, 180
- doctest block** 65
- doctest examples**
  - writing, for floating-point values 658, 659
  - writing, with object IDs 657, 658
  - writing, with unpredictable set ordering 657
- doctest issues**
  - handling 654–657, 659, 660
- doctest parser**
  - test case 648
- doctest tool** 643, 647, 653
- Document Object Model (DOM)** 530
- documentation** 743, 765
  - creating 749–752
  - including 58, 61
  - including, in script 57
  - writing, for library modules 58
  - writing, scripts 58
- Docutils** 62, 67, 133, 561
  - reference link 67, 133
- domain validation** 239
- Don't Repeat Yourself (DRY)** 84
- duck typing** 343
- dynamic content**
  - creating, methods 558, 559
- E**
  - eager function** 374
  - eager processing** 307, 308
  - edge case** 649
  - elevator pitch** 746
  - else clause**
    - using, on for statement 80
  - Else-Raise design pattern** 71, 110
  - end-to-end testing** 641
  - environment variables** 257
  - environments**
    - creating, with built-in venv 706–708, 710
  - escape sequences** 27
  - essential features** 336
  - except:clause**
    - used, for avoiding issue 86–88
  - exception matching rules**
    - leveraging 81–84
  - exception root cause**
    - concealing 88–90
  - exceptions**
    - handling, options 81
    - internal attributes 90
    - raising, functions testing 651–653
  - execve** 51
  - explicit line joining** 56
  - exponent** 7
  - Extensible Markup Language (XML)** 516
  - extensions for Sphinx** 752
  - external resources**

- mocking 693–698
- F**
- f-strings
  - reference link 23
  - used, for building complicated strings 18–22
- Facade class 615
- factory function/factory class 225, 338
- fail-safe file output 483
- Fibonacci number
  - computing 137–139
- field list 755
- field() function 291
- file dates
  - comparing 478
- filenames
  - pathlib, using to work with 474–480
- files 471, 472
  - finding, to match given pattern 478, 479
  - replacing, while preserving previous version 482–486
- filesystem
  - modifying, method 480, 481
- filter() function 368
  - applying, ways 394–398
  - using 397, 398
  - using, in generator expression 396, 397
- First Normal Form (1NF) 158, 489, 505
- First-In-First-Out (FIFO) 299
- flake8 765
- flit 722
- float 6, 7
- floating point 3
- floating-point approximations 5, 6
- floating-point values
  - used, for writing doctest examples 658, 659
- floor division
  - performing 10
- for statement
  - using 383
- format\_map() method 22
- fraction 6, 7
- fraction calculations 4, 5
- frozen dataclasses
  - using, for immutable objects 289–291
- function definitions 97
- function parameters 98–101
  - mutable default values, avoiding 221–225
- functional programming 371
- functional testing 641
- functions
  - designing, with optional parameters 103, 104, 108
  - examples, writing 645
  - logging 599

- testing, that raise exceptions
  - 651–653
- wrapping 125
- functools.partial()
  - using 420
- f“{value=}” strings
  - debugging with 241, 242

## G

- garbage collection 25
- gedit 48, 51
- general to particular design approach
  - 106–108
- generalization-specialization
  - relationship 305, 328
- generator expression
  - filter, using 396, 397
  - using 383, 384
- generator functions
  - writing, with yield statement 373–379
- get\_options() function 626
- getpass()
  - using, for user input 235–239
- Git 708
- GIVEN-WHEN-THEN 642
- global object
  - managing 347–349, 352, 353
- globbing feature 247
- GNU License List
  - URL 717
- graphs 150

## H

- happy path 661
- has-a relationship 329
- hashes 150
- hatch 718
- header row
  - excluding 390
- higher-order function 422
- HTML documents
  - reading 523–530
- Hue-Saturation-Lightness (HSL) values
  - converting, RGB numbers 101–103

## I

- identity element 404, 405
- IEEE Standard for Floating-Point Arithmetic (IEEE 754)
  - reference link 9
- immutable mapping 149
- immutable objects 283
  - frozen dataclasses, using for 289–291
  - typing.NamedTuple, using for 280–282
- immutable sequence 149
- immutable set 149
- implicit line joining 56
- index-only tuples 42
- inheritance
  - and extension 332, 333
- inheritance and composition
  - selecting between 328–331, 333,

- 334
- inline markup
  - using 66, 67
- input filename's suffix
  - output filename, created by
    - modifying 476, 477
- input()
  - using, for user input 235–239
- insertion 198
- Integrated Development Environment (IDE) 2, 533
- integration testing 641
- interactive input 257
- intermediate results
  - assigning, to separate variables 55, 56
  - saving, with := walrus operator 73–75
- Internet of Things 462
- Inverse Power Law of Reuse 612
- invoke 59
  - reference link 253
  - used, for obtaining command-line input 249–252
- Invoke-WebRequest Powershell
  - command 733
- is-a relationship 328, 330
- isinstance() function 443
- items
  - adding, to set 182, 183
  - extracting, from tuple 38, 39
  - removing, from set 186, 187

## J

- JavaScript Object Notation (JSON)
  - 508
  - URL 508
- JSON documents
  - reading 508–512
  - structures 512
- Jupyter Lab 533
  - ways, to stop processing 540
- Jupyter Book 558, 561

## K

- keyword parameters
  - used, for creating partial function 126
- keyword-only arguments
  - forcing, with \* separator 114–118
- Komodo IDE
  - download link 48

## L

- lambda object
  - building 128
- Last-In-First-Out (LIFO) 205, 299
- Law of the Excluded Middle 68
- lazy attributes
  - properties, using for 307–312
- lazy generator 373
- lazy processing 307, 308
- library modules 57
  - docstrings, writing for 60, 61
- line joining 56
- linting 765

- Liskov Substitution Principle (LSP)
    - 329, 617
  - list
    - building 151, 152, 154, 155
      - building, with append() method 152, 153
    - dicing 157–162
    - extending, ways 156, 157
    - shrinking 163, 164, 166
    - slicing 157–162
  - list comprehension
    - writing 153, 154
  - list function
    - using 226
  - list, of complicated objects
    - deleting from 364–367
  - list-related type hints
    - writing 169–172
  - lists of strings
    - complicated strings, building from 23–25
  - literal 152, 179, 198
  - load testing 642
  - locks, using in with statement
    - reference link 95
  - log\_parser() function
    - using 504, 505
  - logging
    - used, for control and audit output 596–601
  - logical layout
    - handling, in CSV files 488, 489
  - logical line 53
  - logical reduction functions 404
  - long lines of code
    - writing 52, 53, 56
  - long statement
    - breaking, into logical lines with backslash 53, 54
    - breaking, into sensible pieces with () characters 54
- ## M
- magic bytes 51
  - man page
    - reference link 62
  - mantissa/significand 7
  - map and reduce transformations
    - combining 405–411
  - map() function 384–386
  - markdown cells 538
    - details, adding to 558–560
  - Markdown markup 745
  - Markov chain state changes 606
  - match statement
    - using 444–446
  - math module 7, 8
  - matplotlib 533, 546, 555
  - Matplotlib Examples Gallery 551
  - measure
    - reference link 52
  - memoization technique 138
  - memory leak 91
  - memray 760
  - Method Resolution Order (MRO) 333, 339

- miniconda
    - Linux installer 728
    - macOS installer 728
    - reference link 728
    - Windows installer 728
  - mixed content model 519
  - mixin 327, 336
    - design concerns, separating 340
  - mkdir() method
    - keyword parameters 481
  - model validation 462
  - modeline 51
  - module files
    - writing 47–51
  - module global variables 349–351
  - monotonic superclass linearization for Dylan
    - reference link 342
  - multiple applications
    - combining, with Command design pattern 614–617
  - multiple contexts
    - managing, with multiple resources 319–324
  - multiple hosts 302
  - multiple inheritance
    - using, to separate rules for specific games 335–340
  - multiple processes 302
  - multiple processing
    - classes, designing with 274–279
    - multiset 149, 151, 298
    - mutable default values
      - avoiding, for function parameters 221–225
    - mutable mapping 149
    - mutable objects 283
      - dataclasses, using for 283–287
    - mutable sequence 149
    - mutable set 149
    - mypy 43, 97, 98, 100, 101, 128, 209, 269, 270, 272, 273, 303, 313, 318, 334, 337, 341, 346, 432, 436, 692, 738, 765
- ## N
- NamedTuples
    - using, to simplify item access in tuples 40–42
  - narrow data validation rules
    - applying 467–469
  - National Oceanographic and Atmospheric Administration (NOAA) 462
  - National Weather Service
    - reference link 33
  - nested try statements
    - including 85
  - NetworkX 150
    - reference link 151
  - Newline Delimited JSON URL 513

NIST Aerosol Particle Size 274, 308

notebook

- cells, updating to add linear regression computation 556
- data, ingesting into 541–544
- unit test cases, including 562–565
- working with 535–540

Notepad++

URL 48

nox 764

number of sibling output files

- creating, with distinct names 477, 478

numeric tower 448, 449

NumPy

URL 150

## O

object IDs

- used, for writing doctest examples 657, 658

operating system (OS) 471

operation definition 403

optional parameters

- used, for designing functions 103, 104, 108

orderable objects

- class, creating 358–363

ordinary input 235

OS environment settings

- using 257–261

OS-specific tools 705

output

- checking 632–637

output filename

- created, by modifying input filename's suffix 476, 477

overloaded function 103

## P

packages

- installing, with requirements.txt file 711–715

Pandoc 561

parse() function 453

- defining 502, 503

partial function

- creating 417–421
  - creating, with keyword parameters 126
  - creating, with positional parameters 126, 127
- order for parameters, selecting based on 122, 124, 125

particular to general design approach 104–106

pathlib 487

- using, to work with filenames 474–480

pattern for collections

- aggregation 329
- composition 329

- inheritance 330
- patterns, for applying function to set of
  - data
  - filtering 372
  - mapping 372
  - reducing 372
- performance testing 641, 642, 765
- persistent environment, viewpoints
  - Actual Environment 704
  - Virtual Environment 704
- physical format decoding 473
- PIP 706, 714, 719, 727, 729, 731, 761
- PIP operation 720, 721
- pip-compile 723, 729
- pip-sync 723
- pip-tools
  - used, for managing
    - requirements.txt file 723–725
- pkg 707
- plain text 745
- poetry tool 718, 761, 762
  - download link 732
  - reference link 732, 736
  - using 732–735
- pop() method 165, 184, 186, 187, 204, 205
- popitem() method 205
- position-only parameters
  - defining, with / separator 119–122
- prefixes, for types 20
- print() function
  - features, using 230–234
- program
  - wrapping 628, 632–637
- properties
  - using, for lazy attributes 307–312
- property file format 77
- Property Inspector 561
- protocol
  - defining 346
- pure paths 475, 571
- PyCharm Professional
  - download link 48
- Pydantic 210, 462, 547, 553, 589
  - reference link 211
  - supporting, for JSON Schema 460, 461
  - used, for implementing strict type checks 455–460
  - validation features 464
- pydoc 51, 133
- pylama 765
- pylint 765
- Pyoxigraph 150
- pyoxigraph 0.3.22
  - reference link 151
- pyplot 546
  - using, to create scatter plot 546–550
- pyproject.toml file
  - CI/CD tools, identifying 760–763
  - creating 716–721



pyre 765  
pyright 98, 765  
    reference link 103  
pytest 594, 662, 682  
pytest and doctest tests  
    combining 677–680  
pytest module  
    used, for unit testing 672–676  
Python  
    download link 706  
    duck typing, leveraging  
        342–346  
    using, for configuration files  
        580–584  
Python code  
    used, for creating cells 535–539  
Python Package Index (PyPI)  
    URL 711  
Python Packaging User Guide  
    reference link 712  
Python script  
    writing 47–51  
Python Semantic Release  
    reference link 722  
Python syntax 45, 46  
Python-based tools 706  
Python's stack limits  
    recursive functions, designing  
        134–137  
pytype 765  
PyYAML 6.0.1  
    reference link 508

## Q

Quarto 558, 561

## R

randomness  
    involving, items testing  
        686–688, 690  
Rate-Time-Distance (RTD) 109  
rational fraction calculations 11, 12  
rational fraction value 9  
rational numbers/fractions 3  
Raw cell 538  
RDFLib 150  
rdflib 7.0  
    reference link 151  
read-evaluate-print loop (REPL) 2,  
    253  
README file  
    writing, challenges 744  
README.rst file 744–748  
recursive functions  
    designing, around Python's stack  
        limits 134–137  
recursive generator functions  
    writing, with yield from statement  
        423–427  
reduction 135  
references 211–214  
regular expressions  
    syntax rules 17  
    used, for reading complex  
        formats 500–502,  
        505

- used, for string parsing 13–16
- remove() method 164, 165, 184, 186, 187
- requirements.txt file
  - managing, with pip-tools 723–725
  - used, for installing packages 711–715
- ReStructured Text (RST) 58, 745
  - syntax rule 61
  - used, for writing clear
    - documentation strings 129–133
  - used, for writing docstrings 62–65
- RGB numbers
  - converting, into Hue-Saturation-Lightness (HSL) values 101–103
- Rich's documentation
  - reference link 239
- role 67
- row objects
  - creating 390–392
- row-level cleansing function 492–494
- rows
  - restructuring 388, 389
- rpm 707
- ruff 738, 760, 765
- run-time environment
  - Persistent Aspects 703, 704
  - Transient Aspects 704

- run-time valid value checks
  - including 462–467

## S

- scatter plot
  - creating, with axes directly 552–556
  - creating, with pyplot 546–550
- schema 488
- script-library switch
  - used, for writing testable scripts 139–143
- scripts 58
  - designing, for composition 591–594
  - docstrings, writing for 58–60
- secure, no echo input 235
- sequence 149
- serialization 472
- set builder 372
- set comprehension 372
  - writing 180, 181
- set operators 183, 184
- set-related type hints
  - writing 188–192
- setdefault() method 201
- sets 148
  - building 177–179, 181, 182
  - building with add method 179, 180
  - shrinking 184–186
- setuptools 718, 719
- severity levels, logging package

- CRITICAL 598
- DEBUG 598
- ERROR 598
- INFO 598
- WARNING 598
- shallow copies of objects
  - creating 215–220
- shallow copy 162
- shared global state 353
- sharp 51
- shebang 51
- Shields.io
  - reference link 748
- sight-gauge 158
- signature 103
- single process 302
- Single Responsibility Principle 612
- singleton object 347, 580, 581
  - managing 347–349, 352, 353
- skip-when-delete problem
  - avoiding 167, 168
- slice assignment 165, 166
- slice operator 161, 162
- Slide Type 561
- small objects
  - optimizing, with `__slots__` 292–296
- SOLID design principles 268, 269, 329, 612
- solid-state drives (SSD) 471
- Spark 302
- SPDX License List
  - URL 717
- Sphinx 51, 61, 66, 133, 561, 595, 765
  - installing 749–752
  - reference link 62, 67
  - used, for creating API reference 754–759
- Sphinx Themes Gallery
  - reference link 750
- spike solution 143
  - reference link 629, 634
- stacked generator expressions
  - using 386–392
- stateful objects
  - examples, writing 646
- strict type checks
  - implementing, with Pydantic 455–460
- string
  - encoding 30–32
  - parsing, with regular expressions 13–16
- string literal concatenation
  - using 55
- style and formatting 765
- Sublime Text
  - URL 48
- subset
  - selecting 394–398
- Summation of Primes
  - reference link 416
- super flexible keyword parameters
  - using 109–114
- sys module 234
- system testing 641

**T**

- table 576
  - tail recursion 135
  - testable scripts
    - writing, with script-library switch 139–143
  - testing 641
    - docstrings, used for 643–645, 648
  - tests
    - running 647
    - writing, with datetime objects 681–685
  - text mode 472
  - there exists processing
    - implementing 412–415
  - TimeComplexity
    - reference link 157
  - TOML
    - configuration
      - points 602, 603
    - reference link 575
    - using, for configuration files 575–579
  - Tox 725, 763, 765
    - reference link 736
    - using, to run comprehensive quality checks 764–767
  - transformations
    - applying, to collection 381–385
  - trees 150
  - true division
    - performing 11
  - true division and floor division
    - selecting, between 9, 10
  - true value 9
  - tuples 218
    - creating 37, 38
    - items, extracting from 38, 39
    - NamedTuples, using to simplify item access 40–42
  - tuples of items
    - using 36–40
  - twine 722
  - two applications
    - combining, into single application 606–612
  - type conversions
    - handling 448–452
  - type hint checking 765
  - type hinting
    - composition layer 172
    - foundation layer 172
  - type hints 98–101, 312
    - designing with 432–434, 437, 438
    - using, aspects 432, 433
  - type hints first design 434–436, 439
  - typing.NamedTuple
    - using, for immutable objects 280–282
- 
- U**
- Unicode 15.1 Character Code Charts
    - reference link 29

**Unicode characters**

- reference link 26
- using 26–29

**Unicode encodings**

- reference link 32

**unit test cases**

- including, in notebook 562–565

**unit testing 641, 764**

- with pytest module 672–676
- with unittest module 661–667

**unittest and doctest tests**

- combining 667–670

**unpredictable set ordering**

- used, for writing doctest examples  
657

**unsafe hash 287****User Experience (UX) design 621****user input**

- getpass(), using for 235–239
- input(), using for 235–239
- reference link 240

**UTF-8 bytes**

- creating 29–32
- reference link 36

**V****variables 211–214**

- assigning to 214

**venv 727****Vim 48, 51, 52****virtual environment**

- activating 708, 709

- creating 708
- deactivating 709
- principle use cases 706

**W****with statement**

- used, for managing context  
91–94

**X****XML documents**

- reading 516–522

**XML Path Language (XPath) 522****Y****YAML Ain't Markup Language (YAML)**

- URL 508

**YAML documents**

- reading 508–512

**yield from statement**

- used, for writing recursive  
generator functions  
423–427

**yield statement**

- used, for writing generator  
functions 373–379
- using 383

**yum 707****Z****z-scores 418****zsh 258, 570, 709, 730**

## Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781805127161>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

