Modern Python Cookbook

The latest in modern Python recipes for the busy modern programmer

Packt>

www.packt.com

Steven F. Lott

Modern Python Cookbook

The latest in modern Python recipes for the busy modern programmer

Steven F. Lott

BIRMINGHAM - MUMBAI

Modern Python Cookbook

Copyright © 2016 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, and its dealers and distributors will be held liable for any damages caused or alleged to be 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.

First published: November 2016

Production reference: 1200918

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78646-925-0

www.packtpub.com

GET

80%

OFF ANY PACKT

EBOOK AND VIDEO

Go to www.packtpub.com
and use this code in the checkout:

HBBI80OFF

Packt›

Credits

Author
Steven F. Lott

Reviewers
Sanjeev Kumar Jaiswal
Dr. Vahid Mirjalili

Commissioning Editor
Kunal Parikh

Acquisition Editor
Sonali Vernekar

Content Development Editor
Zeeyan Pinheiro

Technical Editors
Pratish Shetty
Abhishek Sharma

Copy Editor
Safis Editing

Project Coordinator
Suzanne Coutinho

Proofreader
Safis Editing

Indexer
Tejal Daruwale Soni

Graphics
Kirk D'Penha

Production Coordinator
Aparna Bhagat

About the Author

Steven F. Lott has been programming since the 70s, when computers were large, expensive, and rare. As a contract software developer and architect, he has worked on hundreds of projects, from very small to very large. He's been using Python to solve business problems for over 10 years.

He's currently leveraging Python to implement microservices and ETL pipelines. His other titles with Packt Publishing include *Python Essentials*, *Mastering Object-Oriented Python*, *Functional Python Programming*, and *Python for Secret Agents*.

Steven is currently a technomad who lives in various places on the east coast of the U.S. His technology blog is `http://slott-softwarearchitect.blogspot.com` and his LinkedIn address is `https://www.linkedin.com/in/steven-lott-029835`.

About the Reviewers

Sanjeev Jaiswal is a computer graduate with 7 years of industrial experience in web development and cyber security. He basically uses Perl, Python, and GNU/Linux for his day-to-day activities. He is currently working on projects involving penetration testing, source code review, and security design and implementations.

He is very much interested in web and cloud security. You can follow him on Twitter at `@aliencoders` and on GitHub at `https://github.com/jassics`.

He has written *Instant PageSpeed Optimization* and co-authored *Learning Django Web Development* for Packt Publishing. He has reviewed more than 5 books for Packt Publishing and looks forward to authoring or reviewing more books for Packt Publishing and other publishers.

Vahid Mirjalili is a software engineer and data scientist, currently working towards his PhD study in Computer Science at Michigan State University. His research at the i-PRoBE (integrated pattern recognition and biometrics) lab involves attribute classification of face images from large image datasets.

Furthermore, he teaches Python programming as well as computing concepts for data analysis and databases. Owing to his specialty in data mining, he is very interested in predictive modeling and getting insights from data. He is also a Python developer and likes to contribute to the open source community.

Moreover, he enjoys making tutorials for different directions of data science and computer algorithms, which can be found in his GitHub repository at `http://github.com/mirjalil/DataScience`.

www.PacktPub.com

For support files and downloads related to your book, please visit `www.PacktPub.com`.

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

At `www.PacktPub.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.

`https://www.packtpub.com/mapt`

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Table of Contents

Preface 1

Chapter 1: Numbers, Strings, and Tuples 6
- **Introduction** 6
- **Creating meaningful names and using variables** 7
  - Getting ready 7
  - How to do it... 8
    - Choosing names wisely 8
    - Assigning names to objects 10
  - How it works... 11
  - There's more... 12
  - See also 12
- **Working with large and small integers** 13
  - Getting ready 13
  - How to do it... 13
  - How it works... 14
  - There's more... 16
  - See also 17
- **Choosing between float, decimal, and fraction** 17
  - Getting ready 17
  - How to do it... 18
    - Doing currency calculations 18
    - Fraction calculations 19
    - Floating-point approximations 20
    - Converting numbers from one type to another 21
  - How it works... 22
  - There's more... 23
  - See also 24
- **Choosing between true division and floor division** 25
  - Getting ready 25
  - How to do it... 25
    - Doing floor division 26
    - Doing true division 26
    - Rational fraction calculations 27
  - How it works... 27
  - See also 28
- **Rewriting an immutable string** 28
  - Getting ready 28
  - How to do it... 29
    - Slicing a piece of a string 29
    - Updating a string with a replacement 30

Making a string all lowercase 31
Removing extra punctuation marks 31
How it works... 31
There's more... 33
See also 33
String parsing with regular expressions 34
Getting ready 34
How to do it... 34
How it works... 36
There's more... 37
See also 37
Building complex strings with "template".format() 38
Getting ready 38
How to do it... 38
How it works... 40
There's more... 42
See also 42
Building complex strings from lists of characters 43
Getting ready 43
How to do it... 44
How it works... 44
There's more 45
See also 46
Using the Unicode characters that aren't on our keyboards 46
Getting ready 46
How to do it... 47
How it works... 47
See also 48
Encoding strings – creating ASCII and UTF-8 bytes 49
Getting ready 49
How to do it... 49
How it works... 50
See also 51
Decoding bytes – how to get proper characters from some bytes 51
Getting ready 52
How to do it.. 53
How it works... 54
See also 55
Using tuples of items 55
Getting ready 55
How to do it... 55
Creating tuples 56
Extracting items from a tuple 56
How it works... 57
There's more 58

See also... 58

Chapter 2: Statements and Syntax 59
Introduction 59
Writing Python script and module files – syntax basics 60
Getting ready 61
How to do it... 61
How it works... 63
There's more... 64
See also 65
Writing long lines of code 65
Getting ready 66
How to do it... 66
Using backslash to break a long statement into logical lines 67
Using the () characters to break a long statement into sensible pieces 67
Using string literal concatenation 68
Assigning intermediate results to separate variables 69
How it works... 69
There's more... 70
See also 70
Including descriptions and documentation 71
Getting ready 71
How to do it... 72
Writing docstrings for scripts 72
Writing docstrings for library modules 74
How it works... 74
There's more... 75
See also 76
Writing better RST markup in docstrings 76
Getting ready 77
How to do it... 77
How it works... 78
There's more... 79
Using directives 81
Using inline markup 81
See also 82
Designing complex if...elif chains 82
Getting ready 83
How to do it... 84
How it works... 85
There's more... 86
See also 87
Designing a while statement which terminates properly 87
Getting ready 87
How to do it... 88
How it works... 91

See also 92
Avoiding a potential problem with break statements 93
Getting ready 93
How to do it... 94
How it works... 95
There's more... 96
See also 96
Leveraging the exception matching rules 97
Getting ready 98
How to do it... 99
How it works... 100
There's more... 100
See also 101
Avoiding a potential problem with an except: clause 102
Getting ready 102
How to do it... 102
How it works... 103
See also 103
Chaining exceptions with the raise from statement 104
Getting ready 104
How to do it... 104
How it works... 105
There's more... 106
See also 107
Managing a context using the with statement 107
Getting ready 107
How to do it... 108
How it works... 109
There's more... 110
See also 110
Chapter 3: Function Definitions 111
Introduction 111
Designing functions with optional parameters 112
Getting ready 112
How to do it... 112
Particular to General Design 113
General to Particular design 115
How it works... 116
There's more... 117
See also 118
Using super flexible keyword parameters 118
Getting ready 119
How to do it... 119
How it works... 121
There's more... 121

See also 122
Forcing keyword-only arguments with the * separator 122
Getting ready 123
How to do it... 124
How it works... 125
There's more... 125
See also 126
Writing explicit types on function parameters 126
Getting ready 127
How to do it... 128
How it works... 129
There's more... 131
See also 132
Picking an order for parameters based on partial functions 132
Getting ready 132
How to do it... 134
Wrapping a function 134
Creating a partial function with keyword parameters 135
Creating a partial function with positional parameters 135
How it works... 136
There's more... 137
See also 137
Writing clear documentation strings with RST markup 137
Getting ready 138
How to do it... 138
How it works... 141
There's more... 142
See also 142
Designing recursive functions around Python's stack limits 142
Getting ready 143
How to do it... 144
How it works... 145
There's more... 146
See also 148
Writing reusable scripts with the script library switch 148
Getting ready 148
How to do it... 149
How it works... 151
There's more... 151
See also 152
Chapter 4: Built-in Data Structures – list, set, dict 153
Introduction 153
Choosing a data structure 154
Getting ready 154
How to do it... 154

How it works... 156
There's more... 158
See also 158
Building lists – literals, appending, and comprehensions 159
Getting ready 159
How to do it... 160
Building a list with the append() method 160
Writing a list comprehension 161
Using the list function on a generator expression 162
How it works... 163
There's more... 163
Other ways to extend a list 164
See also 165
Slicing and dicing a list 165
Getting ready 165
How to do it... 167
How it works... 169
There's more... 169
See also 170
Deleting from a list – deleting, removing, popping, and filtering 170
Getting ready 170
How to do it... 171
Deleting items from a list 171
The remove() method 172
The pop() method 173
The filter() function 173
How it works... 174
There's more... 175
See also 176
Reversing a copy of a list 176
Getting ready 176
How to do it... 178
How it works... 179
See also 179
Using set methods and operators 179
Getting ready 181
How to do it... 182
How it works... 183
There's more... 184
See also 185
Removing items from a set – remove(), pop(), and difference 186
Getting ready 186
How to do it... 187
How it works... 188
There's more... 188
See also 189

Creating dictionaries – inserting and updating 189
Getting ready 190
How to do it... 191
How it works... 192
There's more... 192
See also 194
Removing from dictionaries – the pop() method and the del statement 194
Getting ready 195
How to do it... 196
How it works... 197
There's more... 197
See also 197
Controlling the order of dict keys 198
Getting ready 198
How to do it... 199
How it works... 200
There's more... 200
See also 201
Handling dictionaries and sets in doctest examples 201
Getting ready 201
How to do it... 202
How it works... 203
There's more... 203
Understanding variables, references, and assignment 204
How to do it... 204
How it works... 205
There's more... 206
See also 207
Making shallow and deep copies of objects 207
Getting ready 208
How to do it... 209
How it works... 210
See also 211
Avoiding mutable default values for function parameters 211
Getting ready 212
How to do it... 213
How it works... 214
There's more... 215
See also 216
Chapter 5: User Inputs and Outputs 217
Introduction 217
Using features of the print() function 218
Getting ready 218

How to do it... 220
How it works... 221
There's more... 222
See also 223
Using input() and getpass() for user input 223
Getting ready 224
How to do it... 225
How it works... 227
There's more... 228
Input string parsing 228
Interaction via the cmd module 229
See also 229
Debugging with "format".format_map(vars()) 230
Getting ready 230
How to do it... 231
How it works... 231
There's more... 232
See also 232
Using argparse to get command-line input 232
Getting ready 233
How to do it... 235
How it works... 237
There's more... 238
See also 240
Using cmd for creating command-line applications 240
Getting ready 242
How to do it... 243
How it works... 246
There's more... 247
See also 248
Using the OS environment settings 248
Getting ready 249
How to do it... 250
How it works... 251
There's more... 252
See also 252
Chapter 6: Basics of Classes and Objects 253
Introduction 253
Using a class to encapsulate data and processing 254
Getting ready 254
How to do it... 255
How it works... 258
There's more... 259
See also 260
Designing classes with lots of processing 260

Getting ready 260
How to do it... 263
How it works... 265
There's more... 266
See also 268
Designing classes with little unique processing 268
Getting ready 268
How to do it... 269
Stateless objects 269
Stateful objects with a new class 270
Stateful objects using an existing class 271
How it works... 271
There's more... 272
See also 273
Optimizing small objects with __slots__ 273
Getting ready 273
How to do it... 274
How it works... 276
There's more... 276
See also 277
Using more sophisticated collections 277
Getting ready 277
How to do it... 278
How it works... 280
There's more... 281
See also 282
Extending a collection – a list that does statistics 282
Getting ready 282
How to do it... 283
How it works... 285
There's more... 285
See also 286
Using properties for lazy attributes 286
Getting ready... 286
How to do it... 288
How it works... 290
There's more... 291
See also... 292
Using settable properties to update eager attributes 292
Getting ready 293
How to do it... 294
How it works... 296
There's more... 296
Initialization 297
Calculation 297
See also 298

Chapter 7: More Advanced Class Design 299
Introduction 299
Choosing between inheritance and extension – the is-a question 300
Getting ready 301
How to do it... 302
Wrapping – aggregation and composition 303
Extending - inheritance 305
How it works... 306
There's more... 307
See also 308
Separating concerns via multiple inheritance 309
Getting ready 309
How to do it... 310
How it works... 312
There's more... 313
See also 315
Leveraging Python's duck typing 315
Getting ready 316
How to do it... 316
How it works... 318
There's more... 319
See also 319
Managing global and singleton objects 320
Getting ready 320
How to do it... 321
Module global variable 321
Class-level static variable 323
How it works... 324
There's more... 325
Using more complex structures – maps of lists 325
Getting ready 326
How to do it... 326
How it works... 327
There's more... 328
See also 328
Creating a class that has orderable objects 329
Getting ready 329
How to do it... 330
How it works... 332
There's more... 333
See also 334
Defining an ordered collection 335
Getting ready 336
How to do it... 337
How it works... 339

There's more... 340
See also 342
Deleting from a list of mappings 342
Getting ready 343
How to do it... 345
How it works... 346
There's more... 347
See also 348

Chapter 8: Input/Output, Physical Format, and Logical Layout 349
Introduction 349
Using pathlib to work with filenames 351
Getting ready 352
How to do it... 353
Making the output filename by changing the input suffix 353
Making a number of sibling output files with distinct names 354
Creating a directory and a number of files 355
Comparing file dates to see which is newer 356
Removing a file 357
Finding all files that match a given pattern 358
How it works... 358
There's more... 359
See also 360
Reading and writing files with context managers 361
Getting ready 361
How to do it... 363
How it works... 364
There's more... 365
See also 365
Replacing a file while preserving the previous version 366
Getting ready 366
How to do it... 367
How it works... 369
There's more... 369
See also 370
Reading delimited files with the CSV module 370
Getting ready 371
How to do it... 372
How it works... 373
There's more... 374
See also 376
Reading complex formats using regular expressions 376
Getting ready 376
How to do it... 378
Defining the parse function 378
Using the parse function 379

How it works... 380
There's more... 381
See also 382
Reading JSON documents 382
Getting ready 383
How to do it... 385
How it works... 386
There's more... 386
Serializing a complex data structure 387
Deserializing a complex data structure 389
See also 390
Reading XML documents 390
Getting ready 391
How to do it... 394
How it works... 395
There's more... 395
See also 397
Reading HTML documents 397
Getting ready 398
How to do it... 400
How it works... 401
There's more... 403
See also 404
Upgrading CSV from DictReader to namedtuple reader 404
Getting ready 404
How to do it... 405
How it works... 407
There's more... 407
See also 409
Upgrading CSV from a DictReader to a namespace reader 410
Getting ready 410
How to do it... 411
How it works... 412
There's more... 412
See also 414
Using multiple contexts for reading and writing files 414
Getting ready 414
How to do it... 416
How it works... 418
There's more... 419
See also 420
Chapter 9: Testing 421
Introduction 421
Using docstrings for testing 422
Getting ready 423

How to do it... 424
Writing examples for stateless functions 425
Writing examples for stateful objects 425
How it works... 427
There's more... 428
See also 429
Testing functions that raise exceptions 430
Getting ready 430
How to do it... 431
How it works... 432
There's more... 433
See also 434
Handling common doctest issues 434
Getting ready 435
How to do it... 437
Writing doctest examples for mapping or set values 437
Writing doctest examples for floating-point values 438
How it works... 439
There's more... 439
See also 440
Creating separate test modules and packages 441
Getting ready 441
How to do it... 443
How it works... 444
There's more... 445
Some other assertions 447
Separate tests directory 448
See also 449
Combining unittest and doctest tests 449
Getting ready 449
How to do it... 451
How it works... 452
There's more... 453
See also 453
Testing things that involve dates or times 453
Getting ready 454
How to do it... 455
How it works... 457
There's more... 458
See also 460
Testing things that involve randomness 460
Getting ready 460
How to do it... 461
How it works... 462
There's more... 463
See also 464

Mocking external resources 464
Getting ready 465
Creating an entry document in the entrylog collection 466
Seeing a typical response 468
Client class for database access 469
How to do it... 471
How it works... 474
Creating a context manager 474
Creating a dynamic, stateful test 475
Mocking a complex object 476
Using the load_tests protocol 477
There's more... 478
See also 478

Chapter 10: Web Services 479
Introduction 479
Implementing web services with WSGI 482
Getting ready 482
How to do it... 487
How it works... 490
There's more... 491
See also 494
Using the Flask framework for RESTful APIs 495
Getting ready 495
How to do it... 496
How it works... 499
There's more... 500
See also 502
Parsing the query string in a request 502
Getting ready 504
How to do it... 504
How it works... 506
There's more... 507
See also 509
Making REST requests with urllib 509
Getting ready 509
How to do it... 511
How it works... 514
There's more... 515
The OpenAPI (Swagger) specification 516
Adding Swagger to the server 516
See also 518
Parsing the URL path 518
Getting ready 519
How to do it... 520
Server 520

Client 523
How it works... 525
Deck slicing 526
Client side 527
There's more... 528
Providing a Swagger specification 528
Using a Swagger specification 531
See also 531
Parsing a JSON request 531
Getting ready 532
How to do it... 533
Swagger specification 533
Server 536
Client 537
How it works... 539
There's more... 540
Location header 541
Additional resources 541
Query for a specific player 542
Exception handling 543
See also 544
Implementing authentication for web services 544
Getting ready 546
Configuring SSL 546
Users and credentials 547
Flask view function decorator 549
How to do it... 549
Defining the User class 550
Defining a view decorator 553
Creating the server 554
Creating an example client 556
How it works... 559
There's more... 560
Creating a command-line interface 562
Building the Authentication header 562
See also 563
Chapter 11: Application Integration 564
Introduction 564
Finding configuration files 565
Getting ready 566
Why so many choices? 567
How to do it... 568
How it works... 570
There's more... 570
See also 575
Using YAML for configuration files 575
Getting ready 575

How to do it... 576
How it works... 577
There's more... 579
See also 582
Using Python for configuration files 582
Getting ready 582
How to do it... 584
How it works... 584
There's more... 585
See also 586
Using class-as-namespace for configuration 586
Getting ready 587
How to do it... 588
How it works... 589
There's more... 589
Configuration representation 591
See also 592
Designing scripts for composition 593
Getting ready 593
How to do it... 595
How it works... 597
There's more... 599
Designing as a class hierarchy 600
See also 601
Using logging for control and audit output 602
Getting ready 602
How to do it... 606
How it works... 610
There's more... 612
Combining two applications into one 612
Getting ready 613
How to do it... 615
How it works... 616
There's more... 617
Refactoring 617
Concurrency 618
Logging 620
See also 621
Combining many applications using the Command design pattern 621
Getting ready 621
How to do it... 622
How it works... 623
There's more... 624
See also 626
Managing arguments and configuration in composite applications 626

Getting ready 627
How to do it... 628
How it works... 630
The Command design pattern 631
There's more... 632
See also 633
Wrapping and combining CLI applications 633
Getting ready 633
How to do it... 634
How it works... 635
There's more... 636
Unit test 637
See also 640
Wrapping a program and checking the output 640
Getting ready 640
How to do it... 641
How it works... 643
There's more... 644
See also 645
Controlling complex sequences of steps 645
Getting ready 645
How to do it... 646
How it works... 648
There's more... 649
Building conditional processing 650
See also 650
Index 651

Preface

Python is the preferred choice of developers, engineers, data scientists, and hobbyists everywhere. It is a great scripting language that can power your applications and provide great speed, safety, and scalability. By exposing Python as a series of simple recipes, you can gain insights into specific language features in a particular context. Having a tangible context helps make the language or standard library feature easier to understand.

This book takes a recipe-based approach, where each recipe addresses specific problems and issues.

What this book covers

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

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

`Chapter 3`, *Function Definitions*, will look at a number of function definition techniques. We'll also look at the Python 3.5 typing module and see how we can create more formal annotations for our functions.

`Chapter 4`, *Built-in Data Structures – list, set, dict*, will look at an overview of the various structures that are available and what problems they solve. From there, we can look at lists, dictionaries, and sets in detail, and also look at some more advanced topics related to how Python handles references to objects.

`Chapter 5`, *User Inputs and Outputs*, will explain how to use the different features of the print() function. We'll also look at the different functions used to provide user input.

`Chapter 6`, *Basics of Classes and Objects*, will create classes that implement a number of statistical formulae.

`Chapter 7`, *More Advanced Class Design*, will dive a little more deeply into Python classes. We will combine some features we have previously learned about to create more sophisticated objects.

`Chapter 8`, *Input/Output, Physical Format, Logical Layout*, will work with different file formats such as JSON, XML, and HTML.

`Chapter 9`, *Testing*, will give us a detailed description of the different testing frameworks used in Python.

`Chapter 10`, *Web Services*, will look at a number of recipes for creating RESTful web services and also serving static or dynamic content.

`Chapter 11`, *Application Integration*, will look at ways that we can design applications that can be composed to create larger, more sophisticated composite applications. We'll also look at the complications that can arise from composite applications and the need to centralize some features, such as command-line parsing.

`Chapter 12`, *Functional and Reactive Programming Features*, provides us with methods to writing small, expressive functions that perform the required data transformations. Moving ahead, you will learn about the idea of reactive programming, that is, having processing rules that are evaluated when the inputs become available or change. This chapter is not present in the book, it is available for download at `https://www.packtpub.com/sites/default/files/downloads/Functional_and_Reactive_Programming_Features.pdf`.

`Chapter 13`, *Statistical Programming and Linear Regression*, will look at some basic statistical calculations that we can do with Python's built-in libraries and data structures. We'll look at the questions of correlation, randomness, and the null hypothesis. This chapter is not present in the book, it is available for download at `https://www.packtpub.com/sites/default/files/downloads/Statistical_Programming_and_Linear_Regression.pdf`.

What you need for this book

All you need to follow through the examples in this book is a computer running any recent version of Python. While the examples all use Python 3, they can be adapted to work with Python 2 only a few changes.

Who this book is for

The book is for web developers, programmers, enterprise programmers, engineers, and big data scientists. If you are a beginner, Python Cookbook will get you started. If you are experienced, it will expand your knowledge base. A basic knowledge of programming would help.

Conventions

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:

```
>>> circumference_diameter_ratio = 355/113
>>> target_color_name = 'FireBrick'
>>> target_color_rgb = (178, 34, 34)
```

New terms and important words are shown in bold.

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on Code Download.

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

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

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

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the Errata section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Numbers, Strings, and Tuples

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

- Creating meaningful names and using variables
- Working with large and small integers
- Choosing between float, decimal, and fraction
- Choosing between true division and floor division
- Rewriting an immutable string
- String parsing with regular expressions
- Building complex strings with "template".format()
- Building complex strings from lists of characters
- 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

Introduction

This chapter will look at some central types of Python objects. We'll look at the different kinds of numbers, working with strings, and using tuples. We'll look at these first because they're the simplest kinds of data Python works with. In later chapters, we'll look at data collections.

Most of these recipes assume a beginner's level of understanding of Python 3. We'll be looking at how we use the essential built-in types available in Python—numbers, strings, and tuples. Python has a rich variety of numbers, and two different division operators, so we'll need to look closely at the choices available to us.

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—as used by Python. 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 sometimes called the **read-eval-print loop** (**REPL**). In later chapters, we'll look more closely at writing script files. The goal is to encourage interactive exploration because it's a great way to learn the language.

Creating meaningful names and using variables

How can we be sure our programs make sense? One of the core elements of making expressive code is to use *meaningful* names. But what counts as meaningful? In this recipe, we'll review some common rules for creating meaningful Python names.

We'll also look at some of Python's assignment statement variations. We can, for example, assign more than one variable in a single statement.

Getting ready

The core issue when creating a name is to ask ourselves the question *what is this thing?* For software, we'd like a name that's descriptive of the object being named. Clearly, a name like `x` is not very descriptive, it doesn't seem to refer to an actual thing.

Vague, non-descriptive names are distressingly common in some programming. It's not helpful to others when we use them. A descriptive name helps everyone.

When naming things, it's also important to separate the problem domain—what we're really trying to accomplish—from the solution domain. The solution domain consists of the technical details of Python, OS, and Internet. Anyone who reads the code can see the solution; it doesn't require deep explanation. The problem domain, however, can be obscured by technical details. It's our job to make the problem clearly visible. Well-chosen names will help.

How to do it...

We'll look at names first. Then we'll move on to assignment.

Choosing names wisely

On a purely technical level, Python names must begin with a letter. They can include any number of letters, digits, and the _ character. Python 3 is based on Unicode, so a letter is not limited to the Latin alphabet. While the A-Z Latin alphabet is commonly used, it's not required.

When creating a descriptive variable, we want to create names that are both specific and articulate the relationships among things in our programs. One widely used technique is to create longer names in a style that moves from particular to general.

The steps to choosing a name are as follows:

1. The last part of the name is a very broad summary of the thing. In a few cases, this may be all we need; context will supply the rest. We'll suggest some typical broad summary categories later.
2. Use a prefix to narrow this name around your application or problem domain.
3. If needed, put more narrow and specialized prefixes on this name to clarify how it's distinct from other classes, modules, packages, functions, and other objects. When in doubt about prefixing, remember how domain names work. Think of `mail.google.com`—the name flows from particular to general. There's no magic about the three levels of naming, but it often happens to work out that way.
4. Format the name depending on how it's used in Python. There are three broad classes of things we'll put names on, which are shown as follows:
    - **Classes**: A class has a name that summarizes the objects that are part of the class. These names will (often) use `CapitalizedCamelCase`. The first letter of a class name is capitalized to emphasize that it's a class, not an instance of the class. A class is often a generic concept, rarely a description of a tangible thing.
    - **Objects**: A name for an object usually uses `snake_case` - all lowercase with multiple _ characters between words. In Python, this includes variables, functions, modules, packages, parameters, attributes of objects, methods of classes, and almost everything else.

- **Script and module files**: These are really the OS resources, as seen by Python. Therefore, a filename should follow the conventions for Python objects, using letters, the _ characters and ending with the `.py` extension. It's technically possible to have pretty wild and free filenames. Filenames that don't follow Python rules can be difficult to use as a module or package.

How do we choose the broad category part of a name? The general category depends on whether we're talking about a thing or a property of a thing. While the world is full of things, we can create some board groupings that are helpful. Some of the examples are Document, Enterprise, Place, Program, Product, Process, Person, Asset, Rule, Condition, Plant, Animal, Mineral, and so on.

We can then narrow these with qualifiers:

```
FinalStatusDocument
ReceivedInventoryItemName
```

The first example is a class called `Document`. We've narrowed it slightly by adding a prefix to call it a `StatusDocument`. We narrowed it even further by calling it a `FinalStatusDocument`. The second example is a `Name` that we narrowed by specifying that it's a `ReceivedInventoryItemName`. This example required a four-level name to clarify the class.

An object often has properties or attributes. These have a decomposition based in the kind of information that's being represented. Some examples of terms that should be part of a complete name are amount, code, identifier, name, text, date, time, datetime, picture, video, sound, graphic, value, rate, percent, measure, and so on.

The idea is to put the narrow, more detailed description first, and the broad kind of information last:

```
measured_height_value
estimated_weight_value
scheduled_delivery_date
location_code
```

In the first example, `height` narrows a more general representation term `value`. And `measured_height_value` further narrows this. Given this name, we can expect to see other variations on height. Similar thinking applies to `weight_value`, `delivery_date` and `location_code`. Each of these has a narrowing prefix or two.

Some things to avoid: Don't include detailed technical type information using coded prefixes or suffixes. This is often called **Hungarian Notation**; we don't use `f_measured_height_value` where the `f` is supposed to mean a floating-point. A variable like `measured_height_value` can be any numeric type and Python will do all the necessary conversions. The technical decoration doesn't offer much help to someone reading our code, because the type specification can be misleading or even incorrect. Don't waste a lot of effort forcing names to look like they belong together. We don't need to make `SpadesCardSuit`, `ClubsCardSuit`, and so on. Python has many different kinds of namespaces, including packages, modules, and classes, as well as namespace objects to gather related names together. If you combine these names in a `CardSuit` class, you can use `CardSuit.Spades`, which uses the class as namespace to separate these names from other, similar names.

Assigning names to objects

Python doesn't use static variable definitions. A variable is created when a name is assigned to an object. It's important to think of the objects as central to our processing, and variables as little more than sticky notes that identify an object. Here's how we use the fundamental assignment statement:

1. Create an object. In many of the examples we'll create objects as literals. We'll use `355` or `113` as literal representations of integer objects in Python. We might use a string like `FireBrick` or a tuple like `(178, 34, 34)`.
2. Write the following kind of statement: *variable = object*. Here are some examples:

```
>>> circumference_diameter_ratio = 355/113
>>> target_color_name = 'FireBrick'
>>> target_color_rgb = (178, 34, 34)
```

We've created some objects and assigned them to variables. The first object is the result of a calculation. The next two objects are simple literals. Generally, objects are created by expressions that involve functions or classes.

This basic statement isn't the only kind of assignment. We can assign a single object to multiple variables using a kind of duplicated assignment like this:

```
>>> target_color_name = first_color_name = 'FireBrick'
```

This creates two names for the same string object. We can confirm this by checking the internal ID values that Python uses:

```
>>> id(target_color_name) == id(first_color_name)
True
```

This comparison shows us that the internal identifiers for these two objects are the same.

A test for equality uses ==. Simple assignment uses =.

When we look at numbers and collections, we'll see that we can combine assignment with an operator. We can do things like this:

```
>>> total_count = 0
>>> total_count += 5
>>> total_count += 6
>>> total_count
11
```

We've augmented assignment with an operator. `total_count += 5` is the same as `total_count = total_count + 5`. This technique has the advantage of being shorter.

How it works...

This approach to creating names follows the pattern of using narrow, more specific qualifiers first and the wider, less-specific category last. This follows the common convention used for domain names and e-mail addresses.

For example, a domain name like `mail.google.com` has a specific service, a more general enterprise, and finally a very general domain. This follows the principle of narrow-to-wider.

As another example, `service@packtpub.com` starts with a specific destination name, has a more general enterprise, and finally a very general domain. Even the name of destination (*PacktPub*) is a two-part name with a narrow enterprise name (*Packt*) followed by a wider industry (*Pub*, short for *publishing*). (*We don't agree with those who suggest it stands for Public House.*)

The assignment statement is the only way to put a name on an object. We noted that we can have two names for the same underlying object. This isn't too useful right now. But in `Chapter 4`, *Built-in Data Structures – list, set, dict* we'll see some interesting consequences of multiple names for a single object.

There's more...

We'll try to show descriptive names in all of the recipes.

We have to grant exceptions to existing software which doesn't follow this pattern. It's often better to be consistent with legacy software than impose new rules even if the new rules are better.

Almost every example will involve assignment to variables. It's central to stateful object-oriented programming.

We'll look at classes and class names in `Chapter 6`, *Basics of Classes and Objects*; we'll look at modules in `Chapter 11`, *Application Integration*.

See also

The subject of descriptive naming is a source of ongoing research and discussion. There are two aspects—syntax and semantics. The starting point for thoughts on Python syntax is the famous **Python Enhancement Proposal number 8** (**PEP-8**). This leads to use of `CamelCase`, and `snake_case` names.

Also, be sure to do this:

```
>>> import this
```

This will provide more insight into Python ideals.

For information on semantics, look at the legacy UDEF and NIEM Naming and Design Rules standards (`http://www.opengroup.org/udefinfo/AboutTheUDEF.pdf`). Additional details are in ISO11179 (`https://en.wikipedia.org/wiki/ISO/IEC_11179`), which talks in detail about meta-data and naming.

Working with large and small integers

Many programming languages make a distinction between integers, bytes, and long integers. Some languages include distinctions for *signed versus unsigned* integers. How do we map these concepts to Python?

The easy answer is that we don't. Python handles integers of all sizes in a uniform way. From bytes to immense numbers with hundreds of digits, it's all just integers to Python.

Getting ready

Imagine you need to calculate something really big. For example, calculate the number of ways to permute the cards in a 52-card deck. The number 52! = 52 × 51 × 50 × ... × 2 × 1, is a very, very large number. Can we do this in Python?

How to do it...

Don't worry. Really. Python behaves as if it has one universal type of integer, and this covers all of the bases from bytes to numbers that fill all of the memory. Here are the steps to using integers properly:

1. Write the numbers you need. Here are some smallish numbers: 355, 113. There's no practical upper limit.

2. Creating a very small value—a single byte—looks like this:

```
>>> 2
2
```

   Or perhaps this, if you want to use base 16:

```
>>> 0xff
255
```

   In later recipes, we'll look at a sequence of bytes that has only a single value in it:

```
>>> b'\xfe'
b'\xfe'
```

   This isn't—technically—an integer. It has a prefix of `b'` that shows us it's a 1-byte sequence.

3. Creating a much, much bigger number with a calculation might look like this:

```
>>> 2**2048
323...656
```

This number has 617 digits. We didn't show all of them.

How it works...

Internally, Python uses two kinds of numbers. The conversion between these two is seamless and automatic.

For smallish numbers, Python will generally use 4 or 8 byte integer values. Details are buried in CPython's internals, and depend on the facilities of the C-compiler used to build Python.

For largish numbers, over `sys.maxsize`, Python switches to large integer numbers which are sequences of digits. Digit, in this case, often means a 30-bit value.

How many ways can we permute a standard deck of 52 cards? The answer is $52! \approx 8 \times 10^{67}$. Here's how we can compute that large number. We'll use the factorial function in the `math` module, shown as follows:

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

Yes, these giant numbers work perfectly.

The first parts of our calculation of 52! (from $52 \times 51 \times 50 \times$ ... down to about 42) could be performed entirely using the smallish integers. After that, the rest of the calculation had to switch to largish integers. We don't see the switch; we only see the results.

For some of the details on the internals of integers, we can look at this:

```
>>> import sys
>>> import math
>>> math.log(sys.maxsize, 2)
63.0
>>> sys.int_info
sys.int_info(bits_per_digit=30, sizeof_digit=4)
```

The `sys.maxsize` value is the largest of the small integer values. We computed the log to base 2 to find out how many bits are required for this number.

This tells us that our Python uses 63-bit values for small integers. The range of smallish integers is from -2^{64} ... $2^{63} - 1$. Outside this range, largish integers are used.

The values in `sys.int_info` tells us that large integers are a sequence of numbers that use 30-bit digits, and each of these digits occupies 4 bytes.

A large value like 52! consists of 8 of these 30-bit-sized digits. It can be a little confusing to think of a digit as requiring 30 bits to represent. Instead of 10 symbols used to represent base 10 numbers, we'd need *2**30* distinct symbols for each digit of these large numbers.

A calculation involving a number of big integer values can consume a fair bit of memory. What about small numbers? How can Python manage to keep track of lots of little numbers like one and zero?

For the commonly used numbers (-5 to 256) Python actually creates a secret pool of objects to optimize memory management. You can see this when you check the `id()` value for integer objects:

```
>>> id(1)
4297537952
>>> id(2)
4297537984
>>> a=1+1
>>> id(a)
4297537984
```

We've shown the internal `id` for the integer `1` and the integer `2`. When we calculate a value, the resulting object turns out to be the same integer `2` object that was found in the pool.

When you try this, your `id()` values may be different. However, every time the value of `2` is used, it will be the same object; on the author's laptop, it's id = `4297537984`. This saves having many, many copies of the 2 object cluttering up memory.

Here's a little trick for seeing exactly how huge a number is:

```
>>> len(str(2**2048))
617
```

We created a string from a calculated number. Then we asked what the length of the string was. The response tells us that the number had 617 digits.

There's more...

Python offers us a broad set of arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, and `**`. The `/` and `//` are for division; we'll look at these in a separate recipe named *Choosing between true division and floor division*. The `**` raises a number to a power.

For dealing with individual bits, we have some additional operations. We can use `&`, `^`, `|`, `<<`, and `>>`. These operators work bit-by-bit on the internal binary representations of integers. These compute a binary **AND**, a binary **Exclusive OR**, I**nclusive OR**, **Left Shift**, and **Right Shift** respectively.

While these will work on very big integers, they don't really make much sense outside the world of individual bytes. Some binary files and network protocols will involve looking at the bits within an individual byte of data.

We can play around with these operators by using the `bin()` function to see what's going on.

Here's a quick example of what we mean:

```
>>> xor = 0b0011 ^ 0b0101
>>> bin(xor)
'0b110'
```

We've used `0b0011` and `0b0101` as our two strings of bits. This helps to clarify precisely what the two numbers have as their binary representation. We applied the exclusive or (`^`) operator to these two sequences of bits. We used the `bin()` function to see the result as a string of bits. We can carefully line up the bits to see what the operator did.

We can decompose a byte into portions. Say we want to separate the left-most two bits from the other six bits. One way to do this is with bit-fiddling expressions like these:

```
>>> composite_byte = 0b01101100
>>> bottom_6_mask =  0b00111111
>>> bin(composite_byte >> 6)
'0b1'
>>> bin(composite_byte & bottom_6_mask)
'0b101100'
```

We've defined a composite byte which has `01` in the most significant two bits, and `101100` in the least significant six bits. We used the `>>` shift operator to shift the value by six positions, removing the least significant bits and preserving the two most significant bits. We used the `&` operator with a mask. Where the mask has 1 bit, a position's value is preserved in the result, where a mask has `0` bits, the result position is set to `0`.

See also

- We'll look at the two division operators in the *Choosing between true division and floor division* recipe
- We'll look at other kinds of numbers in the *Choosing between float, decimal, and fraction* recipe
- For details on integer processing, see `https://www.python.org/dev/peps/pep-0237/`

Choosing between float, decimal, and fraction

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

- Float
- Decimal
- Fraction

With so many choices, when do we use each of these?

Getting ready

It's important to be sure about our core mathematical expectations. If we're not sure what kind of data we have, or what kinds of results we want to get, we really shouldn't be coding. We need to take a step back and review things with pencil and paper.

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

1. **Currency**: Dollars, cents, or euros. Currency generally has a fixed number of decimal places. There are rounding rules used to determine what 7.25% of $2.95 is.
2. **Rational Numbers or Fractions**: When we're working with American units for feet and inches, or cooking measurements in cups and fluid ounces, we often need to work in 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 5/8 . How do we apply this to 2/3 cup of rice and still get a measurement that fits an American kitchen gadget?

3. **Irrational Numbers**: This includes all other kinds of calculations. It's important to note that digital computers can only approximate these numbers, and we'll occasionally see odd little artifacts of this approximation. The float approximations are very fast, but sometimes suffer from truncation issues.

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. First, we'll look at computing with currency. Then we'll look at rational numbers, and finally irrational or floating-point numbers. Finally, we'll look at making explicit conversions among these various types.

Doing currency calculations

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

1. To work with currency, we'll do this. Import the `Decimal` class from the `decimal` module:

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

2. Create `Decimal` objects from strings or integers:

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

   We created the `tax_rate` from two `Decimal` objects. One was based on a string, the other based on an integer. We could have used `Decimal('0.0725')` instead of doing the division explicitly.

   The result is a hair over $0.21. It's computed out correctly to the full number of decimal places.

3. If you try to create decimal objects from floating-point values, you'll see unpleasant artifacts of float approximations. Avoid mixing `Decimal` and `float`. To round to the nearest penny, create a `penny` object:

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

4. Quantize your data using this penny object:

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

This shows how we can use the default rounding rule of `ROUND_HALF_EVEN`.

Every financial wizard has a different style of rounding. The `Decimal` module offers every variation. 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. This provides us handy rational numbers that we can use. 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. If you create fraction objects from floating-point values, you may see unpleasant artifacts of float approximations. When the denominator is a power of 2, things can work out exactly:

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

We created one fraction from a string, `2.5`. We created the second fraction from a floating-point calculation, `5/8`. Because the denominator is a power of 2, this works out exactly.

The result, 25/16, is a complex-looking fraction. What's a nearby fraction that might be simpler?

```
>>> Fraction(24,16)
Fraction(3, 2)
```

We can see that we'll use almost a cup and a half to scale the recipe for five people instead of eight.

Floating-point approximations

Python's built-in `float` type is capable of representing a wide variety of values. The trade-off here is that float often involves an approximation. In some cases—specifically when doing division that involves powers of 2—it can be as exact as a `fraction`. In all other cases, there may be small discrepancies that reveal 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. Recognize that all calculations are an approximation:

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

2. Mathematically, the value should be `1`. Because of the approximations used for `float`, the answer isn't exact. It's not wrong by much, but it's wrong. When we round appropriately, the value is more useful:

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

3. Know the error term. In this case, we know what the exact answer is supposed to be, so we can compare our calculation with the known correct answer. This gives us the general error value that can creep into floating-point numbers:

   ```
   >>> 1-answer
   1.1102230246251565e-16
   ```

For most floating-point errors, this is the typical value—about 10^{-16}. Python has clever rules that hide this error some of the time by doing some automatic rounding. For this calculation, however, the error wasn't hidden.

This is a very important consequence.

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

When we see code that uses an exact == test between floating-point numbers, there are going to be problems when the approximations differ by a single bit.

Converting numbers from one type to another

We can use the `float()` function to create a `float` value from another value. It looks like this:

```
>>> float(total_amount)
3.163875
>>> float(sugar_cups * scale_factor)
1.5625
```

In the first example, we converted a `Decimal` value to `float`. In the second example, we converted a `Fraction` value to `float`.

As we just saw, we're never happy trying to convert `float` to `Decimal` or `Fraction`:

```
>>> Fraction(19/155)
Fraction(8832866365939553, 72057594037927936)
>>> Decimal(19/155)
Decimal('0.122580645161290316402791233940661186352372169494628906 25')
```

In the first example, we did a calculation among integers to create a `float` value that has a known truncation problem. When we created a `Fraction` from that truncated `float` value, we got some terrible looking numbers that exposed the details of the truncation.

Similarly, the second example tried to create a `Decimal` value from a `float`.

How it works...

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

Python is adept at converting numbers between the various 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` and the results will be `Fractions`. We can also mix `int` and `Decimal`. We cannot casually mix `Decimal` with `float` or `Fraction`; we need to provide explicit conversions.

It's important to note that `float` values are really approximations. The Python syntax allows us to write numbers as decimal values; that's not how they're processed internally.

We can write a value like this in Python, using ordinary base-10 values:

```
>>> 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, `8.066e+67`, is this:

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

The numerator is a big number, `6737037547376141`. The denominator is always 2^{53}. Since the denominator is fixed, the resulting fraction can only have 53 meaningful bits of data. Since more bits aren't available, values might get truncated. This leads to tiny discrepancies between our idealized abstraction and actual numbers. The exponent (2^{226}) is required to scale the fraction up to the proper range.

Mathematically, $6737037547376141 * 2^{226}/2^{53}$.

We can use `math.frexp()` 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** 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.

The error we noticed earlier matches this quite nicely: $10^{-16} \approx 2^{-53}$.

Unlike the built-in `float`, a `Fraction` is an exact ratio of two integer values. As we saw in the *Working with large and small integers* recipe, integers in Python can be very large. We can create ratios which involve integers with a large number of digits. We're not limited by a fixed denominator.

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

Why use floating-point? Two reasons: Not all computable numbers can be represented as fractions. That's why mathematicians introduced (or perhaps discovered) irrational numbers. The built-in float type is as close as we can get to the mathematical abstraction of irrational numbers. A value like √2, for example, can't be represented as a fraction. Also, float values are very fast.

There's more...

The Python `math` module contains a number of specialized functions for working with floating-point values. This module includes common functions such as square root, logarithms, and various trigonometry functions. It 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 to see if they're nearly equal:

```
>>> (19/155)*(155/19) == 1.0
False
>>> math.isclose((19/155)*(155/19), 1)
True
```

This function provides us with a way to compare floating-point numbers meaningfully.

Python also offers complex data. This involves a real and an imaginary part. In Python, we write `3.14+2.78j` to represent the complex number 3.14 + 2.78 √-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 a `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):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
>>> cmath.sqrt(-2)
1.4142135623730951j
```

This is essential when working with complex numbers.

See also

- We'll talk more about floating point and fractions in the *Choosing between true division and floor division* recipe
- See `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 doing division:

- A *div-mod* pair: We want two parts—the quotient and the remainder. We often use this when converting values from one base to another. When we convert seconds to 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, the remainder will be converted to 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 the three cases separately. First we'll look at truncated floor division. Then we'll look at true floating-point division. Finally, we'll look at division of fractions.

Doing floor division

When we are doing the *div-mod* kind of calculations, we might use floor division, `//`, and modulus, `%`. Or, we might use the `divmod()` function.

1. We'll divide the number of seconds by 3600 to get the value of `hours`; the modulus, or remainder, can be converted separately to `minutes` and `seconds`:

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

2. Again, using remaining values, we'll divide the number of seconds by 60 to get `minutes`; the remainder is a 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:

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

A true value calculation gives as a floating-point approximation. For example, about how many hours is 7386 seconds? Divide 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 for using floating-point values, we rounded the result to avoid having to look at tiny error values.

This true division is a feature of Python 3. We'll look at this from a Python 2 perspective in the next sections.

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)
```

3. If necessary, convert the exact fraction to 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 be fractions; this promotion means that the math is done as exactly as possible.

How it works...

Python 3 has two division operators.

- The `/` true division operator always tries to produce 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 try to preserve the type of the data. The true division operation - when applied to integers - produces a `float` result.

- The `//` truncated division operator always tries to produce a truncated result. For two integer operands, this is the truncated quotient. For two floating-point operands, this is a truncated floating-point result:

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

By default, Python 2 only has one division operator. For programmers still using Python 2, we can start using these new division operators with this:

```
>>> from __future__ import division
```

This import will install the Python 3 division rules.

See also

- For more on the choice between floating-point and fractions, see the *Choosing between float, decimal, and fraction* recipe
- See `https://www.python.org/dev/peps/pep-0238/`

Rewriting an immutable string

How can we rewrite an immutable string? We can't change individual characters inside a string:

```
>>> title = "Recipe 5: Rewriting, and the Immutable String"
>>> title[8]= ''
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Since this doesn't work, how do we make a change to a string?

Getting ready

Let's assume we have a string like this:

```
>>> title = "Recipe 5: Rewriting, and the Immutable String"
```

We'd like to do two transformations:

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

Since we can't replace characters in a string object, we have to work out some alternatives. There are several common things we can do, shown as follows:

- A combination of slicing and concatenating a string to create a new string.
- When shortening, we often use the `partition()` method.
- We can replace a character or a substring with the `replace()` method.
- We can expand the string into a list of characters, then join the string back into a single string again. This is the subject for a separate recipe, *Building complex strings with a list of characters*.

How to do it...

Since we can't update a string in place, we have to replace the string variable's object with each modified result. We'll use a statement that looks like this:

```
some_string = some_string.method()
```

Or we could even use:

```
some_string = some_string[:chop_here]
```

We'll look at a number of specific variations on this general theme. We'll slice a piece of a string, we'll replace individual characters within a string, and we'll apply blanket transformations such as making the string lowercase. We'll also look at ways to remove extra _ that show up in our final string.

Slicing a piece of a string

Here's how we can shorten a string via slicing:

1. Find the boundary:

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

   The index function locates a particular substring and returns the position where that substring can be found. If the substring doesn't exist, it raises an exception. This is always `true` of the result `title[colon_position] == ':'`.

2. Pick the substring:

```
      >>> discard_text, post_colon_text = title[:colon_position],
title[colon_position+1:]
      >>> discard_text
      'Recipe 5'
      >>> post_colon_text
      ' Rewriting, and the Immutable String'
```

We've used the slicing notation to show the `start:end` of the characters to pick. We also used multiple assignment to assign two variables, `discard_text` and `post_colon_text`, from two expressions.

We can use `partition()` as well as manual slicing. Find the boundary and partition:

```
>>> pre_colon_text, _, post_colon_text = title.partition(':')
>>> pre_colon_text
'Recipe 5'
>>> post_colon_text
' Rewriting, and the Immutable String'
```

The `partition` function returns three things: the part before the target, the target, and the part after the target. We used multiple assignment to assign each object to a different variable. We assigned the target to a variable named _ because we're going to ignore that part of the result. This is a common idiom for places where we must provide a variable, but we don't care about using the object.

Updating a string with a replacement

We can use `replace()` to remove punctuation marks. When using `replace` to switch punctuation marks, save the results back into the original variable. In this case, `post_colon_text`:

```
>>> post_colon_text = post_colon_text.replace(' ', '_')
>>> post_colon_text = post_colon_text.replace(',', '_')
>>> post_colon_text
'_Rewriting__and_the_Immutable_String'
```

This has replaced the two kinds of punctuation with the desired _ characters. We can generalize this to work with all punctuation. This leverages the `for` statement, which we'll look at in `Chapter 2`, *Statements and Syntax*.

We can iterate through all punctuation characters:

```
>>> from string import whitespace, punctuation
>>> for character in whitespace + punctuation:
...     post_colon_text = post_colon_text.replace(character, '_')
>>> post_colon_text
'_Rewriting__and_the_Immutable_String'
```

As each kind of punctuation character is replaced, we assign the latest and greatest version of the string to the `post_colon_text` variable.

Making a string all lowercase

Another transformational step is changing a string to all lowercase. As with the previous examples, we'll assign the results back to the original variable. Use the `lower()` method, assigning the result to the original variable:

```
>>> post_colon_text = post_colon_text.lower()
```

Removing extra punctuation marks

In many cases, there are some additional steps we might follow. We often want to remove leading and trailing _ characters. We can use `strip()` for this:

```
>>> post_colon_text = post_colon_text.strip('_')
```

In some cases, we'll have multiple _ characters because we had multiple punctuation marks. The final step would be something like this to cleanup up multiple _ characters:

```
>>> while '__' in post_colon_text:
...     post_colon_text = post_colon_text.replace('__', '_')
```

This is yet another example of the same pattern we've been using to modify a string in place. This depends on the `while` statement, which we'll look at in `Chapter 2`, *Statements and Syntax*.

How it works...

We can't—technically—modify a string in place. The data structure for a string is immutable. However, we can assign a new string back to the original variable. This technique behaves the same as modifying a string in place.

When a variable's value is replaced, the previous value no longer has any references and is garbage collected. We can see this by using the `id()` function to track each individual string object:

```
>>> id(post_colon_text)
4346207968
>>> post_colon_text = post_colon_text.replace('_','-')
>>> id(post_colon_text)
4346205488
```

Your actual id numbers may be different. What's important is that the original string object assigned to `post_colon_text` had one id. The new string object assigned to `post_colon_text` has a different id. It's a new string object.

When the old string has no more references, it is removed from memory automatically.

We made use of **slice notation** to decompose a string. A slice has two parts: `[start:end]`. A slice always includes the starting index. String indices always start with zero as the first item. It never includes the ending index.

The items in a slice have an index from `start` to `end-1`. This is sometimes called a **half-open** interval.

Think of a slice like this: all characters where the index, *i*, are in the range $start \le i < end$.

We noted briefly that we can omit the start or end indices. We can actually omit both. Here are the various options available:

- `title[colon_position]`: A single item, the `:` we found using `title.index(':')`.
- `title[:colon_position]`: A slice with the start omitted. It begins at the first position, index of zero.
- `title[colon_position+1:]`: A slice with the end omitted. It ends at the end of the string, as if we said `len(title)`.
- `title[:]`: Since both start and end are omitted, this is the entire string. Actually, it's a *copy* of the entire string. This is the quick and easy way to duplicate a string.

There's more...

There are more features to indexing in Python collections like a string. The normal indices start with 0 at the left end. We have an alternate set of indices using negative names that work from the right end of a string.

- `title[-1]` is the last character in the title, `g`
- `title[-2]` is the next-to-last character, `n`
- `title[-6:]` is the last six characters, `String`

We have a lot of ways to pick pieces and parts out of a string.

Python offers dozens of methods for modifying a string. *Section 4.7* of the *Python Standard Library* describes the different kinds of transformations that are available to us. There are three broad categories of string methods. We can ask about a string, we can parse a string, and we can transform a string. Methods such as `isnumeric()` tell us if a string is all digits.

Here's an example:

```
>>> 'some word'.isnumeric()
False
>>> '1298'.isnumeric()
True
```

We've looked at parsing with the `partition()` method. And we've looked at transforming with the `lower()` method.

See also

- We'll look at the string as list technique for modifying a string in the *Building complex strings from lists of characters* recipe.
- Sometimes we have data that's only a stream of bytes. In order to make sense of it, we need to convert it into characters. That's the subject for the *Decoding bytes – how to get proper characters from some bytes* recipe.

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 can't use 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, amount is represented as digits. Import the `re` module:

   ```
   >>> import re
   ```

3. Rewrite the pattern into **Regular Expression** (**RE**) notation:

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

We've replaced representation hints such as *words* with `\w+`. We've replaced *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 the regular expression notation, a colon matches itself.

For each of the fields of data, we've used `?P<name>` to provide a name that identifies the data we want to extract. We didn't do this around the colon or the spaces because we don't want those characters.

REs use a lot of `\` characters. To make this work out nicely in Python, we almost always use *raw* strings. The `r'` prefix 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 input text. If the input matches the pattern, we'll get a match object that shows details of the matching:

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

This, by itself, is pretty cool: we have a tuple of the different fields within the string. We'll return to the use of tuples in a recipe named *Using tuples*.

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

```
>>> match.group('ingredient')
'Kumquat'
>>> match.group('amount')
'2'
>>> match.group('unit')
'cups'
```

Each group is identified by the name we used in the `(?P<name>...)` part of the RE.

How it works...

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

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 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 preceeding 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 slightly different meaning. We saw it in `(?P<name>...)` where it was inside the `()` to define special properties for the grouping.
- The `.` 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 with the name of `name`.

It uses `\s*` to match an optional sequence of spaces.

It matches any character in the set `[=:]`. 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+' # 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.

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 complex strings with "template".format()

Creating complex strings is, in many ways, the polar opposite of parsing a complex string. We generally find that we'll 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 including 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 a template string from the result, replacing all of the data items with `{}` placeholders. Inside each placeholder, put the name of the data item.

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

2. For each data item, append `:data type` information to the placeholders 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:

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

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 assures 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 this: `5.2f` to specify the total length of five characters with two to the right of the decimal point. Here's the whole format:

```
      '{id:3d}  : {location:19s} : {max_temp:3d} / {min_temp:3d} /
{precipitation:5.2f}'
```

4. Use the `format()` method of this string to create the final string:

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

We've provided all of the variables by name in the `format()` method of the template string. This can get tedious. In some cases, we might want to build a dictionary object with the variables. In that case, we can use the `format_map()` method:

```
>>> 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'll return to dictionaries in `Chapter 4`, *Build-in Data Structures – list, set, dict*.

The built-in `vars()` function builds a dictionary of all of the local variables for us:

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

The `vars()` function is very handy for building a dictionary automatically.

How it works...

The string `format()` and `format_map()` methods can do a lot of relatively sophisticated string assembly for us.

The basic feature is to interpolate data into a string based on names of keyword arguments or keys in a dictionary. Variables can also be interpolated by position—we can provide position numbers instead of names. We can use a format specification like `{0:3s}` to use the first positional argument to `format()`.

We've seen three of the formatting conversions—`s`, `d`, `f`—there are many others. Details are in *Section 6.1.3* of the *Python Standard Library*. 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 to a character. Often, we use hexadecimal numbers for this so you might want to try values such as `0x2661` through `0x2666` for fun.
- `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. 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 a thousand separators defined. For more information, see the `locale` module.
- `o` is for octal, base 8.
- `s` is for string.
- `X` and `x` is 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 includes the `%`.

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. 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 must use `{value: 5.3f}`. The space makes sure that room is left for the sign, assuring 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}`, `{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)`
- `{name!a}` shows the ASCII value that would be produced by `ascii(name)`

In `Chapter 6`, *Basics of Classes and Objects*, we'll leverage the idea of the `{name!r}` format specification to simplify displaying information about related objects.

There's more...

A handy debugging hack this:

```
print("some_variable={some_variable!r}".format_map(vars()))
```

The `vars()` function—with no arguments—collects all of the local variables into a mapping. We provide that mapping for `format_map()`. The format template can use lots of `{variable_name!r}` to display details about various objects we have in local variables.

Inside a class definition we can use techniques such as `vars(self)`. This looks forward to `Chapter 6`, *Basics of Classes and Objects*:

```
>>> class Summary:
...     def __init__(self, id, location, min_temp, max_temp,
precipitation):
...         self.id= id
...         self.location= location
...         self.min_temp= min_temp
...         self.max_temp= max_temp
...         self.precipitation= precipitation
...     def __str__(self):
...         return '{id:3s}  : {location:19s} :  {max_temp:3d} /
{min_temp:3d} / {precipitation:5.2f}'.format_map(
...             vars(self)
...         )
>>> s= Summary('IAD', 'Dulles Intl Airport', 13, 32, 0.4)
>>> print(s)
IAD  : Dulles Intl Airport :   32 /  13 /  0.40
```

Our class definition includes a `__str__()` method. This method relies on `vars(self)` to create a useful dictionary of just the attribute of the object.

See also

- The *Python Standard Library, Section 6.1.3* has all of the details on the format method of a string

Building complex strings from lists of characters

How can we make very complex 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. We'll look at using a list object. This will dovetail with some of the recipes in `Chapter 4`, *Built-in Data Structures – list, set, dict*.

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 the `:`
- 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 common whitespace characters, including space and tab
- `string.punctuation` lists the common ASCII punctuation marks. Unicode has a larger list of punctuation marks; that's also available based on your locale settings

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`, *Built-in Data Structures – list, set, dict*.

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 for a list. It locates the position with the given value:

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

3. Delete the characters no longer needed. The `del` statement can remove items from a list. Lists are a mutable data structures:

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

   We don't need to carefully work with the useful piece of the original string. We can remove items from a list.

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 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 removed from memory. 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 do 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.

Strings provide a number of features that lists don't. Conversely, strings provide a number of features a list doesn't have. We can't convert a list to lowercase the way we can convert a string.

There's an important trade-off here:

- Strings are immutable, that 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 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: `list[-1]` is the last item in a list object.

We'll return to mutable data structures in `Chapter 4`, *Built-in Data Structures – list, set, dict*.

There's more

Once we've started working with a list of characters instead of a string, we no longer have the string processing methods. We do have a number of list-processing techniques available to us. In addition to being able to delete items from a list, we can append an item, extend a list with another list, and insert a character into the list.

We can also change our viewpoint slightly, and look at a list of strings instead of a list of characters. The technique of doing `''.join(list)` will work when we have a list of strings as well as a list of characters. For example, we might do this:

```
>>> title_list.insert(0, 'prefix')
>>> ''.join(title_list)
'prefix_Rewriting_an_Immutable_String'
```

Our `title_list` object will be mutated into a list that contains a six-character string, prefix, plus 30 individual characters.

See also

- We can also work with strings using the internal methods of a string. See the *Rewriting an immutable string* recipe for more techniques.
- 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.

Using the Unicode characters that aren't on our keyboards

A big keyboard might have almost 100 individual keys. Fewer than 50 of these 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—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 million characters from the world's alphabets. And there are icons, emoticons, and dingbats galore in our computer fonts. How do we get to all of those glyphs?

Getting ready

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

We can see all the available characters at
`https://en.wikipedia.org/wiki/List_of_Unicode_characters and also`
`http://www.unicode.org/charts/.`

We'll need the Unicode character number. We might 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 Windows command to change to code page 65001 is sometimes necessary:

```
chcp 65001
```

Linux and Mac OS X 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. The escape sequences start with a `\` character. The next character tells exactly how the Unicode character will be represented. Locate the character that's needed. Get the name or the number. The numbers are always given as hexadecimal, base 16. They're often written as `U+2680`. The name might be `DIE FACE-1`. Use `\unnnn` with up to a four-digit number. 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 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 `\` character in the string, we need to use `\\`. For example, we might need this for Windows filenames.

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\u004f'
```

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

When the string is being compiled by Python, the `\uxx`, `\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.

Back in the *String parsing with regular expressions* recipe, we noted that regular expressions use a lot of `\` characters and we specifically do not want Python's normal compiler to touch them; we used the `r'` prefix on a regular expression string to prevent the `\` from being treated as an escape and possibly converted to something else.

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 this `'\\w+[\u2680\u2681\u2682\u2683\u2684\u2685]\\d+'`. We skipped the `r'` prefix on the string. We doubled up the `\` used for Regular Expressions. We used `\uxxxx` for the Unicode characters that are part of the pattern. Python's internal compiler will replace the `\uxxxx` with Unicode characters and the `\\` with a single `\` internally.

When we look at a string at the `>>>` prompt, Python will display the string in its canonical form. Python prefers to use the `'` as a delimiter even though we can use either `'` or `"` for a string delimiter. 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 to 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 for writing to a file or transmitting?

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 proper strings of Unicode characters.

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

Plus, we also have some encodings that only work for a small subset of Unicode characters. We try to avoid this, but there are some situations where we'll need to use a subset encoding scheme.

Unless we have a really good reason, we almost always use the UTF-8 encoding for Unicode characters. Its main advantage is that it's a compact representation for the Latin alphabet 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 assure that a particular encoding is used everywhere. Set the environment variable as:

```
export PYTHONIOENCODING=UTF-8
```

2. Run Python:

```
python3.5
```

3. We sometimes need to make specific settings when we open a file inside our script. We'll return this in `Chapter 8`, *Input/Output, Physical Format, Logical Layout*. 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 need to use manual encoding:

```
>>> 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 are also UTF-16 and UTF-32. The number is the typical number of bits per character. A file with 1000 characters encoded in UTF-32 would be 4000 8-bit bytes. A file with 1000 characters encoded in UTF-8 could be as few as 1000 bytes, depending on the exact mix of characters. In the UTF-8 encoding, characters with Unicode numbers above `U+007F` require multiple bytes.

Various OS's have their own coding schemes. Mac OS X files are often 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 tricky part is to avoid writing any more bytes than is necessary.

The historical `ASCII` encoding can only represent about 250 of the Unicode characters as bytes. It's easy to create a string which cannot be encoded using the ASCII scheme.

Here's what the error looks like:

```
>>> 'You drew \U0001F000'.encode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
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 a poorly chosen encoding. When we see this, we'll need to change our processing to select a more useful encoding; ideally, UTF-8.

Bytes vs Strings Bytes are often displayed using printable characters. We'll see `b'hello'` as a short-hand for a five-byte value. The letters are chosen using the old ASCII encoding scheme. Many byte values from about `0x20` to `0xFE` will be shown as characters. This 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 complex strings with "template".format()* and the *Building complex strings from lists of characters* 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 the 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`.

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 the 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 8`, *Input/Output, Physical Format, Logical Layout*. We've included the 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 because we own a large sailboat. Or perhaps because good friends of ours have a large sailboat and 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 where we can get the warnings: `http://www.nws.noaa.gov/view/national.php?prod=SMWsid=AKQ`.

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

```
>>> import urllib.request
>>> warnings_uri=
'http://www.nws.noaa.gov/view/national.php?prod=SMW&sid=AKQ'
>>> with urllib.request.urlopen(warnings_uri) as source:
...     warnings_text= source.read()
```

Or, we can use programs like `curl` or `wget` to get this. We might do:

```
curl -O http://www.nws.noaa.gov/view/national.php?prod=SMW&sid=AKQ
mv national.php\?prod\=SMW AKQ.html
```

Since `curl` left us with an awkward file name, we needed to rename the file.

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

```
>>> warnings_text[:80]
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.or'
```

And goes on for a while providing details. Because it 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.

Bytes vs Strings Bytes are often displayed using printable characters. We'll see `b'hello'` as a short-hand for a five-byte value. The letters are chosen using the old ASCII encoding scheme. Many byte values from about `0x20` to `0xFE` will be shown as characters. This can be confusing. The prefix of `b'` is our hint that we're looking at bytes, not proper Unicode characters.

Generally, bytes behave somewhat like strings. Sometimes we can work with bytes directly. Most of the time, we'll want to decode the bytes and create proper Unicode characters.

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 with 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. Other good guesses include ISO-8859-1. In some cases, the guess will depend on the language.

2. *Section 7.2.3, 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 Transitional//EN"
"http://www.w3.or'
```

The `b'` prefix is gone. 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. Parse the resulting document.

Since this is an HTML document, we should use **Beautiful Soup**. 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
>>> title_pattern = re.compile(r"\<h3\>(.*?)\</h3\>")
>>> title_pattern.search( document )
<_sre.SRE_Match object; span=(3438, 3489), match='<h3>There are no products
active at this time.</h>
```

This tells us what we need to know: there are no warnings at this time. That doesn't mean smooth sailing, but it does mean that there aren't any major weather systems that can 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 operating system, 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, or 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`.

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 which 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 the 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 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 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 a `,`.

```
>>> 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 an extra `,` 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, but we can see funny things when we have an extra comma: `>>> 355, (355,)`

The extra comma after `355` makes the value into a singleton tuple.

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, `(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 two ways:

- By index position: 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 class of `Sequence`. 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):
  File "<stdin>", line 1, in <module>
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 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've looked at one other sequence, the list, in the *Building complex strings from lists of characters* recipe
- We'll also look at sequences in `Chapter 4`, *Built-in Data Structures – list, tuple, set, dict*

2
Statements and Syntax

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

- Writing python script and module files
- Writing long lines of code
- Including descriptions and documentation
- Better RST markup in docstrings
- Designing complex if...elif chains
- Designing a while statement which terminates
- Avoiding a potential problem with break statements
- Leveraging the exception matching rules
- Avoiding a potential problem with an except: clause
- Chaining exceptions with the raise from statement
- Managing a context using the with statement

Introduction

Python syntax is designed to be very simple. There are a few rules; we'll look at some of the interesting statements in the language as a way to understand those rules. Just looking at the rules without concrete examples can be confusing.

We'll cover some 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 twenty or so different kinds of imperative statements in the language. We've already looked at two kinds of statements in `Chapter 1`, *Numbers, Strings, and Tuples*: 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. Sometimes, however, we might be assigning two variables at the same time, like this:

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

These recipes will look at some of the more complex statements, including `if`, `while`, `for`, `try`, `with`, and `raise`. We'll touch on a few others as we explore the different recipes.

Writing Python script and module files – syntax basics

We'll need to write Python script files in order to do anything truly useful. We can experiment with the language at the interaction >>> prompt. For real work, however, we'll need to create files. The whole point of writing software is to create repeatable processing for our data.

How can we avoid syntax errors and be sure our code matches what's in common use? We need to look at some common aspects of *style*—how we use whitespace to clarify our programming.

We'll also look at a number of more technical considerations. For example, we need to be sure to save our files in the 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 to use spaces instead of tabs. If we use Unix newlines as much as possible, we'll also find things are slightly simpler.

Most text editing tools will work properly with Unix (newline) line endings as well as Windows or DOS (return-newline) line endings. Any tool that can't work with both kinds of line endings should be avoided.

Getting ready

To edit Python scripts, we'll need a good programming text editor. Python comes with a handy editor, IDLE. It works pretty well. It lets us jump back and forth between a file and an interactive >>> prompt, but it's not a great programming editor.

There are dozens and dozens of good programming editors. It's nearly impossible to suggest just one. So we'll suggest a few.

ActiveState has Komodo IDE, which is very sophisticated. The Komodo Edit version is free, and does some of the same things as the full Komodo IDE. This runs on all common OS's; it's a good first choice because it's consistent no matter where we're writing code.

See `http://komodoide.com/komodo-edit/`.

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

BBEdit is very nice for Mac OS X developers. See `http://www.barebones.com/products/bbedit/`.

For Linux developers, there are several built-in editors, including VIM, gedit, or Kate. These are all good. Since Linux tends to be biased toward developers, the editors available are all suitable for writing Python.

What's important is that we'll often have two windows open while we're working:

- The script or file that we're working on.
- Python's >>> prompt (perhaps from a shell or perhaps from IDLE) where we can try things out to see what works and what doesn't. We may be creating our script in Notepad++, but using IDLE to experiment with data structures and algorithms.

We actually have two recipes here. First, we need to set some defaults for our editor. Then, once the editor is set up properly, we can create a generic template for our script files.

How to do it...

First, we'll look at the general setup that we need to do in our editor of choice. We'll use Komodo examples, but the basic principles apply to all editors. Once we've set the edit preferences, we can create our script file.

1. Open the editor of choice. Look at the preferences page.

2. Find the settings for preferred file encoding. With Komodo Edit Preferences, it's on the **Internationalization** tab. Set this to **UTF-8**.
3. Find the settings for indentation. If there's a way to use spaces instead of tabs, check this option. With Komodo Edit, we actually do this backwards—we uncheck **prefer spaces over tabs**.

The rule is this: we want *spaces*; we do not want *tabs*.

Also, set the spaces per indent to be four. That's typical for Python code. It allows us to have several levels of indentation and still keep the code fairly narrow.

Once we're sure that our files will be saved in UTF-8 encoding, and we're also sure we're using spaces instead of tabs, we can create an example script file:

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

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

   This sets an association between the file you're writing and Python.

   For Windows, the file name to program association is done through a setting in one of the Windows control panels. Within the **Default Programs** control panel, there's a panel to **Set Associations**. This control panel shows that `.py` files are bound to the Python program. This is normally set by the installer, and we rarely need to change it or set it manually.

Windows developers can include the preamble line anyway. It will make Mac OS X and Linux folks happy when they download the project from GitHub.

2. After the preamble, there should be a triple-quoted block of text. This is the documentation string (called a **docstring**) for the file we're going to create. It's not technically mandatory, but it's essential for explaining what a file contains.

```
'''
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')
```

With this, our script does something. In other recipes we'll look at a number of other statements for doing things. It's common to create function and class definitions, as well as write statements to use the functions and classes to do things.

At the top level of our scripts, all of the statements must begin at the left margin and must be complete on a single line. There are some complex statements which will have blocks of statements nested inside them. These internal blocks of statements must be indented. Generally—because we set indentation to four spaces—we can hit the *Tab* key to indent.

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? The entire language is designed to work using just the original 128 ASCII characters.

We often find that ASCII is limiting. It's easier to set our editor to use UTF-8 encoding. With this setting, we can simply use any character that makes sense. We can use characters like μ as Python variables if we save our programs in UTF-8 encoding.

This is legal Python if we save our file in UTF-8:

```
π=355/113
print(π)
```

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

When we set up our editor to use a four-space indent, we can then use the button labeled Tab on our keyboard to insert four spaces. Our code will align properly, and the indentation will show how our statements nest inside each other.

The initial `#!` line is a comment: everything between a `#` and the end of the line is ignored. OS shell programs like **bash** and **ksh** look at the first line of a file to see what the file contains. The first few bytes are sometimes called *magic* because the shell is peeking at them. Shell programs look for the two-character sequence of `#!` to identify the program responsible for this data. We prefer to use `/usr/bin/env` to start the Python program for us. We can leverage this to make Python-specific environment settings via the `env` program.

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. There are tools like Pydoc and Sphinx that can reformat the module docstrings into elegant documentation. We'll look at this in separate recipes.

Additionally, unit test cases can be included in the docstrings. Tools like **doctest** can extract examples from the document string and execute the code to see if the answers in the documentation match the answers found by running the code. Most of this book is validated with doctest.

The triple-quoted documentation strings are preferred over the `#` comments. The text between `#` and the end of the line is ignored, and counts as a comment. Since this is limited to a single line, it is used sparingly. A docstring can be of indefinite size; they are used widely.

In Python 3.5, we'll sometimes see this kind of thing in a script file:

```
color = 355/113 # type: float
```

The `# type: float` comment can be used by a type inferencing system to establish that the various data types can occur when the program is actually executed. For more information on this, see **Python Enhancement Proposal 484**: `https://www.python.org/dev/peps/pep-0484/`.

There's another bit of overhead that's sometimes included in a file. The VIM editor lets us keep edit preferences in the file. This is called a **modeline**. We often have to enable modelines by including the `set modeline` setting in our `~/.vimrc` file.

Once we've enabled modelines, we can include a special `# vim` comment at the end of our file to configure VIM.

Here's a typical modeline that's useful for Python:

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

This sets the Unicode `u+0009` TAB characters to be transformed to eight spaces when we hit the *Tab* key, we'll shift four spaces. This setting is carried in the file; 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 the *Writing better RST markup in docstrings* recipes
- For more information in suggested style, see `https://www.python.org/dev/peps/pep-0008/`

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 line is easier to read; opinions vary, but 65 characters is often cited as ideal. See `http://webtypography.net/2.1.2`.

While shorter lines are easier on the eyes, our code can refuse to cooperate with this principle. Long statements are a common problem. How can we break long Python statements into more manageable pieces?

Getting ready

Often, we'll have a statement that's awkwardly long and hard to work with. 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 = 'the internal representation is
{mantissa:d}/2**53*2**{exponent:d}'.format(mantissa=mantissa_whole,
exponent=exponent)
>>> 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. It looks bad on our screen when trying to edit this script.

We can't simply 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 is 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 onto the next line.
- We can leverage Python's rule that a statement can span multiple logical lines because the `()`, the `[]`, and the `{}` characters must balance. In addition to using `()` and `\`, we can also exploit the way Python automatically concatenates adjacent string literals to make a single, longer literal; `("a" "b")` is the same as `ab`.
- In some cases, we can decompose a statement by assigning intermediate results to separate variables.

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

Using backslash to break a long statement into logical lines

Here's the context for this technique:

```
>>> 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)
```

Python allows us to use \ and break the line.

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

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

2. If there's a *logical* break, insert the \ there. Sometimes, there's no really good break:

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

For this to work, the \ must be the last character on the line. We can't even have a single space after the \. This is fairly hard to see; for this reason, we don't encourage it.

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))
```

2. Add the extra `()` characters that 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
```

3. 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 `()` character's 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 the rare cases when we can't add `()` characters, or adding `()` characters doesn't improve things, 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 combines string literals. This is particularly effective for long, complex format strings:

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

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

We can always break a long string into adjacent pieces. Generally, this is most effective when the pieces are surrounded by `()` characters. We can then use as many physical line breaks as we need. This is limited to those situations where we have particularly long string values.

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))
   ```

   This is generally quite simple. It may require a little care to do the algebra to locate sensible sub-expressions.

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

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

This is an essential textual replacement of the original complex sub-expression with a variable.

We didn't give these variables descriptive names. In some cases, the sub-expressions have some semantics that we can capture with meaningful names. In this case, we didn't understand the expression well enough to provide deeply meaningful names. Instead, we chose short, arbitrary identifiers.

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 techniques called **line joining**. The manual calls the 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. It is 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. We may have the `()` characters as a required syntax. For example, we already have `()` characters as part of the syntax for the `print()` function. We might do this to break up a long statement:

```
>>> print(
...     'several values including',
...     'mantissa =', mantissa,
...     'exponent =', exponent
... )
```

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 an expression. The most notable example of this is the `import` statement—it can become long, but doesn't use any expressions that can be parenthesized.

The language designers, however, allow us to use `()` characters so that a long list of names can be broken up into multiple logical lines:

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

In this case, the `()` characters are emphatically not part of an expression. The `()` characters are just extra syntax, included to make the statement consistent with other statements.

See also

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

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.

Because clarity is important, there are some formatting recipes that can help make the documentation very clear. This recipe also contains a suggested outline so that the documentation will be reasonably complete.

Getting ready

If we've used the *Writing python script and module files - syntax basics* recipe to build a script file, we'll have put a small documentation string in our script file. We'll expand on this documentation string.

There are other places where documentation strings should be used. We'll look that these additional locations in `Chapter 3`, *Function Definitions*, and `Chapter 6`, *Basics of Classes and Objects*.

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. In this case, the docstring summary can focus on what the module is more than what it does. The docstring can provide examples of using the functions and classes that are defined in the module. In `Chapter 3`, *Function Definitions*, and `Chapter 6`, *Basics of Classes and Objects*, we'll look more closely at this idea of a package of functions or classes.
- **Scripts**: These are files that we generally expect will do some real work. In this case, we want to focus on doing rather than being. The docstring should describe what it 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 some careful editing to strike a proper balance between doing and being. In most cases, we'll simply provide both kinds of documentation.

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 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.

The way the information is displayed by tools like sphinx and pydoc suggests a specific style hint. In the output from these tools, the context is pretty clear, therefore it's common to omit a subject in the summary sentence. The sentence often begins with the verb.

For example, a summary like this: *This script downloads and decodes the current Special Marine Warning (SMW) for the area AKQ* has a needless *This script*. We can drop that and begin with the verb phrase *Downloads and decodes....*

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 them 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 summary text.
- **DESCRIPTION**: A more complete explanation of what this script does.
- **OPTIONS**: If `argparse` is used, this is a place to put the details of each argument. Often we'll repeat the `argparse` help parameter.
- **ENVIRONMENT**:If `os.environ` is used, this is the place to describe the environment variables and what they mean.
- **FILES**: 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.
- **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 belongs elsewhere in the project's GitHub or SourceForge pages.

3. Fill in the details under each topic. It's important to be accurate. Since we're embedding this documentation within the same file as the code, it's easy to check elsewhere in the module to be sure that the content is correct and complete.
4. For code samples, there's a cool bit of RST markup we can use. Recall that all elements are separated by blank lines. In one paragraph, use `::` by itself. In the next paragraph, provide the code example indented by four spaces.

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

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

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
    <h3>There are no products active at this time.</h3>
'''
```

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.

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 = 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 modules 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. This should be a summary. Within each class or function, we'll have a separate docstring with the details for that item.
4. For code examples, see the previous examples. Use `::` as a paragraph or the ending of a paragraph. Indent the code example by four spaces.

How it works...

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

These two recipes for describing software are based on summaries of many individual pages of documentation. The goal is to leverage the well-known set of topics. This makes our module documentation mirror the common practice.

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 has a simple syntax rule that 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 extremely nice.

When we want to include a block of code, we'll have some special paragraphs:

- Separate the code from the text by blank lines.
- Indent the code by four spaces.
- Provide a prefix of `::`. We can either do this as its own separate paragraph, or as a special double-colon at the end of the lead-in paragraph:

```
Here's an example::

    more_code()
```

- The `::` is used on the lead-in paragraph.

There are places for novelty and art in software development. Documentation is not really the place to push the envelope. Clever algorithms and sophisticated data structures can be novel and clever.

A unique voice, or quirky presentation isn't fun for users who simply want to use the software. An amusing style isn't helpful when debugging. Documentation should be commonplace and conventional.

It can be challenging to write good software documentation. There's a broad chasm between too little information and documentation which simply recapitulates the code. Somewhere, there's a good balance. 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. Provide this *semi-knowledgeable* user the information they need to describe what the software does and how to use it.

In many cases, we need to address two parts of the use cases:

- The intended use of the software
- How to customize or extend the software

These may be two distinct audiences. There may be users who are distinct from developers. Each has a distinct perspective, and different parts of the documentation need to respect these two perspectives.

See also

- We look at additional techniques in *Writing better RST markup in docstrings*.
- 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`, *Function Definitions*, and classes in `Chapter 6`, *Basics of Classes and Objects*, 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`.

Writing better RST markup in docstrings

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. What RST features can we use to make documentation more readable?

Getting ready

In the *Including descriptions and documentation* recipe, we looked at putting a basic set of documentation into a module. This is the starting point for writing our documentation. There are a large number of RST formatting rules. We'll look at a few which are important for creating readable documentation.

How to do it...

1. Be sure to write an outline of the key points. This may lead to creating RST section titles to organize the material. A section title is a two-line paragraph with the title followed by an underline using =, −, ^, ~, or one of the other Docutils characters for underlining.

   A heading will look like this.

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

   The heading text is on one line, 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 not fewer.

   The RST tools will deduce our pattern of using underlining characters. As long as the underline characters are used consistently, the algorithm for matching underline character to desired heading will detect the pattern. The keys to this are consistency and a clear understanding of section and subsection.

   When starting out, it can help to make an explicit reminder sticky note like this:

Character	Level
=	1
-	2
^	3
~	4

2. Fill in the various paragraphs. Separate paragraphs (including the section titles) by blank lines. Extra blank lines don't hurt. Omitting blank lines will lead the RST parsers to see a single, long paragraph, which may not be what we intended.

   We can use inline markup for emphasis, strong emphasis, code, hyperlinks, and inline math, among other things. If we're planning on using Sphinx, then we have an even larger collection of text roles that we can use. We'll look at these techniques soon.

3. If the programming editor has a spell checker, use that. This can be frustrating because we'll often have code samples that may include 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. The algorithm for deducing this is relatively simple and has these rules:

- If the underline character has been seen before, the level is known
- If the underline character has not been seen before, then it must be indented one level below the previous outline level
- If there is no previous level, this is level one

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

```
====
-----
^^^^^^
^^^^^^
-----
^^^^^^
~~~~~~~~
^^^^^^
```

We can see that the first outline character, =, will be level one. The next, −, is unknown, but appears after a level one, so it must be level two. The third headline has, ^, which is previously unknown, and must be level three. The next ^ is still level three. The next two, − and ^, are level two and three respectively.

When we encounter the new character, ~, it's beneath a level three and must, therefore, be a level four heading.

From this overview, we can see that inconsistency will lead to confusion.

If we change our mind part-way through a document, this algorithm can't detect that. If—for inexplicable reasons—we decide to skip over a level and try to have a level four heading inside a level two section, that simply can't be done.

There are several different kinds of body element that the RST parser can recognize. We've shown a few. The 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 for spaces. They may also be introduced with the `.. parsed-literal::` directive. A doctest block is indented 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 that can be put on 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 specialized text items.

There's more...

For completeness, we'll note here that RST paragraphs are separated by blank lines. There's quite a bit more to RST than this core rule.

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 to use a font to show that we're referring to elements of our code. We'll look at inline markup in the *Using Inline Markup* recipe.

- **Lists**: These are paragraphs that begin with something that looks like a number or a bullet. For bullets, use a simple `-` or `*`. Other characters can be used, but these are common. We might have paragraphs like this.

  It helps to have bullets because:

  - They can help clarify
  - They can help organize

- **Numbered Lists**: There are a variety of patterns that are recognized. We might use something like this.

  Four common kinds of numbered paragraphs:

  - Numbers followed by punctuation like `.` or `)`.
  - A letter followed by punctuation like `.` or `)`.
  - A roman numeral followed by punctuation.
  - A special case of `#` with the same punctuation used on the previous items. This continues the numbering from the previous paragraphs.

- **Literal Blocks**: A code sample must be presented literally. The text for this must be indented. We also need to prefix the code with `::`. The `::` character must either be a separate paragraph or the end of a lead-in to the code example.
- **Directives**: A directive is a paragraph that generally looks like `.. directive::`. It may have some content that's indented so that it's contained within the directive. It might look like this:

```
..  important::

    Do not flip the bozo bit.
```

  The `.. important::` paragraph is the directive. This is followed by a short paragraph of text indented within the directive. In this case, it creates a separate paragraph that includes the admonition of *important*.

Using directives

Docutils has many built-in directives. Sphinx adds a large number of directives with a variety of features.

Some of the most commonly used directives are the admonition directives: *attention*, *caution*, *danger*, *error*, *hint*, *important*, *note*, *tip*, *warning*, and the generic *admonition*. These are compound body elements because they can have multiple paragraphs and nested directives within them.

We might have things like this to provide appropriate emphasis:

```
..  note:: Note Title

    We need to indent the content of an admonition.
    This will set the text off from other material.
```

One of the other common directives is the `parsed-literal` directive.

```
..  parsed-literal::

    any text
        *almost* any format
    the text is preserved
        but **inline** markup can be used.
```

This can be handy for providing examples of code where some portion of the code is highlighted. A literal like this is a simple body element, which can only have text inside. It can't have lists or other nested structures.

Using inline markup

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

- We can surround a word or phrase with `*` for `*emphasis*`.
- We can surround a word or phrase with `**` for `**strong**`.
- We surround references with single back-tick (`). Links are followed by a _. We might use `` `section title`_ `` to refer to a specific section within a document. We don't generally need to put anything around URL's. The Docutils tools recognize these. Sometimes we want a word or phrase to be shown and the URL concealed. We can use this: `` `the Sphinx documentation <http://www.sphinx-doc.org/en/stable/>`_ ``.

- We can surround code-related words with double back-tick (``) to make them look like ``code``.

There's also a more general technique called a text role. A role is a little more complex-looking than simply wrapping a word or phrase in the * characters. We use `: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**`. The rest are only available as explicit roles.

Also, we can define new roles with a simple directive. If we want to do very sophisticated processing, we can provide docutils with class definitions for handling roles, allowing us to tweak the way our document is processed. Sphinx adds a large number of roles to support detailed cross references among functions, methods, exceptions, classes, and modules.

See also

- For more information on RST syntax, see `http://docutils.sourceforge.net`. This includes a description of the docutils tools.
- For information on **Sphinx Python Documentation Generator**, see `http://www.sphinx-doc.org/en/stable/`.
- The Sphinx tool adds many additional directives and text roles to the basic definitions.

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 complex, 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, C . These are the two conditions for an `if...else` statement. One condition, $\neg C$, is stated in the `if` clause, the other is implied in the `else`.

We'll use $p \vee q$ to mean Python's **OR** operator in this explanation. We can call these two conditions *complete* because:

$$C \vee C = \neg \boldsymbol{T}$$

We call this complete because no other conditions can exist. There's no third choice. This is the **Law of the Excluded Middle**. It's also the operating principle behind the `else` clause. The `if` statement body is executed or the `else` statement is executed. There's no third choice.

In practical programming, we often have complex choices. We may have a set of conditions, $C = \{C_1, C_2, C_3, ..., C_n\}$.

We don't want to simply assume that:

$$C_1 \vee C_2 \vee C_3 \vee ... \vee C_n = \boldsymbol{T}$$

We can use $\bigvee_{c \in C} c$ to have a meaning similar to `any(C)`, or perhaps `any([C_1, C_2, C_3, ..., C_n])`. We need to prove that $\bigvee_{c \in C} c = \boldsymbol{T}$; we can't assume this is `true`.

Here's what might go wrong—we might miss some condition, C_{n+1}, that got lost in the tangle of logic. Missing this will mean that our program will fail to work properly for this case.

How can we be sure we haven't missed something?

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 all bets placed on the pass line
- 7 or 11 is a winner for all bets placed on the pass line
- The remaining numbers establish a *point*

Many players place their bets on the pass line. There's also a *don't pass* line, which is less commonly used. 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 alternatives we know. In our example, we have three rules: (2, 3, 12), (7, 11), and the vague remaining numbers.
2. Determine the universe of all possible conditions. For this example, there are 10 conditions: the numbers from 2 to 12.
3. Compare the known alternatives with the universe. There are three possible outcomes of this comparison between the set of conditions, C, and the universe of all possible conditions, U:

The known alternatives have more conditions than the universe; $C \supset U$. This is a huge design problem. This requires rethinking the design from the foundations.

There's a gap between the known conditions and the universe of possible conditions; $U \setminus C \neq \emptyset$. In some cases, it's clear that we haven't covered all of the possible conditions. In other cases, it takes some careful reasoning. We'll need to replace any vague or poorly defined terms with something much more precise.

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

The known alternatives match the universe of possible alternatives; $U \equiv C$. There are two common cases:

- We have something as simple as $C \vee \neg C$. We can use a single `if` and `else` clause—we do not need to use this recipe because we can easily deduce $\neg C$.
- We might have something more complex. Since we know the entire universe, we can show that $\bigvee_{c \in C} c = T$. We need to use this recipe to write a chain of `if` and `elif` statements, one clause per condition.

The distinction is not always crisp. In our example, we don't have a detailed specification for one of the conditions, but the condition is *mostly* clear. If we think the missing condition is obvious, we can use an `else` clause instead of writing it out explicitly. If we think the missing condition might be misunderstood, we should treat it as vague and use this recipe.

1. Write the `if...elif...elif` chain that covers all of the known conditions. For our example, it will look 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(die)
```

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

```
else:
    raise Exception('Design Problem Here: not all conditions
accounted for')
```

This extra `else` crash condition gives us a way to positively identify when a logic problem is found. We can be sure that any error we make will lead to a conspicuous problem.

How it works...

Our goal is to be sure that our program always works. While testing helps, we can still have wrong assumptions in both design and test cases.

While rigorous logic is essential, we can still make errors. Further, someone else could try to tweak our code and introduce an error. More embarrassingly, we could make a change in our own code that leads to breakage.

The `else` crash option 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 when an exception gets raised.

The `else` crash option doesn't have a significant performance impact. A simple `else` clause is slightly faster than an `elif` clause with a condition. If we think that our application performance depends in any way on the cost of a single expression, we've got more serious design problems to solve. The cost of evaluating a single expression is rarely the costliest part of an algorithm.

Crashing with an exception is a sensible behavior in the presence of a design problem. It doesn't make much sense to follow the design pattern of writing a warning message to a log. If we have this kind of logic gap, the program is fatally broken and it's important to find and fix this as soon as it's known.

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 simple like *m* the larger of *a* or *b*.

(For the sake of working through the logic, we'll avoid `m = max(a, b)`.)

We can formalize the final condition like this:

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

We can work backwards 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` and `m = b` will be appropriate, but only under certain conditions.

Each of these statements is part of the solution, and we can derive a precondition that shows when the statement should be used. The preconditions for each assignment statement are the `if` and `elif` expressions. We need to use `m = a` when `a >= b`; we need to use `m=b` when `b >= 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 we use. 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 `if...elif...elif` chains. We should always write the conditions without regard to order of evaluation of the clauses.

See also

- This is similar to the syntactic problem of a **dangling else**. See `https://en.wikipedia.org/wiki/Dangling_else`.
- Python's indentation removes the dangling else syntax problem. It doesn't remove the semantic issue of trying to be sure that all conditions are properly accounted for in a complex `if...elif...elif` chain.
- Also, see `https://en.wikipedia.org/wiki/Predicate_transformer_semantics`.

Designing a while statement which terminates properly

Much of the time, the Python `for` statement provides all of the iteration controls we need. In many cases, we can use built-in functions like `map()`, `filter()`, and `reduce()` to process collections of data.

There are a few situations, however, where we need to use a `while` statement. Some of those situations involve data structures where we can't create a proper iterator to step through the items. Other items involve interactions with human users, where we don't have the data until we get input from the person.

Getting ready

Let's say that we're going to be prompting a user for their password. We'll use the `getpass` module so that there's no echo.

Further, to be sure they've entered it properly, we'll want to prompt them twice and compare the results. This is a situation where a simple `for` statement isn't going to work out well. It can be pressed into service, but the resulting code looks strange: `for` statements have an explicit upper bound; prompting a user for input doesn't really have an upper bound.

How to do it...

We'll look at a six-step process that outlines the core of designing this kind of iterative algorithm. This is the kind of thing we need to do when a simple `for` statement doesn't solve our problem.

1. Define done. In our case, we'll have two copies of the password, `password_text` and `confirming_password_text`. The condition which must be `true` after the loop is that `password_text == confirming_password_text`. Ideally, reading from people (or files) is a bounded activity. Eventually, people will enter the matching pair of values. Until they enter the matching pair, we'll iterate indefinitely.

   There are other boundary conditions. For example, end of file. Or we allow the person to go back to a previous prompt. Generally, we handle these other conditions with exceptions in Python.

   Of course, we can always add these additional conditions to our definition of done. We may need to have a complex terminating condition like end of file OR `password_text == confirming_password_text`.

   In this example, we'll opt for exception handling and assume that a `try:` block will be used. It greatly simplifies the design to have only a single clause in the terminating condition.

   We can rough out the loop like this:

   ```
   # initialize something
   while # not terminated:
       # do something
   assert password_text == confirming_password_text
   ```

   We've written our definition of done as a final `assert` statement. We've included comments for the rest of the iteration that we'll fill in in subsequent steps.

2. Define a condition that's `true` while the loop is iterating. This is called an **invariant** because it's always `true` at the start and end of loop processing. It's often created by generalizing the post-condition or introducing another variable.

   When reading from people (or files) we have an implied state change that is an important part of the invariant. We can call this the *get the next input* change in state. We often have to articulate clearly that our loop will be acquiring some next value from an input stream.

   We have to be sure that our loop properly gets the next item in spite of any complex logic in the body of the `while` statement. It's a common bug to have a condition where a next input is not actually fetched. This leads to programs which *hang*—there's no state change in one logic path through the `if` statements in the body of the `while` statement. The invariant wasn't reset properly, or it wasn't articulated properly when designing the loop.

   In our case, the invariant will use a conceptual `new-input()` condition. This condition is `true` when we've read a new value using the `getpass()` function. Here's our expanded loop design:

```
# initialize something
# assert the invariant new-input(password_text)
# and new-input(confirming_password_text)
while # not terminated:
    # do something
    # assert the invariant new-input(password_text)
    # and new-input(confirming_password_text)
assert password_text == confirming_password_text
```

3. Define the condition for leaving the loop. We need to be sure that this condition depends on the invariant being `true`. We also need to be sure that, when this termination condition is finally `false`, the target state will become `true`.

   In most cases, the loop condition is the logical negation of the target state. Here's the expanded design:

```
# initialize something
# assert the invariant new-input(password_text)
# and new-input(confirming_password_text)
while password_text != confirming_password_text:
    # do something
    # assert the invariant new-input(password_text)
    # and new-input(confirming_password_text)
assert password_text == confirming_password_text
```

4. Define the initialization that will make sure that both the invariant will be `true` and that we can actually test the terminating condition. In this case, we need to get values for the two variables. The loop now looks like this:

```
password_text= getpass()
confirming_password_text= getpass("Confirm: ")
# assert new-input(password_text)
# and new-input(confirming_password_text)
while password_text != confirming_password_text:
    # do something
    # assert new-input(password_text)
    # and new-input(confirming_password_text)
assert password_text == confirming_password_text
```

5. Write the body of the loop which will reset the invariant to `true`. We need to write the fewest statements that will do this. For this example loop, the fewest statements are pretty obvious—they match the initialization. Our updated loop looks like this:

```
password_text= getpass()
confirming_password_text= getpass("Confirm: ")
# assert new-input(password_text)
# and new-input(confirming_password_text)
while password_text != confirming_password_text:
    password_text= getpass()
    confirming_password_text= getpass("Confirm: ")
    # assert new-input(password_text)
    # and new-input(confirming_password_text)
assert password_text == confirming_password_text
```

6. Identify a clock—a monotonically decreasing function that shows that each iteration of the loop really does make progress toward the terminating condition.

   When gathering input from people, we're forced to make an assumption that—eventually—they'll enter a matching pair. Each trip through the loop brings us one step closer to that matching pair. To be properly formal, we can assume that there will be n inputs before they match; we have to show that each trip through the loop decreases the number which remain.

   In complex situations, we may need to treat the user's input as a list of values. For our example, we'd think of the user input as a sequence of pairs: $[(p_1, q_1),(p_2, q_2),(p_3, q_3),\ldots,(p_n, q_n)]$. With a finite list, we can more easily reason about whether or not our loop really is making progress towards completion.

Because we built the loop based on the target `final` condition, we can be absolutely sure that it does what we want it to do. If our logic is sound, the loop will terminate, and will terminate with the expected results. This is the goal of all programming—to have the machine reach a desired state given some initial state.

Removing some comments, we have this as our final loop:

```
password_text= getpass()
confirming_password_text= getpass("Confirm: ")
while password_text != confirming_password_text:
    password_text= getpass()
    confirming_password_text= getpass("Confirm: ")
assert password_text == confirming_password_text
```

We left the final post-condition in place as an `assert` statement. For complex loops it's both a built-in test, as well as a comment that explains how the loop works.

This design process often produces a loop that looks similar to one we might develop based on intuition. There's nothing wrong with having a step by step justification for an intuitive design. Once we've done this a few times, we can be much more confident in using a loop knowing that we can justify the design.

In this case, the loop body and the initialization happen to be the same code. If this is a problem, we can define a tiny two-line function to avoid repeating the code. We'll look at this in `Chapter 3`, *Function Definitions*.

How it works...

We start out by articulating the goal for the loop. Everything else that we do will assure that the code written leads to that goal condition. Indeed, this is the motivation behind all software design—we're always trying to write the fewest statements that lead to a given goal state. We're often working *backwards* from goal to initialization. Each step in the chain of reasoning is essentially stating the weakest precondition for some statement, `S`, that leads to our desired outcome condition.

Given a post-condition, we're trying to solve for a statement and a precondition. We're always building this pattern:

```
assert pre-condition
S
assert post-condition
```

The post-condition is our definition of done. We need to hypothesize a statement, `S`, that leads to done, and a precondition for that statement. There are always an infinite number of alternative statements; we focus on the weakest precondition—the one that has the fewest assumptions.

At some point—usually when writing the initialization statements—we find that the pre-condition is merely `true`: any initial state will do as the precondition for a statement. That's how we know that our program can start from any initial state and complete as expected. This is ideal.

When designing a `while` statement, we have a nested context inside the statement's body. The body should always be in a process of resetting the invariant condition to be `true` again. In our example, this means reading more input from the user. In other examples, we might be processing another character in a string, or another number from a set of numbers.

We need to prove that when the invariant is `true` and the loop condition is `false` then our final goal is achieved. This proof is easier when we start from the final goal and create the invariant and the loop condition based on that final goal.

What's important is patiently doing each step so that our reasoning is solid. We need to be able to prove that the loop will work. Then we can run unit tests with confidence.

See also

- We look at some other aspects of advanced loop design in the *Avoiding a potential problem with break statements* recipe.
- We also looked at this concept in the *Designing complex if...elif chains* recipe.
- 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`.
- Algorithm design is a big subject. A good introduction is by Skiena, *Algorithm Design Manual*. See `http://www3.cs.stonybrook.edu/~algorith/`.

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, some processing has been done.

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

There's a side issue here. What if the loop ends without executing the `break`? We are forced to assert that there does not exist even one item that triggered the `break`. **DeMorgan's Law** tells us that a not exists condition can be restated as a *for all* condition: $\neg\exists_x B(x) \equiv \forall_x \neg B(x)$. In this formula, *B(x)* is the condition on the `if` statement that includes the `break`. If we never found *B(x)*, then for all items $\neg B(x)$ was `true`. This shows some of the symmetry between a typical *for all* loop and a *there exists* loop which includes a `break`.

The condition that's `true` upon leaving a `for` or `while` statement can be ambiguous. Did it end normally? Did it execute the `break`? We can't *easily* tell, so we'll provide a recipe that gives us some design guidance.

This can become an even bigger problem when we have multiple `break` statements, each with its own condition. How can we minimize the problems created by having complex `break` conditions?

Getting ready

Let's find the first occurrence of a `:` or `=` in a string. This is a good example of a *there exists* modification to a `for` statement. We don't want to process all characters, we want to know where there exists the left-most `:` or `=`.

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

What about this edge case?

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

That's awkwardly wrong. What happened?

How to do it...

As we noted in the *Designing a while statement which terminates properly* recipe, every statement establishes a post-condition. When designing a loop, we need to articulate that condition. In this case, we didn't properly articulate the post-condition.

Ideally, the post-condition would be something simple like `text[position] in '=:'`. However, if there's no = or `:` in the given text, the simple post-condition doesn't make logical sense. When no character exists which matches the criteria, we can't make any assertion about the position of a character that's not there.

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.

   ```
   text[position] in '=:'
   ```

2. Add post-conditions for the edge cases. In this example, we have two additional conditions:
   - There's no = or `:`.
   - There are no characters at all. The `len()` is zero, and the loop never actually does anything. In this case, the position variable will never be created.

     ```
     (len(text) == 0
     or not('=' in text or ':' in text)
     or text[position] in '=:')
     ```

3. If a `while` statement is being used, consider redesigning it to have completion conditions. This can eliminate the need for a `break` statement.
4. If a `for` statement is being used, be sure a proper initialization is done, and add the various terminating conditions to the statements after the loop. It can look redundant to have `x = 0` followed by `for x = ...`. It's necessary in the case of a loop which doesn't execute the `break` statement, though.

```
>>> position = -1 # If it's zero length
>>> for position in range(len(sample_2)):
...     if sample_2[position] in '=:':
...         break
...
>>> if position == -1:
...     print("name=", None, "value=", None)
... elif not(text[position] == ':' or text[position] == '='):
...     print("name=", sample_2, "value=", None)
... else:
...     print('name=', sample_2[:position],
...      'value=', sample_2[position+1:])
name= name_only value= None
```

In the statements after the `for`, we've enumerated all of the terminating conditions explicitly. The final output, `name= name_only value= None`, confirms that we've correctly processed the sample text.

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 for the loop terminating.

In more complex loops—with multiple `break` statements—the post-condition can be difficult to work out fully. The post-condition for a loop must include all of the reasons for leaving the loop—the *normal* reasons plus all of the `break` conditions.

In many cases, we can refactor the loop to push the processing into the body of the loop. Rather than simply assert that `position` is the index of the `=` or `:` character, we include the next processing steps of assigning the `name` and `value` values. We might have something like this:

```
if len(sample_2) > 0:
    name, value = sample_2, None
else:
    name, value = None, None
```

```
for position in range(len(sample_2)):
    if sample_2[position] in '=:':
        name, value = sample_2[:position], sample2[position:]
print('name=', name, 'value=', value)
```

This version pushes some of the processing forward, based on the complete set of post-conditions evaluated previously. This kind of refactoring is common.

The idea is to forego any assumptions or intuition. With a little bit of discipline, we can be sure of the post-conditions from any statement.

Indeed, the more we think about post-conditions, the more precise our software can be. It's imperative to be explicit about the goal for our software and work backwards from the goal by choosing the simplest statements that will make the goal become `true`.

There's more...

We can also use an `else` clause on a `for` statement to determine if the loop 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
```

The `else` condition is sometimes confusing, and we don't recommend it. It's not clear that it 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 the 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 let it propagate; generally this will stop the program.
- **Recover from it**: We can write an `except` clause to do some recovery action to undo the effects of something that was only partially completed in the `try` clause. We can take this a step further and wrap the `try` statement in a `while` statement and keep retrying until it succeeds.
- **Silence it**: If we do nothing (that is, `pass`) then processing is resumed after the `try` statement. This silences the exception.
- **Rewrite it**: We can raise a different exception. The original exception becomes a context for the newly-raised exception.
- **Chain it**: We chain a different exception to the original exception. We'll look at this in the *Chaining exceptions with the raise from statement* recipe.

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 are the same. The overall behavior of the software depends on the nested definitions.

Our design of a `try` statement depends on the way that Python exceptions form a class hierarchy. For details, see *Section 5.4, Python Standard Library*. For example, `ZeroDivisionError` is also an `ArithmeticError` and an `Exception`. For another example, a `FileNotFoundError` 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 simple use of the `shutil` 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 rare event of a `FileExistsError`, 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
import os
source_path = Path(os.path.expanduser(
   '~/Documents/Writing/Python Cookbook/source'))
target_path = Path(os.path.expanduser(
   '~/Dropbox/B05442/demo/'))
for source_file_path in source_path.glob('*/*.rst'):
    source_file_detail = source_file_path.relative_to(source_path)
    target_file_path = target_path / source_file_detail
    shutil.copy( str(source_file_path), str(target_file_path
```

We have two paths, `source_path` and `target_path`. We've located all of the directories under the `source_path` that have `*.rst` files.

The expression `source_file_path.relative_to(source_path)` gives us the tail end of the file name, the portion after the base directory. We use this to build a new path under the `target` directory.

While we can use `pathlib.Path` objects for a lot of ordinary path processing, in Python 3.5 modules like `shutil` expect string filenames instead of `Path` objects; we need to explicitly convert the `Path` objects. We can only hope that Python 3.6 changes this.

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:

```
    FileNotFoundError: [Errno 2]
        No such file or directory:
'/Users/slott/Dropbox/B05442/demo/ch_01_numbers_strings_and_tuples/index.rs
t'
```

How do we create a `try` statement that handles the exceptions in the proper order?

How to do it...

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

```
try:
    shutil.copy( str(source_file_path), str(target_file_path) )
```

2. Include the most specific exception classes first. In this case, we have separate responses for the specific `FileNotFoundError` and the more general `OSError`.

```
try:
    shutil.copy( str(source_file_path), str(target_file_path) )
except FileNotFoundError:
    os.makedir( target_file_path.parent )
    shutil.copy( str(source_file_path), str(target_file_path) )
```

3. Include any more general exceptions later:

```
try:
    shutil.copy( str(source_file_path), str(target_file_path) )
except FileNotFoundError:
    os.makedirs( str(target_file_path.parent) )
    shutil.copy( str(source_file_path), str(target_file_path) )
except OSError as ex:
    print(ex)
```

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

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

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

How it works...

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

- Process the `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 the `Exception` will match almost every kind of exception. We don't want this first, because no other clauses will be checked. We must always put generic exceptions last.

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, which 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:

```
except FileNotFoundError:
    os.makedirs( str(target_file_path.parent) )
    shutil.copy( str(source_file_path), str(target_file_path) )
```

If the `os.makedirs()` or `shutil.copy()` functions raise another exception, it won't be handled by this `try` statement. Any exceptions raised here will crash the program as a whole. We have two ways to handle this, both of which involve nested `try` statements.

We can rewrite this to include a nested `try` during recovery:

```
try:
    shutil.copy( str(source_file_path), str(target_file_path) )
except FileNotFoundError:
    try:
        os.makedirs( str(target_file_path.parent) )
        shutil.copy( str(source_file_path), str(target_file_path) )
    except OSError as ex:
        print(ex)
```

```
except OSError as ex:
    print(ex)
```

In this example, we've repeated the `OSError` processing in two places. In our nested context, we'll log the exception and let it propagate, which will likely stop the program. In the outer context, we'll do the same thing.

We say *likely stop the program* because this code could be used inside a `try` statement, which might handle these exceptions. If there's no other `try` context, then these unhandled exceptions will stop the program.

We can also rewrite our overall statement to have nested `try` statements that separate the two exception handling strategies into more local and more global considerations. It would look like this:

```
try:
    try:
        shutil.copy( str(source_file_path), str(target_file_path) )
    except FileNotFoundError:
        os.makedirs( str(target_file_path.parent) )
        shutil.copy( str(source_file_path), str(target_file_path) )
except OSError as ex:
    print(ex)
```

The copy with `makedirs` processing in the inner `try` statement handles only the `FileNotFoundError` exception. Any other exception will propagate out to the outer `try` statement. In this example, we've nested the exception handling so that the generic processing wraps the specific processing.

See also

- In the *Avoiding a potential problem with an except: clause* recipe we look at some additional considerations when designing exceptions
- In the *Chaining exceptions with the raise from statement* recipe we look at how we can chain exceptions so that a single class of exception wraps different detailed exceptions

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. There are a few other mistakes which 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

In the *Avoiding a potential problem with an except: clause* recipe we looked at some considerations when designing exception handling. In that recipe, we discouraged the use of `BaseException` because we 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...

Use `except Exception:` as the most general kind of exception managing.

Handling too many exceptions can interfere with our ability to stop a misbehaving Python program. When we hit *Ctrl* + *C*, or send a `SIGINT` signal via `kill -2`, we generally want the program to stop. We rarely want the program to write a message and keep running, or stop responding altogether.

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

- SystemError
- RuntimeError
- MemoryError

Generally, these exceptions mean that 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.

How it works...

There are two techniques we should avoid:

- Don't capture the `BaseException` class
- Don't use `except:` with no exception class. This matches all exceptions; this will include exceptions we should avoid trying to handle.

Using `except BaseException` or except without a specific class can cause a program to become unresponsive at exactly the time we need to stop it.

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

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

If we silence, wrap, or rewrite any of these, we may have created a problem where none existed. We may have exacerbated a simple problem into a larger and more mysterious problem.

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

See also

- In the *Leveraging the exception matching rules* recipe we look at some considerations when designing exceptions
- In the *Chaining exceptions with the raise from statement* recipe we look at how we can chain exceptions so that a single class of exception wraps different detailed exceptions.

Chaining exceptions with the raise from statement

In some cases, we may want to merge some seemingly unrelated exceptions into a single generic exception. It's common for a complex module to define a single generic `Error` exception which applies to many situations that can arise within the module.

Most of the time, the generic exception is all that's required. If the module's `Error` is raised, something didn't work.

Less frequently, we want the details for debugging or monitoring purposes. We might want to write them to a log, or include the details in an e-mail. In this case, we need to provide supporting details that amplify or extend the generic exception. We can do this by chaining from the generic exception to the root cause exception.

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 Error(Exception):
    pass
```

   That's sufficient to define a new class of exception.

2. When handling exceptions, we can chain them using the `raise from` statement like this:

```
try:
    something
except (IndexError, NameError) as exception:
    print("Expected", exception)
    raise Error("something went wrong") from exception
except Exception as exception:
    print("Unexpected", exception)
    raise
```

In the first `except` clause, we matched two kinds of exception classes. No matter which kind we get, we'll raise a new exception from the module's generic `Error` exception class. The new exception will be chained to the root cause exception.

In the second `except` clause, we matched the generic `Exception` class. We wrote a log message and re-raised the exception. Here, we're not chaining, but simply continuing exception handling in another context.

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 Exception from Exception` statement.

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

```
>>> class Error(Exception):
...     pass
>>> try:
...     'hello world'[99]
... except (IndexError, NameError) as exception:
...     raise Error("index problem") from exception
...
Traceback (most recent call last):
  File "<doctest default[0]>", line 2, in <module>
    'hello world'[99]
IndexError: string index out of range
```

The exception that we just saw was the direct cause of the following exception:

```
Traceback (most recent call last):
  File
"/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/doctest.py
", line 1318, in __run
    compileflags, 1), test.globs)
  File "<doctest default[0]>", line 4, in <module>
    raise Error("index problem") from exception
Error: index problem
```

This shows a chained exception. The first exception in the `Traceback` message is an `IndexError` exception. This is the direct cause. The second exception in the `Traceback` is our generic `Error` exception. This is a generic summary exception, which was chained to the original cause.

An application will see the `Error` exception in a `try:` statement. We might have something like this:

```
try:
    some_function()
except Error as exception:
    print(exception)
    print(exception .__cause__)
```

Here we've shown a function named `some_function()` that can raise the generic `Error` exception. If this function does raise the exception, the `except` clause will match the generic `Error` exception. We can print the exception's message, `exception`, as well as the root cause exception, `exception.__cause__`. In many applications, the `exception.__cause__` value may get written to a debugging log rather than be displayed to users.

There's more...

If an exception is raised inside an exception handler, this also creates a kind of chained exception relationship. This is the *context* relationship rather than the *cause* relationship.

The context message looks similar. The message is slightly different. It says `During handling of the above exception, another exception occurred:`. The first `Traceback` will show the original exception. The second message is the exception raised without using an explicit from connection.

Generally, the context is something unplanned that indicates an error in the `except` processing block. For example, we might have this:

```
try:
    something
except ValueError as exception:
    print("Some message", exceotuib)
```

This will raise a `NameError` exception with a context of a `ValueError` exception. The `NameError` exception stems from misspelling the exception variable as `exceotuib`.

See also

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

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 memory **leaks** 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:

```
>>> import pathlib
```

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]]
```

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 file, or creating the network connection with `urllib.request.urlopen()`. Other common contexts include archives like `zip` files and `tar` files:

```
target_path = pathlib.Path('code/test.csv')
with target_path.open('w', newline='') as target_file:
```

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

```
target_path = pathlib.Path('code/test.csv')
with target_path.open('w', newline='') as target_file:
    writer = csv.writer(target_file)
    writer.writerow(['column', 'data', 'headings'])
    for data in some_source:
        writer.writerow(data)
```

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 = pathlib.Path('code/test.csv')
with target_path.open('w', newline='') as target_file:

    writer = csv.writer(target_file)
    writer.writerow(['column', 'headings'])
    for data in some_source:
        writer.writerow(data)

print('finished writing', target_path)
```

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.

Even if an exception is raised inside the `with` statement, 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 two kinds of exits from the block of code:

- Normal exit with no exception
- An exception was raised

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:
    target_path = pathlib.Path('code/test.csv')
    with target_path.open('w', newline='') as target_file:
        writer = csv.writer(target_file)
        writer.writerow(['column', 'headings'])
        for data in some_source:
            writer.writerow(data)
            raise Exception("Just Testing")
except Exception as exc:
    print(target_file.closed)
    print(exc)
print('finished writing', target_path)
```

In this example, we've wrapped the real work in a `try` statement. This allows us to raise an exception after writing the first to the CSV file. When the exception is raised, we can print the exception. At this point, the file will also be closed. The output is simply this:

```
True
Just Testing
finished writing code/test.csv
```

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. The output `test.csv` file will only have the first row of data from the `some_source` variable.

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 connect created by `urllib.request.urlopen()`.

For all file operations, and all network connections, we should 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` context 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 `https://www.python.org/dev/peps/pep-0343/` for the origins of the with statement

3
Function Definitions

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

- Designing functions with optional parameters
- Using super flexible keyword parameters
- Forcing keyword-only arguments with the * separator
- Writing explicit types on function parameters
- 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 reusable scripts with the script library switch

Introduction

Function definitions are a way to decompose a large problem into smaller problems. Mathematicians have been doing this for centuries. It's also 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 Python 3.5 typing module and how we can create more formal annotations for our functions. We can start down the road toward using the `mypy` project for making more formal assertions about the data types in use.

Designing functions with optional parameters

When we define a function, we often have a need for optional parameters. This allows us to write functions which are more flexible, and can be used in more situations.

We can also think of this as a way to create a family of closely-related functions, each with a slightly different collection of parameters – called the **signature** – but all sharing the same simple name. The idea of many functions sharing the same name can be a bit confusing. Therefore, we'll focus more on the idea of optional parameters.

An example of optional parameters is the `int()` function. This has two forms:

- `int(str)`: For example, the value of `int('355')` has a value of `355`. In this case, we didn't provide a value for the optional `base` parameter; the default value of `10` was used.
- `int(str, base)`: For example, the value of `int('0x163', 16)` is `355`. In this case, we provided a value for the `base` parameter.

Getting ready

A great many games rely on collections of dice. The casino game of *Craps* uses two dice. A game like *Zilch* (or *Greed* or *Ten Thousand*) uses six dice. Variations on the game may use more.

It's handy to have a dice-rolling function that can handle all of these variations. How can we write a dice-simulator that works for any number of dice, but will use two as a handy default value?

How to do it...

We have two approaches to designing a function with optional parameters:

- **General to Particular**: We start by designing the most general solution and provide handy defaults for the most common case.
- **Particular to General**: We 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.

Particular to General Design

When following the Particular to General strategy, we'll design several individual functions and look for common features:

1. Write one version of the function. We'll start with the *Craps* game because it seems simplest:

```
>>> import random
>>> def die():
...     return random.randint(1,6)
>>> def craps():
...     return (die(), die())
```

   We defined a handy helper function, `die()`, which encapsulates a basic fact about what are sometimes called standard dice. There are five platonic solids that can be pressed into service, yielding four-sided, six-sided, eight-sided, twelve-sided, and twenty-sided dice. The six-sided die has a long history, starting as *Astragali* bones, which were easily trimmed into a six-sided cube.

   Here's an example of the underlying `die()` function:

```
>>> random.seed(113)
>>> die(), die()
(1, 6)
```

   We've rolled two dice to show how the values combine for rolling larger piles of dice.

   Our function for the game of *Craps* looks like this:

```
>>> craps()
(6, 3)
>>> craps()
(1, 4)
```

   This shows some two-dice rolls for the game of *Craps*.

2. Write another version of the function:

```
>>> def zonk():
...     return tuple(die() for x in range(6))
```

We've used a generator expression to create a tuple object with six dice. We'll look at generator expressions in depth in online `Chapter 12`, *Functional And Reactive Programming Features* (link provided in `Preface`).

Our generator expression has a variable, `x`, which 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; this name can be used as a hint that we don't ever want to see the value of this variable.

Here's an example of using the `zonk()` function:

```
>>> zonk()
(5, 3, 2, 4, 1, 1)
```

This shows us a roll of six individual dice. There's a short straight (1-5) as well as a pair of ones. In some versions of the game, this is a good scoring hand.

3. Locate the common features in the two functions. This may require some rewriting of the various functions to locate a common design. In many cases, we'll wind up introducing additional variables to replace constants or other assumptions. In this case, we can generalize the creation of the two-tuple. Rather than hardwiring 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():
...     return tuple(die() for x in range(2))
```

This seems like more code than required for solving the specific two-dice problem. In the long run, using a single general function means that we can eliminate a number of specific functions.

4. Merge the two functions. This will often involve exposing a variable that had previously been a constant or other hardwired assumption:

```
>>> def dice(n):
...     return tuple(die() for x in range(n))
```

This provides a general function that covers the needs of both *Craps* and *Zonk*:

```
>>> dice(2)
(3, 2)
>>> dice(6)
(5, 3, 4, 3, 3, 4)
```

5. 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(n=2):
...     return tuple(die() for x in range(n))
```

Now we can simply use `dice()` for *Craps*. We'll need to use `dice(6)` for *Zonk*.

General to Particular design

When following the General to Particular strategy, we'll identify all of the needs first. We'll often do this by introducing variables into the requirements:

1. Summarize the requirements for dice-rolling. We might have a list like this:
    - *Craps*: two dice.
    - First roll in *Zonk*: six dice.
    - Subsequent rolls in *Zonk*: one to six dice.

   This list of requirements shows a common theme of rolling n 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, and show the values for this new parameter that we've introduced:
    - *Craps*: n dice, where $n = 2$.
    - First roll in *Zonk*: n dice, where $n = 6$.
    - Subsequent rolls in *Zonk*: n dice, where $1 \le n \le 6$.

   The goal here is to be absolutely sure that all of the variations really have a common abstraction. In more complex problems, something that seems similar may not have a common specification.

   We want to be sure, also, that we've properly parameterized each of the various functions. In more complex cases, we may have values that don't really need to be parameterized; they can remain as constants.

3. Write the function that fits the general pattern:

```
>>> def dice(n):
...     return (die() for x in range(n))
```

In the third case – subsequent rolls in *Zonk* – we identified a constraint of $1 \le n \le 6$. We need to determine if this is a constraint that's part of our `dice()` function, or if this constraint is imposed on the dice by the simulation application that uses the `dice` function.

In this case, the constraint is incomplete. The rules for *Zonk* require that the dice which are not being rolled form some kind of scoring pattern. The constraint isn't merely that the number of dice is between one and six; the constraint is tied to the game state. There doesn't seem to be a good reason to tie the `dice()` function to the game state.

4. Provide a default value for the most common use case. If our most common simulation was *Craps*, we might do this:

```
>>> def dice(n=2):
...     return tuple(die() for x in range(n))
```

Now we can simply use `dice()` for *Craps*. We'll need to use `dice(6)` for *Zonk*.

How it works...

Python's rules for providing parameter values are very flexible. There are several ways to assure that each parameter has a value. We can think of it working like this:

1. Set each parameter to any provided default value.
2. For arguments without names, the argument values are assigned to the parameters by position.
3. For arguments with names – for example, `dice(n=2)` – the parameter values are assigned using the name. It's an error to assign a parameter both by position and by name.
4. If any parameter doesn't have a value, this is an error.

These rules allow us to provide defaults as needed. They also allow us to mix positional values with named values. The presence of a default value is what makes a parameter optional.

The use of optional parameters stems from two considerations:

- Can we parameterize the processing?
- What's the most common argument value for that parameter?

Introducing parameters into a process definition can be challenging. In some cases, it helps to have code so that we can replace literal values (such as 2 or 6) with a parameter.

In some cases, however, the literal value doesn't need to be replaced with a parameter. It can be left as a literal value. We don't always want to replace every literal with a parameter. Our `die()` function, for example, has a literal value of 6 because we're only interested in standard, cubic dice. This isn't a parameter because we don't see a need to make a more general kind of die.

There's more...

If we want to be very thorough, we can write functions that are specialized versions of our more generalized function. These functions can simplify an application:

```
>>> def craps():
...     return dice(2)
>>> def zonk():
...     return dice(6)
```

Our application features – `craps()` and `zonk()` – depend on a general function, `dice()`. This, in turn, depends on another function, `die()`. We'll revisit this idea in the *Picking an order for parameters based on partial functions* recipe.

Each layer in this stack of dependencies introduces a handy abstraction that saves us from having to understand too many details. This idea of layered abstractions is sometimes called **chunking**. It's a way of managing complexity by isolating the details.

A common extension to this design pattern is to provide parameters at multiple levels in this hierarchy of functions. If we want to parameterize the `die()` function, we'll be providing parameters to both `dice()` and `die()`.

For this more complex parameterization, we'll need to introduce more parameters with default values into our hierarchy. We'll start by adding a parameter to `die()`. This parameter must have a default value so that we don't break any of our existing test cases:

```
>>> def die(sides=6):
...     return random.randint(1,6)
```

After introducing this parameter at the bottom of the stack of abstractions, we'll need to provide this parameter to higher-level functions:

```
>>> def dice(n=2, sides=6):
... return tuple(die(sides) for x in range(n))
```

We now have many ways of using the `dice()` function:

- All default values: `dice()` covers *Craps* nicely.
- All positional arguments: `dice(6, 6)` would cover *Zonk*.
- A mixture of positional and named arguments: The positional values must be provided first because the order matters. For example, `dice(2, sides=8)` would cover a game that uses two eight-sided dice.
- All named arguments: `dice(sides=4, n=4)` this would handle the case where we needed to emulate rolling four tetrahedral dice. When using all named arguments, order doesn't matter.

In this example, our stack of functions only has two layers. In a more complex application, we may have to introduce parameters at many layers in a hierarchy.

See also

- We'll extend some of these ideas in the *Picking an order for parameters based on partial functions* recipe.
- We've made use of optional parameters that involve immutable objects. In this recipe, we focused on numbers. In `Chapter 4`, *Built-in Data Structures – list, set, dict*, 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 that are important for designing functions that have optional values that are mutable objects.

Using super flexible keyword parameters

Some design problems involve solving a simple equation for one unknown given enough known values. For example, rate, time, and distance have a simple linear relationship. We can solve for any one given the other two. Here are the three rules that we can use as an example:

- $d = r \times t$

- $r = d / t$
- $t = d / r$

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 to provide a simple, high-performance software implementation that can perform any of the three different calculations based on what's known and what's unknown. We don't want to use a general algebraic framework; we want to bundle the three solutions into a simple, efficient function.

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.

There's a trick here. We don't necessarily want a single value answer. We can slightly generalize this by creating a small Python dictionary with the three values in it. We'll look more at dictionaries in `Chapter 4`, *Built-in Data Structures – list, set, dict*.

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...

Solve the equation for each of the unknowns. We've shown that previously for $d = r * t$, the RTD calculation:

1. This leads to 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`, *Statements and Syntax*, for guidance on designing these complex `if...elif` chains. Include a variation on the `else` crash option:

```
else:
    warnings.warning( "Nothing to solve for" )
```

4. Build the resulting dictionary object. In simple cases, we can use the `vars()` function to simply emit all of the local variables as a resulting dictionary. In some cases, we may have local variables we don't want to include; in that case, we'll need to build the dictionary explicitly:

```
return dict(distance=distance, rate=rate, time=time)
```

5. Wrap all of this as a function using keyword parameters:

```
def rtd(distance=None, rate=None, time=None):
    if distance is None:
        distance = rate * time
    elif rate is None:
        rate = distance / time
    elif time is None:
        time = distance / rate
    else:
        warnings.warning( "Nothing to solve for" )
    return dict(distance=distance, rate=rate, time=time)
```

We can use the resulting function like this:

```
>>> def rtd(distance=None, rate=None, time=None):
...     if distance is None:
...         distance = rate * time
...     elif rate is None:
...         rate = distance / time
...     elif time is None:
...         time = distance / rate
...     else:
...         warnings.warning( "Nothing to solve for" )
...     return dict(distance=distance, rate=rate, time=time)
>>> rtd(distance=31.2, rate=6)
{'distance': 31.2, 'time': 5.2, 'rate': 6}
```

This shows us that going 31.2 nautical miles at a rate of 6 knots will take 5.2 hours.

For 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 the *Designing complex if...elif chains* recipe from `Chapter 2`, *Statements and Syntax*.

How it works...

Because we've provided default values for all of the parameters, we can provide argument values for 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 simply handy. It allows us to have 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 `**`. It is often shown like this:

```
def rtd2(distance, rate, time, **keywords):
    print(keywords)
```

Any additional keyword arguments are collected into a dictionary that is provided to the `**keywords` parameter. We can then call this function with extra parameters. Evaluate this function like this:

```
rtd2(rate=6, time=6.75, something_else=60)
```

We'll then see that the value of the `keywords` parameter is a dictionary object with the value of `{'something_else': 60}`. We can then use ordinary dictionary processing techniques on this structure. The keys and values in this dictionary are the names and values provided when the function was evaluated.

We can leverage this and insist that all arguments be provided with keywords:

```
def rtd2(**keywords):
    rate= keywords.get('rate', None)
    time= keywords.get('time', None)
    distance= keywords.get('distance', None)
    etc.
```

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.

(Returning a default of `None` is the default behavior of the `get()` method. Our example contains some redundancy to clarify the processing. For some very complex situations, we might have defaults other than `None`.)

This has the potential advantage of being slightly more flexible. It has the potential disadvantage of making the actual parameter names very hard to discern.

We can follow the *Writing Clear documentation strings with RST markup* recipe and provide a good docstring. It seems somehow better, though, to provide the parameter names explicitly as part of the Python code rather than implicitly through documentation.

See also

- We'll look at documentation of functions in the *Writing Clear documentation strings with RST markup* recipe

Forcing keyword-only arguments with the * separator

There are some situations where we have a large number of positional parameters to a function. Perhaps we've followed the *Designing functions with optional parameters* recipe and this leads us to design a function with so many parameters that it gets confusing.

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, there are too many parameters.

Getting ready

We'll look at a function that has a large number of parameters. We'll use a function which prepares a wind-chill table and writes 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.

The basic formula is this:

$$T_{wc}(T_a, V) = 13.12 + 0.6215T_a - 11.37V^{0.16} + 0.3965T_aV^{0.16}$$

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 from °F to °C: $C = 5(F-32) / 9$
- Convert windspeed from MPH, V_m , to KPH, V_k: $V_k = V_m \times 1.609344$
- The result needs to be converted from °C back to °F: $F = 32 + C(9/5)$

We won't fold these into this solution. We'll leave that as an exercise for the reader.

One approach for creating a wind-chill table is to create something like this:

```
import pathlib

def Twc(T, V):
    return 13.12 + 0.6215*T - 11.37*V**0.16 + 0.3965*T*V**0.16

def wind_chill(start_T, stop_T, step_T,
    start_V, stop_V, step_V, path):
    """Wind Chill Table."""
    with path.open('w', newline='') as target:
        writer= csv.writer(target)
        heading = [None]+list(range(start_T, stop_T, step_T))
        writer.writerow(heading)
        for V in range(start_V, stop_V, step_V):
            row = [V] + [Twc(T, V)
                for T in range(start_T, stop_T, step_T)]
            writer.writerow(row)
```

We've opened an output file using the `with` context. This follows the *Managing a context using the with statement* recipe in `Chapter 2`, *Statements and Syntax*. Within this context, we've created a write for the CSV output file. We'll look at this in more depth in `Chapter 8`, *Input/Output, Physical Format, Logical Layout*.

We've used an expression, `[None]+list(range(start_T, stop_T, step_T))`, to create a heading row. This expression includes a list literal and a generator expression that builds a list. We'll look at lists in `Chapter 4`, *Built-in Data Structures – list, set, dict*. We'll look at the generator expression in online `Chapter 12`, *Functional And Reactive Programming Features* (link provided in `Preface`).

Similarly, each cell of the table is built by a generator expression, `[Twc(T, V) for T in range(start_T, stop_T, step_T)]`. This is a comprehension that builds a list object. The list consists of values computed by the wind-chill function, `Twc()`. We provide the wind velocity based on the row in the table. We provide a temperature based on the column in the table.

While the details involve forward-looking sections, the `def` line presents a problem. This `def` line is very complex.

The problem with this design is that the `wind_chill()` function has seven positional parameters. When we try to use this function, we wind up with code like the following:

```
import pathlib
p=pathlib.Path('code/wc.csv')
wind_chill(0,-45,-5,0,20,2,p)
```

What are all those numbers? Is there something we can do to help explain what this line of code means?

How to do it...

When we have a large number of parameters, it helps to use keyword arguments instead of positional arguments.

In Python 3, we have a technique that mandates the use of keyword arguments. We can use the `*` as a separator between two groups of parameters:

1. Before the `*`, 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.
2. After the `*`, we list the argument values that must be given with a keyword. For our example, this is all of the parameters.

For our example, the resulting function looks like this:

```
    def wind_chill(*, start_T, stop_T, step_T, start_V, stop_V, step_V,
path):
```

When we try to use the confusing positional parameters, we'll see this:

```
>>> wind_chill(0,-45,-5,0,20,2,p)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: wind_chill() takes 0 positional arguments but 7 were given
```

We must use the function as follows:

```
    wind_chill(start_T=0, stop_T=-45, step_T=-5,
        start_V=0, stop_V=20, step_V=2,
        path=p)
```

This use of mandatory keyword parameters forces us to write a clear statement each time we use this complex function.

How it works...

The `*` character has two meanings in the definition of a function:

- It's used as a prefix for a special parameter that receives all the unmatched positional arguments. We often use `*args` to collect all of the positional arguments into a single parameter named `args`.
- It's used by itself, as a separator between parameters that may be applied positionally and parameters which must be provided by keyword.

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:

```
    import sys
    def wind_chill(*, start_T, stop_T, step_T, start_V, stop_V, step_V,
output=sys.stdout):
```

We can now use this function in two ways:

- Here's a way to print the table on the console:

```
wind_chill(
    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:

```
path = pathlib.Path("code/wc.csv")
with path.open('w', newline='') as target:
    wind_chill(output=target,
        start_T=0, stop_T=-45, step_T=-5,
        start_V=0, stop_V=20, step_V=2)
```

We've changed the approach here, to one which is slightly more generalized. This follows the *Designing functions with optional parameters* recipe.

See also

- See the *Picking an order for parameters based on partial functions* recipe for another application of this technique

Writing explicit types on function parameters

The Python language allows us to write functions (and classes) which are entirely generic with respect to data type. Consider this function as an example:

```
def temperature(*, f_temp=None, c_temp=None):
    if c_temp is None:
        return {'f_temp': f_temp, 'c_temp': 5*(f_temp-32)/9}
    elif f_temp is None:
        return {'f_temp': 32+9*c_temp/5, 'c_temp': c_temp}
    else:
        raise Exception("Logic Design Problem")
```

This follows three recipes shown earlier: *Using super flexible keyword parameters*, *Forcing keyword-only arguments with the * separator* from this chapter, and *Designing complex if...elif chains* from `Chapter 2`, *Statements and Syntax*.

This function will work for argument values of any numeric type. Indeed, it will work for any data structure that implements the `+`, `-`, `*`, and `/` operators.

There are times when we do not want our functions to be completely generic. In some cases, we would like to make some stronger assertions about data types. While we sometimes care about the data type, we do not want to write a lot of code that looks like this:

```
from numbers import Number
def c_temp(f_temp):
    assert isinstance(F, Number)
    return 5*(f_temp-32)/9
```

This introduces performance overhead of an extra `assert` statement. It also clutters our programs with a statement that – generally – should be restating the obvious.

Additionally, we can't rely on docstrings for testing purposes. Here's the recommended style:

```
def temperature(*, f_temp=None, c_temp=None):
    """Convert between Fahrenheit temperature and
    Celsius temperature.

    :key f_temp: Temperature in °F.
    :key c_temp: Temperature in °C.
    :returns: dictionary with two keys:
        :f_temp: Temperature in °F.
        :c_temp: Temperature in °C.
    """
```

The docstring doesn't allow any automated testing to confirm that the documentation actually matches the code. The two could disagree with each other.

What we want are hints about the type of data involved that can be used for testing and confirmation, but don't interfere with performance. How can we provide meaningful type hints?

Getting ready

We'll implement a version of the `temperature()` function. We'll need two modules that will help us provide hints regarding the data types for parameters and return values:

```
from typing import *
```

We've opted to import all of the names from the `typing` module. If we're going to supply type hints, we want them to be terse. It's awkward having to write `typing.List[str]`. We prefer to omit the module name.

We'll also need to install the latest version of `mypy`. This project is undergoing rapid development. Rather than use the `pip` program to get a copy from PyPI, it's better to download the latest version directly from the GitHub repository, `https://github.com/JukkaL/mypy`.

The instructions say that, *Currently, the version of mypy on PyPI is not compatible with Python 3.5. If you run Python 3.5 install directly from git*.

```
$ pip3 install git+git://github.com/JukkaL/mypy.git
```

The `mypy` tool can be used to analyze our Python programs to determine if the type hints match the actual code.

How to do it...

Python 3.5 introduces type hints to the language. We can use them in three places: function parameters, function returns, and type hint comments:

1. Define a handy type for a variety of numbers:

```
from decimal import Decimal
from typing import *
Number = Union[int, float, complex, Decimal]
```

   Ideally, we'd like to use the abstract `Number` class in the numbers module. Currently, this module doesn't have a formal type specification available, so we'll define our own expectation for `Number`. This definition is a union of several numeric types. Ideally, a future release of `mypy` or Python will include the needed definitions.

2. Annotate arguments to functions like this:

```
def temperature(*,
    f_temp: Optional[Number]=None,
    c_temp: Optional[Number]=None):
```

We've added `:` and a type hint as part of the parameter. In this case, we're using our own type definition of `Number` to state that any number is allowed here. We've wrapped this with the `Optional[]` type operation to state that the argument value can be either a `Number` or `None`.

3. Annotate return values from functions like this:

```
def temperature(*,
    f_temp: Optional[Number]=None,
    c_temp: Optional[Number]=None) -> Dict[str, Number]:
```

We've added `->` and a type hint for the return value of this function. In this case, we've stated that the result will be a dictionary object with keys that are strings, `str`, and values that are numbers using our type definition of `Number`.

The `typing` module introduces the type hint names such as `Dict` that we use to explain the results of a function. This is different from the `dict` class which actually builds objects. The `typing.Dict` is merely a hint.

4. If necessary, we can add type hints as comments in assignment and `with` statements. These are rarely needed, but may clarify a long, complex series of statements. If we wanted to add them, the annotations could look like this:

```
result = {'c_temp': c_temp,
    'f_temp': f_temp} # type: Dict[str, Number]
```

We've added `# type: Dict[str, Number]` on the statement that builds the final dictionary object.

How it works...

The type information we've added are called **hints**. They're not requirements that are somehow checked by the Python compiler. They're not checked at runtime either.

The type hints are used by a separate program, `mypy`. See `http://mypy-lang.org` for more information.

The `mypy` program examines the Python code, including the type hints. It applies some formal reasoning and inference techniques to determine if the various type hints will be `true` for any data that the Python program can process.

For larger and more complex programs, the output from `mypy` will include warnings and errors that describe potential problems with either the code itself, or the type hints decorating the code.

For example, here's a mistake that's easy to make. We've assumed that our function returns a single number. Our return statement, however, doesn't match our expectation:

```
def temperature_bad(*,
    f_temp: Optional[Number]=None,
    c_temp: Optional[Number]=None) -> Number:

    if c_temp is None:
        c_temp = 5*(f_temp-32)/9
    elif f_temp is None:
        f_temp = 32+9*c_temp/5
    else:
        raise Exception( "Logic Design Problem" )
    result = {'c_temp': c_temp,
        'f_temp': f_temp} # type: Dict[str, Number]
    return result
```

When we run `mypy`, we'll see this:

```
ch03_r04.py: note: In function "temperature_bad":
ch03_r04.py:37: error: Incompatible return value type:
    expected Union[builtins.int, builtins.float, builtins.complex,
decimal.Decimal],
    got builtins.dict[builtins.str,
    Union[builtins.int, builtins.float, builtins.complex,
decimal.Decimal]]
```

We can see our `Number` type name was expanded to `Union[builtins.int, builtins.float, builtins.complex, decimal.Decimal]` in the error message. More importantly, we can see that line 37, the `return` statement, doesn't match the function definition.

Given this error, we need to either fix the return or the definition to be sure that the expected type and the actual type match. It's not clear which is *right*. Either of these could be the intent:

- Compute and return a single value: This means that there would need to be two `return` statements depending on which value was computed. In this case, there's no reason to build the `result` dictionary object.

- Return the dictionary object: This means we need to correct the `def` statement to have the proper return type. Changing this may spread ripples of change to other functions that expect `temperature` to return a instance of `Number`.

The extra syntax for parameters and return values has no real impact on runtime, and only a very small cost when the source code is first compiled into byte code. They are—after all—merely hints.

There's more...

When using built-in types, we can often create elaborate structures. We might, for example, have a dictionary that maps tuples of three integers to lists of strings:

```
a = {(1, 2, 3): ['Poe', 'E'],
     (3, 4, 5): ['Near', 'a', 'Raven'],
    }
```

If this is the result of a function, how do we describe this?

We'll be creating a fairly complex type expression that summarizes each layer of the structure:

```
Dict[Tuple[int, int, int], List[str]]
```

We've summarized a dictionary that maps one type, `Tuple[int, int, int]`, as the key to another type, `List[str]`, as the value. This captures how several built-in types can be combined to build complex data structures.

In this case, we've treated the tuple of three integers as an otherwise anonymous tuple. In many cases, it's not merely a generic tuple, it's actually an RGB color which is modeled as a tuple. Perhaps the list of strings is really a line of text from a longer document that's been split on spaces into words.

In this case, we should do something like the following:

```
Color = Tuple[int, int, int]
Line = List[str]
Dict[Color, Line]
```

Creating our own application-specific type names can greatly clarify the processing that's being performed using the built-in collection types.

See also

- See `https://www.python.org/dev/peps/pep-0484/` for more information on type hints.
- See `https://github.com/JukkaL/mypy` for the current `mypy` project.
- See `http://www.mypy-lang.org` for documentation on how `mypy` works with Python 3.

Picking an order for parameters based on partial functions

When we look at complex functions, we'll sometimes see a pattern to 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.

It can simplify our programming if our design reflects this concern. We'd like to provide a way to make the common parameters slightly easier to work with than the uncommon parameters. We'd also like to avoid having to repeat the parameters that are part of a larger context.

Getting ready

We'll look at a version of the haversine formula. This computes distances between points on the surface of the Earth, using the latitude and longitude coordinates of that point:

$$a = \sin^2\left(\frac{lat_2 - lat_1}{2}\right) + \cos\left(lat_1\right)\cos\left(lat_2\right)\sin^2\left(\frac{lat_2 - lat_1}{2}\right)$$

$$c = 2arc\ sin(\sqrt{a})$$

The essential calculation yields the central angle, *c*, between two points. The angle is measured in radians. We convert it into distance by multiplying by the Earth's mean radius in some units. If we multiply the angle *c* by a radius of 3,959 miles, the distance, we'll convert the angle to miles.

Here's an implementation of this function. We've included type hints:

```
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 = sin(Δ_lat/2)**2 + cos(lat_1)*cos(lat_2)*sin(Δ_lon/2)**2
c = 2*asin(sqrt(a))

return R * c
```

Note on the doctest example: The doctest example uses an earth radius with an extra decimal point that's not used elsewhere. This is so that this example matches other examples online. The earth isn't spherical. Around the equator, a more accurate radius is 6378.1370 km. Across the poles, the radius is 6356.7523 km. We're using common approximations in the constants.

The problem we often have is that we're generally working in a single context, and we will be providing the same value for `R` all the time. If, for example, we're working in a marine environment, we'd always be using `R = NM` to get nautical miles.

There are two common approaches to providing a consistent value for an argument. We'll look at both.

How to do it...

In some cases, an overall context will establish a variable for a parameter. The value will rarely change. There are several common approaches to providing a consistent value for an argument. These involve wrapping the function in another function. There are several approaches:

- Wrap the function in a new function.
- Create a partial function. This has two further refinements:
    - We can provide keyword parameters
    - Or we can provide positional parameters

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

Wrapping a function

We can provide contextual values by wrapping a general function in a context-specific wrapper function:

1. Make some parameters positional and some parameters keywords. We want the contextual features—the ones which change rarely—to be keywords. The parameters which change more frequently should be left as positional. We can follow the *Forcing keyword-only arguments with the * separator* recipe. We might change the basic haversine function to look like this:

```
def haversine(lat_1: float, lon_1: float,
    lat_2: float, lon_2: float, *, R: float) -> float:
```

   We inserted the `*` to separate parameters into two groups. The first group can have arguments supplied either by position or by keyword. The second group, – `R`, in this case – must be given by keyword.

2. We can then write a wrapper function the 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(*args):
    return haversine(*args, R=NM)
```

We've the `*args` construct in the function declaration to accept all positional argument values in a single tuple, `args`. We've also `*args` when evaluating the `haversine()` function to expand the tuple into all of the positional argument values to this function.

Creating a partial function with keyword parameters

A partial function is a function which has some of the argument values supplied. When we evaluate a partial function, we're mixing the previously supplied parameters with additional parameters. One approach is to use keyword parameters, similar to wrapping a function:

1. We can follow the *Forcing keyword-only arguments with the * separator* recipe. We might change the basic haversine function to look like this:

```
def haversine(lat_1: float, lon_1: float,
    lat_2: float, lon_2: float, *, R: float) -> float:
```

2. Create a partial function using the keyword parameter:

```
from functools import partial
nm_haversine = 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()` 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(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 without having to patiently check that each time we used the `haversine()` function it had `R=NM` as an argument.

Creating a partial function with positional parameters

A partial function is a function which has some of the argument values supplied. When we evaluate a partial function, we're supplying additional parameters. An alternative approach is to use 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 might change the basic `haversine` function to look like this:

```
def haversine(R: float, lat_1: float, lon_1: float,
    lat_2: float, lon_2: float) -> float:
```

2. Create a partial function using the positional parameter:

```
from functools import partial
nm_haversine = partial(haversine, NM)
```

   The `partial()` function builds a new function from an existing function and a concrete set of argument values. The `nm_haversine()` function has a specific value for the first parameter, `R`, provided when the partial was built.

We can use this like we'd use any other function:

```
>>> round(nm_haversine(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 without having to patiently check that each time we used the `haversine()` function it had `R=NM` as an argument.

How it works...

The partial function is—essentially—identical to the wrapper function. While it saves us a line of code, it has a more important purpose. We can build partials freely in the middle of other, more complex pieces of a program. We don't need to use a `def` statement for this.

Note that creating partial functions leads to a few additional considerations when looking at the order for positional parameters:

- When we use `*args`, it must be last. This is a language requirement. It means that the parameters in front of this can be identified specifically, all the rest become anonymous and can be passed – *en masse* – to the wrapped function.
- The leftmost positional parameters are easiest to provide a value when creating a partial function.

These two considerations lead us to look at the leftmost argument as being more of a context: these are expected to change rarely. The rightmost parameters provide details and change frequently.

There's more...

There's a third way to wrap a function—we can also build a `lambda` object. This will also work:

```
nm_haversine = lambda *args: haversine(*args, R=NM)
```

Notice that a `lambda` object is a function that's been stripped of name and body. It's reduced to just two essentials:

- The parameter list
- A single expression that is the result

A `lambda` cannot have any statements. If we need statements, we need to use the `def` statement to create a definition that includes a name and a body with multiple statements.

See also

- We'll also look at further extending this design in the *Writing reusable scripts with the script library switch* recipe

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* in `Chapter 2`, *Statements and Syntax* and *Writing clear documentation strings with RST markup* recipes, 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 that had a large number of parameters and another function that had only two parameters.

Here's a slightly different version of one of those functions, `Twc()`:

```
>>> def Twc(T, V):
...     """Wind Chill Temperature."""
...     if V < 4.8 or T > 10.0:
...         raise ValueError("V must be over 4.8 kph, T must be below 10°C")
...     return 13.12 + 0.6215*T - 11.37*V**0.16 + 0.3965*T*V**0.16
```

We need to annotate this function with some more complete documentation.

Ideally, we've got Sphinx installed to see the fruits of our labor. See `http://www.sphinx-doc.org`.

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 nice documentation for a function. We can apply a similar recipe for a function, or even a module:

1. Write the synopsis: A proper subject isn't required—we don't write *This function computes...*; we start with *Computes...*. There's no reason to overstate the context:

```
def Twc(T, V):
    """Computes the wind chill temperature."""
```

2. Write the description with details:

```
def Twc(T, V):
    """Computes the wind chill temperature

    The wind-chill, :math:`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. If you have LaTeX installed, Sphinx will use that to prepare a little `.png` file with the math. If you want, Sphinx can use MathJax or JSMath to do JavaScript math typesetting instead of creating a `.png` file.

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 which must be keywords, it's common to use `:key name: description`. The word `key` instead of `param` shows that it's a keyword-only parameter:

```
def Twc(T: float, V: 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
    """
```

There are two ways to include type information:

- Using Python 3 type hints
- Using RST `:type name:` markup

We generally don't use both techniques. Type hints are a better idea than the RST `:type:` markup.

4. Describe the return value using `:returns:`:

```
def Twc(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
    """
```

There are two ways to include return type information:

- Using Python 3 type hints
- Using RST `:rtype:` markup

We generally don't use both techniques. The RST `:rtype:` markup has been superseded by type hints.

5. Identify the important exceptions that might be raised. Use the `:raises exception:` reason markup. There are several possible variations, but `:raises exception:` seems to be most popular:

```
def Twc(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 over 4.8 kph or T
above 10°C
    """
```

6. Include a doctest test case, if possible:

```
def Twc(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 over 4.8 kph or T
above 10°C
            >>> round(Twc(-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) = 13.12 + 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 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`, *Statements and Syntax*. 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 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 `Twc()` function.

Sphinx will generate these cross-references using a special `:func:` text role:

```
def wind_chill_table():
    """Uses :func:`Twc` to produce a wind-chill
    table for temperatures from -30°C to 10°C and
    wind speeds from 5kph to 50kph.
    """
```

We've used the `:func:`Twc`` to cross-reference one function in the RST documentation for a different function. Sphinx will turn these into proper hyperlinks.

See also

- See the *Including descriptions and documentation* and *Writing better RST markup in docstrings* recipes in `Chapter 2`, *Statements and Syntax*, 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:

The factorial function:

$$n! = \begin{cases} 1 & if\ \ n = 0 \\ n \times (n-1)!\ if\ \ n > 0 \end{cases}$$

The rule for computing Fibonacci numbers:

$$F_n = \begin{cases} 1 & if\ \ n = 0 \vee n = 1 \\ F_{n-1} + F_{n-2}\ if\ \ n > 0 \end{cases}$$

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 a limitation on the upper limit for these kinds of recursive function definitions. While Python's integers can easily represent *1000!*, the stack limit prevents us from doing this casually.

Computing F_n Fibonacci numbers involves an additional problem. If we're not careful, we'll compute a lot of values more than once:

$$F_5 = F_4 + F_3$$

$$F_5 = (F_3 + F_2) + (F_2 + F_1)$$

And so on.

To compute F_5, we'll compute F_3 twice, and F_2 three times. This is extremely costly.

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(n: int) -> int:
    if n == 0:
        return 1
    return n*fact(n-1)
```

The last expression in the function refers back to the function with 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: $n! = n \times (n\text{-}1) \times (n\text{-}2) \times (n\text{-}3)... \times 1$
2. Write a loop that enumerates all the values: $N = \{n, n\text{-}1, n\text{-}2, ..., 1\}$In Python, it's simply this: `range(1, n+1)`. In some cases, though, we might have to apply some transformation function to the base values: $N = \{f(i): 1 \le i < n+1\}$If we had to perform some kind of transformation, it might look like this in Python:

```
N = (f(i) for i in range(1,n+1))
```

Incorporate the reduction function. In this case, we're computing a large product, using multiplication. We can summarize this using $\prod x$ notation. For this example, we're only imposing a simple boundary on the values computed in the product:

$$\prod_{1 \le x \le n+1} x$$

Here's the implementation in Python:

```
def prod(int_iter):
    p = 1
    for x in int_iter:
        p *= x
    return p
```

We can restate this into a solution like this. This uses higher-level functions:

```
def fact(n):
    return prod(range(1, n+1))
```

This works nicely. We've optimized the first solution to combine the `prod()` and `fact()` functions into a single function. It turns out that doing that optimization doesn't actually shave much time off the operation.

Here are the comparisons, run using the `timeit` module:

Simple	4.7766
Optimized	4.6901

This is in the order of a 2% performance improvement. Not a significant change.

Note that the Python 3 `range` object is lazy—it doesn't create a big `list` object, it returns values as they are requested by the `prod()` function. This is different from Python 2, where the `range()` function eagerly created a big `list` object with all of the values, and the `xrange()` function was lazy.

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 are optimized in a manner similar to the technique we've shown. There are two parts to this optimization:

- Use relatively simple algebraic rules to reorder the statements so that the recursive clause is actually last. The `if` clauses can be reorganized into a different physical order so that the `return fact(n-1) * n` is last. This rearrangement is necessary for code organized like this:

```
def ugly_fact(n):
    if n > 0:
        return fact(n-1) * n
    elif n == 0:
        return 1
    else:
        raise Exception("Logic Error")
```

- Inject a special instruction into the virtual machine's byte code—or the actual machine code—that re-evaluates the function without creating a new stack frame. Python doesn't have this feature. In effect, this special instruction transforms the recursion into a kind of `while` statement:

```
p = n
while n != 1:
    n = n-1
    p *= n
```

This purely mechanical transformation leads to rather ugly code. In Python, it may also be remarkably slow. In other languages, the presence of the special byte code instruction will lead to code that runs quickly.

We prefer not to do this kind of mechanical optimization. First, it leads to ugly code. More importantly – in Python – it tends to create code that's actually slower than the alternative developed above.

There's more...

The Fibonacci problem involves two recursions. If we write it naively as a recursion, it might look like this:

```
def fibo(n):
    if n <= 1:
        return 1
    else:
        return fibo(n-1)+fibo(n-2)
```

It's difficult to do a simple mechanical transformation into a tail recursion. A problem with multiple recursions like this requires some more careful design.

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.lru_cache()` as a decorator. This function will cache previously computed values. This means that we'll only compute a value once; every other time, the `lru_cache` will return the previously computed value.

It looks like this:

```
from functools import lru_cache

@lru_cache(128)
def fibo(n):
    if n <= 1:
        return 1
    else:
        return fibo(n-1)+fibo(n-2)
```

Adding a decorator is a simple way to optimize a more complex multi-way 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 F_n. We only want the last value in this sequence. We compute all the intermediates because it's more efficient to do it that way. Here's a generator function that does this:

```
def fibo_iter():
    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(n):
    """
    >>> fibo(7)
    21
    """
    for i, f_i in enumerate(fibo_iter()):
        if i == n: break
    return f_i
```

This function consumes each value from the `fibo_iter()` iterator. When the desired number has been reached, the `break` statement ends the `for` statement.

When we look back at the *Designing a while statement which terminates properly* recipe in `Chapter 2`, *Statements and Syntax*, we noted that a `while` statement with a `break` may have multiple reasons for terminating. In this example, there is only one way to end the `for` statement.

We can always assert that `i == n` at the end of the loop. This simplifies the design of the function.

See also

- See the *Designing a while statement which terminates properly* recipe in `Chapter 2`, *Statements and Syntax*

Writing reusable scripts with the script library switch

It's common to create small scripts which we want to combine into a larger script. We don't want to copy and paste the code. We want to leave the working code in one file and use it in multiple places. Often we want to combine elements from multiple files to create more sophisticated scripts.

The problem we have is that when we import a script it actually starts running. This is generally not what we expect when we import a script so that we can reuse it.

How can we import the functions (or classes) from a file without having the script start doing something?

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 `ch03_r08.py`.

Initially, the file might look like this:

```
import csv
import pathlib
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:
    ... and more ...

nm_haversine = partial(haversine, R=NM)
```

```
source_path = pathlib.Path("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
```

We've omitted the body of the `haversine()` function, showing only `... and more...`, since it's shown in the *Picking an order for parameters based on partial functions* recipe. We've focused on the context in which the function is in a Python script that also opens a file, `wapypoints.csv`, and does some processing on that file.

How can we import this module without it printing a display of distances between waypoints in our `waypoints.csv` file?

How to do it...

Python scripts can be simple to write. Indeed, it's often too simple to create a working script. Here's how we transform a simple script into a reusable library:

1. Identify the statements that do the work of the script: we'll distinguish between *definition* and *action*. Statements such as `import`, `def`, and `class` are clearly definitional—they support the work but they don't do the work. Almost all other statements take action. In our example, we have four assignment statements that are more definition than action. The distinction is entirely one of intent. All statements, by definition, take an action. These actions, though, are more like the action of the `def` statement than they are like the action of the `with` statement later in the script. Here are the generally definitional statements:

```
MI= 3959
NM= 3440
KM= 6373

def haversine( lat_1: float, lon_1: float,
    lat_2: float, lon_2: float, *, R: float ) -> float:
    ... and more ...

nm_haversine = partial(haversine, R=NM)
```

The rest of the statements clearly take an action toward producing the printed results.

2. Wrap the actions into a function:

```
def analyze():
    source_path = pathlib.Path("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
```

3. Where possible, extract literals and make them into parameters. This is often a simple movement of the literal to a parameter with a default value. From this:

```
def analyze():
    source_path = pathlib.Path("waypoints.csv")
```

To this:

```
def analyze(source_name="waypoints.csv"):
    source_path = pathlib.Path(source_name)
```

This makes the script reusable because the path is now a parameter instead of an assumption.

4. Include the following as the only high-level action statements in the script file:

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

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.

How it works...

The most 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.

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.

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.

When a file is being imported, however, the value of `__name__` is set to the name of the module. It will not be `__main__`. In our example, the value `__name__` during `import` processing will be `ch03_r08`.

There's more...

We can now build useful work around a reusable library. We might make several files that look like this:

File `trip_1.py`:

```
from ch03_r08 import analyze
analyze('trip_1.csv')
```

Or perhaps something even more complex:

File `all_trips.py`:

```
from ch03_r08 import analyze
for trip in 'trip_1.csv', 'trip_2.csv':
    analyze(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

To get to this goal, we'll 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 should evolve towards 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 doesn't get us any higher on the route, but it enables us to climb safely.

See also

- In `Chapter 6`, *Basics of Classes and Objects*, we'll look at class definitions. These are another kind of widely used definitional statement.

4
Built-in Data Structures – list, set, dict

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

- Choosing a data structure
- Building lists – literals, appending, and comprehensions
- Slicing and dicing a list
- Deleting from a list – deleting, removing, popping, and filtering
- Reversing a copy of a list
- Using set methods and operators
- Removing items from a set – remove(), pop(), and difference
- Creating dictionaries – inserting and updating
- Removing from dictionaries – the pop() method and the del statement
- Controlling the order of dict keys
- Handling dictionaries and sets in doctest examples
- Understanding variables, references, and assignment
- Making shallow and deep copies of objects
- Avoiding mutable default values for function parameters

Introduction

Python has a rich collection of built-in data structures. A great deal of useful programming is commonly done with these built-in structures. These collections cover a variety of common situations.

We'll look at an overview of the various structures that are available and what problems they solve. From there, we can look at lists, dictionaries, and sets in detail.

Note that we've set the built-in tuple and string aside as being different from the list structure. There are some important similarities as well as some differences. In `Chapter 1`, *Numbers, Strings, and Tuples*, we emphasized the way strings and tuples behave more like immutable numbers than mutable collections.

We'll also look at some more advanced topics related to how python handles references to objects. We'll look at some issues related to the mutability of these data structures, as well.

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 determine which is appropriate for a given purpose.

How do we choose which structure to use? What are the features of lists, sets, and dictionaries? Why do we have tuples and frozen sets?

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.

How to do it...

1. Is the programming focused on doing membership tests? An example of this is a collection of valid input values. When the user enters something that's in the collection, their input is valid, otherwise it's invalid.

   Simple membership suggests using a `set`:

```
valid_inputs = {"yes", "y", "no", "n"}
answer = None
while answer not in valid_inputs:
    answer = input("Continue? [y, n] ").lower()
```

A `set` holds items in no particular order. Once an item is a member, we can't add it again:

```
>>> valid_inputs = {"yes", "y", "no", "n"}
>>> valid_inputs.add("y")
>>> valid_inputs
{'no', 'y', 'n', 'yes'}
```

We have created a set, `valid_inputs`, with four distinct string items. We can't add another `y` to a set which already contains `y`. The contents of the set doesn't change.

Also note that the order of the items in the set isn't exactly the order in which we initially provided them. A set can't maintain any particular order to the items, it can only determine if an item exists in the set.

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 must 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. We can pick an item by providing its position. We can also use the `index()` method to locate the index of an item in the list.

Lists in Python always start with position zero. This is true for tuples and strings, also.

If the number of items in the collection is fixed—for example RGB colors have three values—then we might be looking at 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 that's not the item's position? An example might include a mapping between strings of characters—words—and integers which represent the frequencies of those words, or a mapping between a color name and the RGB tuple for that color.

   When we must identify items with a non-positional key, we're using some kind of mapping. The built-in mapping is `dict`. There are several extensions that add more features:

```
>>> scheme = {"Crimson": (220, 14, 60),
... "DarkCyan": (0, 139, 139),
... "Yellow": (255, 255, 00)}
>>> scheme['Crimson']
(220, 14, 60)
```

   In this dictionary, `scheme`, we've created a mapping from color names to the RGB color tuples. When we use a key, for example `"Crimson"`, we can retrieve the value bound to that key.

4. Consider the mutability of items in a `set` collection and the keys in a `dict` collection. Each item in a set must be an immutable object. Numbers, strings, and tuples are all immutable, and can be collected into sets. Since a `list`, `dict`, or `set` object is mutable, they can't be used as items in a set. It's impossible to build a `set` of `list` items, for example.

   Rather than create a `set` of `list` items, we can transform each `list` item into a `tuple`. We can create a `set` of immutable `tuple` items.

   Similarly, dictionary keys must be immutable. We can use a number, or a string, or a tuple as a dictionary key. We can't use a `list`, or a `set`, or another mutable mapping 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 identify the unique features of each collection.

It turns out that the `collections.abc` module provides a kind of road map through the built-in collections. The `collections.abc` module defines the **Abstract Base Classes** (**ABC**s) that support 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 actually places for a total of six kinds of collections:

- **Set**: The unique feature is that items are either members or not. This means duplicates can't be handled:
    - **Mutable set**: The `set` collection
    - **Immutable set**: The `frozenset` collection
- **Sequence**: The unique feature is that items are provided with an index position:
    - **Mutable sequence**: The `list` collection
    - **Immutable sequence**: The `tuple` collection
- **Mapping**: The unique feature is that each item has a key that refers to a value:
    - **Mutable mapping**: The `dict` collection
    - **Immutable mapping**: Interestingly, there's no built-in frozen mapping

Python's libraries offer a large number of additional implementations of these core collection types. We can see many of these in the *Python Standard Library*.

The `collections` module contains a number of variations on the built-in collections. These include:

- `namedtuple`: A `tuple` that offers names for each item in a tuple. It's slightly more clear 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 the `deque` is more efficient.
- `defaultdict`: A `dict` that can provide a default value for a missing key.
- `Counter`: A `dict` which is designed to count occurrences of a key. This is sometimes called a multiset or a bag.
- `OrderedDict`: A `dict` which retains the order in which keys were created.
- `ChainMap`: A `dict` which combines several dictionaries into a single mapping.

There's more in the *Python Standard Library*. We can also use the `heapq` module which defines a priority queue implementation. The `bisect` module includes methods for searching a sorted list very quickly. This allows a list to have performance that is a little closer to the fast lookups of a dictionary.

There's more...

We can look at a list of data structures like this: `https://en.wikipedia.org/wiki/List_of_data_structures`.

There are some important summaries that are part of this giant index of data structures. Different parts of the article provide slightly different summaries of data structures. We'll take a quick look at four classifications.

- **Arrays**: There are variant implementations that offer similar features. Python's `list` structure is typical, and offers performance similar to a linked-list implementation of an array.
- **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, rather than a data structure with a unique feature set.
- **Hashes**: Python uses hashes to implement dictionaries and sets. This leads to good speed but 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.

We can—with a little cleverness—implement almost any kind of data structure in Python. Either the built-in structures have the essential features, or we can locate a built-in structure that can be pressed into service.

See also

- For advanced graph manipulation, see `https://networkx.github.io`.

Building lists – literals, appending, and comprehensions

If we've decided to create a collection that uses an item's position—a `list`—we have several ways of building this structure. We'll look at a number of ways we can build a list object from individual items.

In some cases, we'll need a list because it allows duplicate values. A great many statistical operations don't require knowing the position of an item. A multiset would be useful for this, but we don't have this as a built-in structure; it's very common to use a `list` instead of a multiset.

Getting ready

Let's say we need to do some statistical analyses on some file sizes. Here's a short script that will provide us with the sizes of some files:

```
>>> import pathlib
>>> home = pathlib.Path('source')
>>> for path in home.glob('*/index.rst'):
...     print(path.stat().st_size, path.parent)
2353 source/ch_01_numbers_strings_and_tuples
2889 source/ch_02_statements_and_syntax
2195 source/ch_03_functions
3094 source/ch_04_built_in_data_structures_list_tuple_set_dict
725 source/ch_05_user_inputs_and_outputs
1099 source/ch_06_basics_of_classes_and_objects
690 source/ch_07_more_advanced_class_design
1207 source/ch_08_functional_programming_features
926 source/ch_09_input_output_physical_format_logical_layout
758 source/ch_10_statistical_programming_and_linear_regression
615 source/ch_11_testing
521 source/ch_12_web_services
1320 source/ch_13_application_integration
```

We've used a `pathlib.Path` object to represent a directory in our file system. The `glob()` method expands all names that match a given pattern. In this case, we used a pattern of `'*/index.rst'`. We can use the `for` statement to display the size from the file's OS `stat` data.

We'd like to accumulate a `list` object that has the various file sizes. From that we can compute total size, and average size. We can look for files which seem too large or too small.

We have four ways to create `list` objects:

- We can create literal display of a `list` using a sequence of values surrounded by `[]` characters. It looks like this: `[value, ... ]`. Python needs to match the `[` and `]` to see a complete logical line, so the literal can span physical lines. For more information see the *Writing long lines of code* recipe in `Chapter 2`, *Statements and Syntax*.

```
[2353, 2889, 2195, 3094, 725,
1099, 690, 1207, 926, 758,
615, 521, 1320]
```

- We can convert some other data collection into a list using the `list()` function. We can convert a `set`, or the keys of a `dict`, or the values of a `dict`. We'll look at a more sophisticated example of this in the *Slicing and dicing a list* recipe.
- We have `list` methods that allow us to build a `list` one item a time. These methods include `append()`, `extend()` and `insert()`. We'll look at `append()` in the *Building a list with the append() method* section of this recipe. We'll look at the other methods in the *There's More...* section of this recipe.
- We have generator expressions which can be used to build `list` objects. One kind of generator is a list comprehension.

How to do it...

Building a list with the append() method

1. Create an empty list, `[]`:

```
>>> file_sizes = []
```

2. Iterate through some source of data. Append the items to the list using the `append()` method:

```
>>> home = pathlib.Path('source')
>>> for path in home.glob('*/index.rst'):
...     file_sizes.append(path.stat().st_size)
>>> print(file_sizes)
```

```
[2353, 2889, 2195, 3094, 725, 1099, 690,
1207, 926, 758, 615, 521, 1320]
>>> print(sum(file_sizes))
18392
```

We used the path's `glob()` method to find all files that match the given pattern. The `stat()` method of a path provides the OS **stat** data structure, which includes the size, `st_size`, in bytes.

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.

Generally, any method that mutates an object has no return value. Methods like `append()`, `extend()`, `sort()`, and `reverse()` have no return value. They adjust the structure of the `list` object itself. The `append()` method does not return a value. It mutates the `list` object. It's surprisingly common to see wrong code like this: `a = ['some', 'data'] a = a.append('more data')` This is emphatically wrong. This will set `a` to `None`. The correct approach is a statement like this, without any additional assignment: `a.append('more data')`

Writing a list comprehension

The goal of a list comprehension is to create an object that occupies a syntax role similar to a list literal:

1. Write the wrapping `[]` brackets that surround the list object.
2. Write the source of the data. This will include the target variable. Note that there's no `:` at the end because we're not writing a complete statement:

```
for path in home.glob('*/index.rst')
```

3. Prefix this with the expression to evaluate for each value of the target variable. Again, since this is a simple expression we cannot use complex statements here:

```
path.stat().st_size
    for path in home.glob('*/index.rst')
```

In some cases, we'll need to add a filter. This is an `if` clause after the `for` clause. We can make the generator expression quite sophisticated.

Here's the entire `list` object:

```
>>> [path.stat().st_size
...     for path in home.glob('*/index.rst')]
[2353, 2889, 2195, 3094, 725, 1099, 690, 1207, 926, 758, 615, 521, 1320]
```

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 includes a generator expression, called a **comprehension** in the language manual. The generator expression is a data expression attached to a `for` clause. Since this generator is an expression, not a complete statement, there are some limitations on what it can do. The data expression is evaluated repeatedly, and is controlled by the `for` clause.

Using the list function on a generator expression

We'll create a `list` function that uses the generator expression:

1. Write the wrapping `list()` function that surrounds the generator expression.
2. We'll reuse steps two and three from the list comprehension version to create a generator expression. Here's the generator expression:

```
path.stat().st_size
    for path in home.glob('*/index.rst')
```

Here's the entire list object:

```
>>> list(path.stat().st_size
...     for path in home.glob('*/index.rst'))
[2353, 2889, 2195, 3094, 725, 1099, 690, 1207, 926, 758, 615, 521, 1320]
```

How it works...

A Python `list` object has a dynamic size. The bounds of the array are adjusted when items are appended or inserted, or the `list` is extended with another `list`. Similarly, the bounds shrink when items are popped or deleted. We can access any item very quickly, and the speed of access doesn't depend on the size of the list.

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:

```
some_list = [None for i in range(100)]
```

This will create a list with an initial size of 100 items, each of which is `None`. It's rare to need this, though, because lists can grow in size as needed.

The list comprehension syntax and the `list()` function both consume items from a generator and append them to create a new `list` object.

There's more...

Our goal in creating a `list` object was 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('*/index.rst'))
>>> sum(sizes)
18392
>>> max(sizes)
3094
>>> min(sizes)
521
>>> from statistics import mean
>>> round(mean(sizes), 3)
1414.769
```

We've used the built-in `sum()`, `min()`, and `max()` 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))
11
```

We've found the minimum, and then used the `index()` method to locate the position of that minimal value. Recall that the index values start at zero, so the smallest file is for the twelfth chapter.

Other ways to extend a list

We can extend a list, as well as insert 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 `+`:

```
>>> ch1 = list(path.stat().st_size
...     for path in home.glob('ch_01*/*.rst'))
>>> ch2 = list(path.stat().st_size
...     for path in home.glob('ch_02*/*.rst'))
>>> len(ch1)
13
>>> len(ch2)
12
>>> final = ch1 + ch2
>>> len(final)
25
>>> sum(final)
104898
```

We have created a list of sizes of documents with names like `ch_01*/*.rst`. We then created a second list of sizes of documents with a slightly different name pattern, `ch_02*/*.rst`. We then combined the two lists into a final list.

We can do this using the `extend()` method, also. We'll reuse the two lists and build a new list from them:

```
>>> final_ex = []
>>> final_ex.extend(ch1)
>>> final_ex.extend(ch2)
>>> len(final_ex)
25
>>> sum(final_ex)
104898
```

We noted that `append()` does not return a value. Note that `extend()` does not return a value, either. The `extend()` method mutates the `list` object.

We can insert a value prior to any particular position in a list, also. 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 `append()` and `extend()`, `insert()` does not return a value. It mutates the `list` object.

See also

- See the *Slicing and dicing a list* recipe for ways to copy lists and pick sublists from a list.
- See the *Deleting from a list – deleting, removing, popping, and filtering* 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 that $O(1)$ means that the cost is essentially constant, and $O(n)$ means the cost varies with the index of the item we're trying to process. This means that the cost grows as the size of the collection grows.

Slicing and dicing a list

There are many times when we 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, also.

Getting ready

We have a spreadsheet that was used to record fuel consumption on a large sailboat. It has rows which look like this:

date	**engine on**	**fuel height**
	engine off	
	Other notes	
10/25/2013	08:24	29
	13:15	27
	calm seas—anchor solomon's island	
10/26/2013	09:12	27
	18:25	22
	choppy—anchor in jackson's creek	

Fuel height? Yes. There's no float sensor to estimate the level of fuel in the tanks. Instead there's a **sight-gauge** that allows direct observation of the fuel. It's calibrated in inches of depth. For all practical purposes the tank is rectangular, so the depth shown can be converted to volume pretty easily—31 inches of depth is about 75 gallons.

What's important is that the spreadsheet data is not properly normalized. Ideally, each row follows the first normal form for data with each row having identical content, and each cell having only atomic values.

Our data is not properly normalized. We have four rows of headings. This is something the `csv` module can't deal with directly. We need to do some slicing to remove the rows from other notes. We'd like to combine the two rows of each day's travel to make it easier to compute an elapsed time and the number of inches used.

We can read the data like this:

```
>>> from pathlib import Path
>>> import csv
>>> with Path('code/fuel.csv').open() as source_file:
...     reader = csv.reader(source_file)
...     log_rows = list(reader)
>>> log_rows[0]
['date', 'engine on', 'fuel height']
>>> log_rows[-1]
['', "choppy -- anchor in jackson's creek", '']
```

We've used the `csv` module to read the log details. A `csv.reader()` is an iterable object. In order to collect the items into a single list, we applied the `list()` function. We looked at the first and last item in the list to confirm that we really have a list of lists structure.

Each row of the original CSV file is a list. Each of those lists is a three item sublist.

For this recipe, we'll use an extension of a list index expression to slice items from a list. The slice, like the index, follows the list object in `[]` characters. Python offers us several variations on the slice expression. A slice can include two or three values in the slice, separated by `:` characters. We can write `:stop`, `start:`, `start:stop`, `start:stop:step`, or any of several other variations. The default step value is one. The default start value is the beginning of the list and the default stop value is the end of the list.

Here's how we can slice and dice the raw list of rows to pick out the rows we need:

How to do it...

1. The first thing we need to do is remove the four lines of heading from the list of rows. We'll use two partial slice expressions to divide the list at row four:

```
>>> 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 `head` variable will have the four lines of headings. We don't really want to do any processing with the head, so we ignore that variable. The `tail` variable, however, has the rows of the sheet we care about.

2. We'll use slices with steps to pick the interesting rows. The `[start::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 and the other slice starts on row one.

Here's a slice of every third row, starting with row zero:

```
>>> tail[0::3]
[['10/25/13', '08:24:00 AM', '29'],
 ['10/26/13', '09:12:00 AM', '27']]
```

We'll also want every third row, starting with row one:

```
>>> tail[1::3]
[['', '01:15:00 PM', '27'], ['', '06:25:00 PM', '22']]
```

3. These two slices can then be zipped together:

```
>>> list( zip(tail[0::3], tail[1::3]) )
[(['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 sliced the list into two parallel groups:

- The `[0::3]` slice starts with the first row, and includes every third row. This will be rows zero, three, six, nine, and so on.
- The `[1::3]` slice starts with the second row, and includes every third row. This will be rows one, four, seven, ten, and so on.

We've used the `zip()` function to interleave these two sequences from the list. This gives us a sequence of three tuples that's very close to something we can work with.

4. Flatten the results:

```
>>> paired_rows = list( zip(tail[0::3], tail[1::3]) )
>>> [a+b for a,b in 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']]
```

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. Now we're in a position to convert the date and time into a single `datetime` value. We can then compute the difference in times to get the running time for the boat, and the difference in heights to estimate the fuel burned.

How it works...

The slice operator has several different common forms:

- `[:]`: The start and stop are implied. The expression `S[:]` will copy the sequence, *S*.
- `[:stop]`: This makes a new list from the beginning to just before the stop value.
- `[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, the end 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 $i \in \left\{ s \times n : 0 \le n < \frac{|L|}{s} \right\}$.
- `[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 $i \in \left\{ a + s \times n : 0 \le n < \frac{|L| - a}{s} \right\}$.
- `[: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 after stop will not be used.

The slicing technique works for lists, tuples, strings, and any other kind of sequence. This does not cause the object to be mutated; this will make a copy of the items.

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 is called a **shallow copy** because we'll have two collections that 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 in online `Chapter 12`, *Functional And Reactive Programming Features* (link provided in `Preface`).

See also

- See the *Building lists – literals, appending, and comprehensions* recipe for ways to create lists
- See the *Deleting from a list – deleting, removing, popping, and filtering* 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

Deleting from a list – deleting, removing, popping, and filtering

There are many times when we want to remove items from a `list` collection. We might delete items from a list, and then process the items which are left over.

Removing unneeded items has a similar effect as using the `filter()` to create a copy which 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 list.

Getting ready

We have a spreadsheet that is used to record fuel consumption on a large sailboat. It has rows which look like this:

date	**engine on**	**fuel height**
	engine off	
	Other notes	
10/25/2013	08:24	29
	13:15	27
	calm seas—anchor solomon's island	
10/26/2013	09:12	27

	18:25	22
	choppy—anchor in jackson's creek	

For more background on this data, see the *Slicing and dicing a list* recipe.

We can read the data like this:

```
>>> from pathlib import Path
>>> import csv
>>> with Path('code/fuel.csv').open() as source_file:
...     reader = csv.reader(source_file)
...     log_rows = list(reader)
>>> log_rows[0]
['date', 'engine on', 'fuel height']
>>> log_rows[-1]
['', "choppy -- anchor in jackson's creek", '']
```

We've used the `csv` module to read the log details. A `csv.reader()` is an iterable object. In order to collect the items into a single list, we applied the `list()` function. We looked at the first and last item in the list to confirm that we really have a list-of-lists structure.

Each row of the original CSV file is a list. Each of those lists has three items.

How to do it...

We'll look at four ways to remove things from a list:

- The `del` statement
- The `remove()` method
- The `pop()` method
- Using the `filter()` function to create a copy that rejects selected rows

Deleting items from a list

We can remove items from a list using the `del` statement.

To make it easy to follow the examples at the interactive prompt, we'll make a copy of the list. If we deleted rows from the original `log_rows` list, subsequent examples might be hard to follow. In a practical program, we would not make this extra copy. We could also have used `log_rows[:]` to copy the original list.

```
>>> tail = log_rows.copy()
```

Here's how the `del` statement looks:

```
>>> del tail[:4]
>>> tail[0]
['10/25/13', '08:24:00 AM', '29']
>>> tail[-1]
['', "choppy -- anchor in jackson's creek", '']
```

The `del` statement removed the header rows from the tail, leaving behind the rows that we really need to process. We can then combine these 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. 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']
```

This has a useless `''` string in it:

```
>>> 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. This is an important distinction that applies to mutable objects.

The `remove()` method does not return a value. It mutates the list object. It's surprisingly common to see wrong code like this: `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:

```
>>> 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
- It returns the value which was removed

The filter() function

We can also remove items by building a copy that passes the desirable items and rejects the undesirable items. Here's how we can do this with the `filter()` function.

1. Identify the features of the items we wish to pass or reject. The `filter()` function expects a rule for passing data. The logical inverse of that function will reject data.

   In our case, the rows we want have a numeric value in column two. We can best detect this with a little helper function.

2. Write the filter test function. If it's trivial, use a lambda object. Otherwise, write a separate function:

```
>>> def number_column(row, column=2):
...     try:
...         float(row[column])
...         return True
...     except ValueError:
...         return False
```

We've used the built-in `float()` function to see if a given string is a proper number. If the `float()` function does not raise an exception, the data is a valid number, and we want to pass this row. If an exception is raised, the data was not numeric, and we'll reject the row.

3. Use the filter test function (or lambda) with the data in the `filter()` function:

```
>>> tail_rows = list(filter(number_column, log_rows))
>>> len(tail_rows)
4
>>> tail_rows[0]
['10/25/13', '08:24:00 AM', '29']
>>> tail_rows[-1]
['', '06:25:00 PM', '22']
```

We provided our test, `number_column()` and the original data, `log_rows`. The output from the `filter()` function is an iterable. To create a list from the iterable result, we'll use the `list()` function. The result has just the four rows we want; the remaining rows were rejected.

We haven't really deleted the rows. We've created a copy which omits those rows. The end result is the same.

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. All three collections are sequences, but only the list is mutable.

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.

For example:

```
>>> row = ['', '06:25:00 PM', '22']
>>> del row[3]
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
    del row[3]
IndexError: list assignment index out of range
```

There's more...

There are some times where this doesn't work. If we use a list in a `for` statement, we can't delete items from the list.

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 result is clearly not right. Why are some even-valued items left in the list?

Let's look at what happens when processing the item with a value of eight. We'll execute the `remove()` method. The value will be removed, and all the subsequent values will be slid forward one position. The `10` will be moved into the position formerly occupied by the `8`. The list's internal index will move forward to the next position, which will have a `13` in it. The `10` will never be processed.

Bad things also happen if we insert into the middle of a list, the driving iterable in a `for` loop. In that case, items will be processed twice.

We have two ways to avoid the *skip-when-delete* problem:

- Make a copy of the list:

  ```
  for f in data_items[:]:
  ```

- Use a `while` loop with a manual index:

  ```
  >>> data_items = [1, 1, 2, 3, 5, 8, 10,
  ...     13, 21, 34, 36, 55]
  >>> position = 0
  >>> while position != len(data_items):
  ...     f= data_items[position]
  ...     if f%2 == 0:
  ...         data_items.remove(f)
  ...     else:
  ...         position += 1
  >>> data_items
  [1, 1, 3, 5, 13, 21, 55]
  ```

We've designed a loop which only increments the position if the item is odd. If an item is even it's removed, and the other items are moved forward one position in the list.

See also

- See the *Building lists – literals, appending, and comprehensions* recipe for ways to create lists
- See 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

Reversing a copy of a list

Once in a while, we need to reverse the order of the items in a `list` collection. Some algorithms, for example, produce results in a reversed order. We'll look at the way numbers converted to 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.

We have two ways to reverse a list. First, there's the `reverse()` method. Then there's this handy trick.

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:

$$v = d_n \times b^n + d_{n-1} \times b^{n-1} + d_{n-2} \times b^{n-2} + ... + d_1 \times b + d_0$$

A rational number has a finite number of digits. An irrational number would have an infinite series of digits.

For example, the number `0xBEEF` is a base 16 value. The digits are $\{B = 11, E = 14, F = 15\}$, the base $b = 16$.

$$48879 = 11 \times 16^3 + 14 \times 16^2 + 14 \times 16 + 15$$

We can restate this in a form that's slightly more efficient to compute:

$$v = (...((d_n \times b + d_{n-1}) \times b + d_{n-2}) \times b + ... + d_1) \times b + d_0$$

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.

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
```

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, see the *Choosing between True Division and Floor Division* recipe.

The algorithm for converting a timestamp in seconds, t_s, to individual week, day, and time fields looks like this:

$$t_m, s \leftarrow t_s/60, t_s \bmod 60$$

$$t_h, m \leftarrow t_m/60, t_m \bmod 60$$

$$t_d, h \leftarrow t_h/60, t_h \bmod 24$$

$$w, d \leftarrow t_d/60, t_d \bmod 7$$

This has a handy pattern that leads to a very simple implementation. It has a consequence of producing the values in reverse order:

```
>>> t_s = 8063599
>>> fields = []
>>> for b in 60, 60, 24, 7:
...     t_s, f = divmod(t_s, b)
...     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 two approaches: we can use the `reverse()` method or we can use a `[::-1]` slice expression. 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 an error to use a statement like this: `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]` which uses an implied start and stop, and the step was `-1`. This picks all of 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.

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 two methods:

- The `reverse()` function 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.

See also

- See 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
- See the *Building lists – literals, appending, and comprehensions* recipe for ways to create lists
- See the *Slicing and dicing a list* recipe for ways to copy lists and pick sublists from a list
- See the *Deleting from a list – deleting, removing, popping, and filtering* recipe for other ways to remove items from a list

Using set methods and operators

We have several ways to build a `set` collection. We can use the `set()` function to convert an existing collection to a set. We can use the `add()` method to put items into a set. We can also use the `update()` method and the union operator, `|`, to create a larger set from other sets.

We'll show a recipe that uses a `set` to show whether or not we've seen a complete domain of values from a pool of statistical data. The recipe will build a `set` collection as the samples are being scanned.

When doing exploratory data analysis, we need to answer the question: *Is this data random?* Many data collections have variances in the data that are ordinary noise. It's important not to waste time doing complex modeling and analysis of random numbers.

For discrete or continuous numeric data, like the depth of water in meters, or the size of a file in bytes, we can use averages and standard deviations to see if a given collection of data is random. We expect a sample's mean to match the population mean within boundaries that are measured by the standard deviation.

For categorical data, like customer ID numbers or phone numbers, we can't compute averages or standard deviations. These values have to be evaluated in a different way.

One technique for determining the randomness of categorical data is the **Coupon Collector's Test**. With this test, we will see how many items have to be examined before we have found a complete set of *coupons*. Is a sequence of customer visits random? Or is there some other distribution in the sequence of visits? If the data is not random, then we can invest in more research into the causes.

The Python `set` collection is central to how this works. We'll add items to a `set` until we've seen each customer once.

If customers arrive randomly, we can predict an expected number of visits before the business has seen each customer at least once. The overall expected arrival time for the entire domain is the sum of the arrival times for each customer in the domain. This is equal to the number of customers, n, times the n^{th} Harmonic Number, H_n:

$$E = n \times H_n = n \times ((1/1) + (1/2) + (1/3) + (1/n))$$

This is the expected average number of visits before all customers have been seen. If the actual average arrival time matches this expectation that means all customers are visiting; we don't need to waste any more time on studying data that fits our expectations. If the actual average doesn't match expectations, then some customers are not visiting as frequently as others, and we need to pursue a deeper study of why.

Getting ready

We'll use a Python `set` to represent the collection of coupons. We'll need a population of data that may (or may not) have a proper distribution of *coupons*. We'll look at a set of eight customers.

Here's a function that simulates customers arriving in a random order. The customers are represented as numbers in the half-open interval [0, n], we can say that all customers, c, fit the rule $0 \le c < n$.

```
>>> import random
>>> def arrival1(n=8):
...     while True:
...         yield random.randrange(n)
```

The `arrival1()` function will yield an endless sequence of values. We've called this `arrival` with a `1` on the end. It may look like a spelling mistake, but we've used the `1` suffix so that we can create alternative implementations.

We need to put an upper bound on the number of values generated. Here's a function that has an upper limit on the number of samples produced:

```
>>> def samples(limit, generator):
...     for n, value in enumerate(generator):
...         if n == limit: break
...         yield value
```

This generator function uses another generator as a source of items. The idea is that we'll use the `arrival1()` function. The `samples()` function enumerates the items from a larger collection and stops when enough items have been seen. Since the `arrival1()` function is infinite, this boundary is essential.

Here's how we use these functions to simulate the arrival of customers. We'll produce a sequence of customer ID numbers:

```
>>> random.seed(1)
>>> list(samples(10, arrival1()))
[2, 1, 4, 1, 7, 7, 7, 6, 3, 1]
```

We forced the random number generator to have a specific seed value so that we would produce a known test sequence. We applied the `samples()` function to the `arrival1()` function to produce a sequence of 10 customer visits. Customer seven seemed to have a lot of repeat business. Customers zero and five never showed up at all.

This is just a simulation of data. A business would use sales receipts to determine customer visits. A web site might record visits in a database, or might scrape the web logs to determine the sequence of actual values.

What's the expected number of visits before we've seen all eight customers?

```
>>> from fractions import Fraction
>>> def expected(n=8):
...     return n * sum(Fraction(1,(i+1)) for i in range(n))
```

This function creates the series of fractions 1/1, 1/2, up to 1/*n*. These are summed and multiplied by *n*.

```
>>> expected(8)
Fraction(761, 35)
>>> round(float(expected(8)))
22
```

On an average, it will take 22 customer visits before we'll see all eight of our customers once.

How do we use the `set` collection to create statistics on the actual number of visits before we've seen all eight customers?

How to do it...

As we step through each customer visit, we'll put the customer ID into a `set` collection. Duplicates aren't saved in a set. Once a customer ID is a member of the set, adding the value again doesn't change the set. We'll summarize the steps in this recipe and then show the complete function:

1. Start with an empty `set` and a zero counter.
2. Begin a `for` loop to visit all of the data items.
3. Add the next item to the `set`. Add one to the counter.
4. If the `set` is complete, the count can be yielded. This is the number of customers required to see a complete set. After yielding, empty the `set` and initialize the counter to zero in preparation for the next customer.

Here's the function:

```
def coupon_collector(n, data):
    count, collection = 0, set()
    for item in data:
        count += 1
        collection.add(item)
        if len(collection) == n:
            yield count
            count, collection = 0, set()
```

This will start a `count` at zero and create an empty set, `collection`, in which we'll collect customer ID's. We'll step through each item in the sequence of source data values, `data`. The value of `count` shows how many visitors there are. The value of the variable `collection` is the set of distinct visitors.

The `add()` method of a `set` will mutate the set to add a distinct value. If the value is already in the set, there's no change to the content.

When the size of the collection is the size of our target population, we've got a complete set of coupons. We can yield the value of the `count`. We also reset the count of visits and create a new empty set for our collection of coupons.

How it works...

Since this is a generator, we'll need to capture the data by creating a `list` object from the results. Here's how we'd use the `coupon_collector()` function:

```
from statistics import mean
expected_time = float(expected(n))
data = samples(100, arrival1())
wait_times = list(coupon_collector(n, data))
average_time = mean(wait_times)
```

We've computed the expected time to see all `n` customers. We've used `samples(100, arrival1())` as a simulation to create the `data` variable which has a sequence of visits. In real life, we'd analyze sales receipts to gather this sequence of visits.

We applied the Coupon Collector's Test to the data. This emitted a sequence of values that showed how many customers had to arrive to create a complete set of *coupons* or customer ID's. This sequence of counts should be close to the expected number of visits. We've assigned this sequence to the variable `wait_times` because we've measured the time we need to wait before seeing all of the customers in our sample set.

This lets us easily compare the actual data with the expected data. The function that we just saw, `arrival1()`, produces averages that are quite close to the expected values. Since the input data is random, the simulation won't produce values that precisely match the expectation.

The Coupon Collector's Test relies on collecting a set of coupons. In this case, the term **set** is used in exact mathematical formalism that best represents the data.

A given item either is a member of a set or it is not. We can't add it to the set more than once. For example, we can create a set manually and add an item to it:

```
>>> collection = set()
>>> collection.add(1)
>>> collection
{1}
```

When we attempt to add this item again, the value of the `set` doesn't change.

```
>>> collection.add(1)
>>> collection
{1}
>>> 1 in collection
True
```

This is the perfect data representation for collecting coupons.

Note that the `add()` method does not return a value. It mutates the `set` object. 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.

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 is like an operator—it creates a new result `set`. It does not mutate either of the operand sets.
- We can use the `|` union operator to compute the union of two 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, we'll need to create a singleton `set` from the item we're going to add. Here's an example of adding a single item, `3`, to a set by turning it into a singleton set:

```
>>> collection
{1}
>>> item = 3
>>> collection.union( {item} )
{1, 3}
>>> collection
{1}
```

Here, 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 that is the union of `collection` and `{item}`.

Note that `union()` returns a resulting object and leaves the original `collection` set untouched. We would need to use this as `collection = collection.union({item})` to update the `collection` object.

This is yet another alternative that uses the union operator, `|`:

```
>>> collection = collection | {item}
>>> collection
{1, 3}
```

This parallels common mathematical notation for {1, 3} ∪ {3} ≡ {1, 3}.

We can also use the `update()` method:

```
>>> collection.update( {4} )
>>> collection
{1, 3, 4}
```

This method mutates the `set` object. Because it mutates the set, it does not return a value.

Python has a number of set operators. These are ordinary operator symbols that we can use in complex expressions:

- `|` for union, often typeset as $A \cup B$
- `&` for intersection, often typeset as $A \cap B$
- `^` for symmetric difference, often typeset as $A \Delta B$
- `-` for subtraction, often typeset as $A - B$

See also

- In the *Removing items from a set – remove, pop, and difference* recipe we'll look at how we can update a set by removing or replacing items

Removing items from a set – 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 an arbitrary item.

Additionally, we can compute a new set using set intersection, difference, and symmetric difference operators: `&`, `-`, and `^`. These will produce a new set which 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
...
...  - execute the ruby block print IP
... [2016-03-05T09:29:31-05:00] INFO: Chef Run complete in 23.233811181
seconds
...
... Running handlers:
... [2016-03-05T09:29:31-05:00] INFO: Running report handlers
... Running handlers complete
... [2016-03-05T09:29:31-05:00] INFO: Report handlers complete
... Chef Client finished, 2/2 resources updated in 29.233811181 seconds
... '''
```

We need to find the `IP: 111.222.111.222` lines in the log.

Here's how we'd do that:

```
>>> import re
>>> pattern = re.compile(r"IP: \d+\.\d+\.\d+\.\d+")
>>> matches = set( pattern.findall(log) )
>>> matches
{'IP: 111.222.111.222'}
```

The problem with the larger log file is that there are places where the target line has real information. These are mingled with lines that look similar, but are just examples. We'll also find lines like `IP: 1.2.3.4`, which is irrelevant output. It turns out that there are several of these irrelevant kinds of lines that we'd like to ignore.

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:

   ```
   >>> 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. Assume we have data that includes good addresses plus dummy and placeholder addresses from other parts of the log:

   ```
   >>> matches = {'IP: 111.222.111.222', 'IP: 1.2.3.4'}
   ```

3. Remove items from the set of matches using a form of 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'}
   ```

   Note that both of these are operators which return new sets as their results. Neither of these will mutate the underlying set objects.

We'll often use these in statements like this:

```
>>> 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.

In this case, if the item is not present in the set, it does not raise a `KeyError` exception.

How it works...

A `set` object only tracks membership. An item is either in the `set` or it's not in the `set`. We specify the item we want to remove. Removing an item doesn't depend on an index position or a key value.

Because we have `set` operators, we can remove any of the items in one `set` from a target `set`. We don't need to process the items individually.

There's more...

We have several ways to remove items from a set:

- In the 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.

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'}
```

First, we made a copy of the original `matches` set. This created a new set that we assigned the `valid_matches` set. We then applied the `difference_update()` method to remove the undesirable items from this set.

Since the set was mutated, no value is returned. Also, since the set is a copy, this doesn't modify the original `matches` set.

We could do something like this 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. This is one way to avoid the raising a `KeyError` exception. The alternative is to use a `try:` statement to silence the exception.

The `pop()` method removes an arbitrary item. It both mutates the set and returns the item which was removed. If we try to pop items from an empty set, we'll raise a `KeyError` exception.

See also

- In the *Using set methods and operators* recipe we'll look at other ways to create sets

Creating dictionaries – inserting and updating

A dictionary is one kind of Python mapping. The built-in type `dict` class provides a number of common features. There are some common variations on these features defined in the `collections` module.

As we noted in the *Choosing a data structure* recipe, we'll use a dictionary when we have some key that we need to map to a given value. For example, we might want to map a single word to a long, complex definition of the word. Or perhaps some value to a count of the number of times that value has occurred in a dataset.

The *key and count* dictionary is very common. We'll look at a detailed recipe that shows how to initialize the dictionary and update the counter.

In the *Using set methods and operators* recipe we looked at the arrival of customers at a business. In that recipe, we used a set to determine how many visits were required before the business had collected a complete set of visits.

Getting ready

In this recipe, we'll look at creating a histogram that shows how many times each customer visited. In order to create some interesting data, we'll modify the sample generator that was used in the other recipe.

In the earlier example, we used a simple, uniform random number generator to pick the sequence of customers. This is an alternative way to pick customers that generates random numbers with a slightly different distribution:

```
>>> def arrival2(n=8):
...     p = 0
...     while True:
...         step = random.choice([-1,0,+1])
...         p += step
...         yield abs(p) % n
```

This uses a technique called **random walk** to generate the next customer ID number. It will start with zero and then make one of three changes. It may use the same customer or one of the two adjacent customer numbers. Using the expression `abs(p) % n` allows us to compute any integer value and map the number, p, to the range $0 \le p < n$.

Here's a tool to generate some data that we can use to simulate the arrival of customers:

```
>>> import random
>>> from ch04_r06 import samples, arrival2
>>> random.seed(1)
>>> list( samples(10, arrival2(8)) )
[1, 0, 1, 1, 2, 2, 2, 2, 1, 1]
```

This shows us how the `arrival2()` function simulates customers who tend to cluster around the starting value of customer zero. If we use this for the Coupon Collector's Test in the *Using set methods and operators* recipe, we'll see that this generator creates sample data that fails that test spectacularly. The clumpy arrival times mean we have to see an extraordinary number of customers before we've collected all eight distinct customers.

A histogram counts the number of occurrences of each customer. We'll use a dictionary to map from customer ID to the number of times we've seen the customer.

How to do it...

1. Create an empty dictionary with `{}`. We can also use `dict()` to create an empty dictionary. Since we're going to create a histogram that counts the number of times each customer arrived, we'll call it `histogram`:

```
histogram = {}
```

2. For each customer number, if it's new, add an empty list to the dictionary. We can do this with an `if` statement or we can use the `setdefault()` method of the dictionary. We'll show the `if` statement version first. Later, we'll look at the `setdefault()` optimization.
3. Increment the value in the dictionary.

Here's the resulting loop to count occurrences in a dictionary. It works by creating and updating items:

```
for customer in source:
    if customer not in histogram:
        histogram[customer]= 0
    histogram[customer] += 1
```

When this is done, we'll have a count of the total number of simulated visits from each customer.

We can turn this into a handy bar chart to compare the frequencies. We can compute some basic descriptive statistics including the mean and standard deviation to see if any customer is over-represented or under-represented.

How it works...

The core feature of a dictionary is a mapping from an immutable value to an object of any kind. In this case, we've used an immutable number as the key, and another number as the value. As we count, we replace the value associated with the key.

It can seem a little unusual to write:

```
histogram[customer] += 1
```

Or to write:

```
histogram[customer] = histogram[customer] + 1
```

and think of the value in the dictionary as being *replaced*. When we write an expression like `histogram[customer] + 1` we're computing a new integer object from two other integer objects. This new object replaces the old value in the dictionary.

It's essential that dictionary key objects be immutable. We cannot use a `list`, `set`, or `dict` 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.

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. Our loop would look like this:

```
histogram = {}
for customer in source:
    histogram.setdefault(customer, 0)
    histogram[customer] += 1
```

If the key value, `customer`, doesn't exist, a default value is provided. If the key does exist, the `setdefault()` method does nothing.

The `collections` module provides a number of alternative mappings that we can use instead of the default `dict` mapping.

- `defaultdict`: This collection saves us from having to write step two explicitly. We provide an initialization function as part of creating a `defaultdict`. We'll look at an example soon.

- `OrderedDict`: This collection retains the keys in the order they were initially created. We'll save this for the *Controlling the order of dict keys* recipe.
- `Counter`: This collection does the entire **key-and-count** algorithm as it is being created. We'll look at this soon too.

Here's the version using the `defaultdict` class:

```
from collections import defaultdict
def summarize_3(source):
    histogram = defaultdict(int)
    for item in source:
        histogram[item] += 1
    return histogram
```

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` will evaluate the given function object to create default values.

This allows us to simply use `histogram[item] += 1`. If the value of the `item` attribute was previous in the dictionary, it will be incremented. If the value of the `item` attribute was not already in the dictionary, the `int` function is evaluated and that becomes the default value.

The other way we can do this is by creating a `Counter` object. We need to import the `Counter` class so that we can build the `Counter` object from the raw data.

```
>>> from collections import Counter
>>> def summarize_4(source):
...     histogram = Counter(source)
...     return histogram
```

When we create a `Counter` from a source of data, the class will scan the data and count the distinct occurrences. This class implements the entire recipe.

Here's how the result looks:

```
>>> import random
>>> from pprint import pprint
>>> random.seed(1)
>>> histogram = summarize_4(samples(1000, arrival2(8)))
>>> pprint(histogram)
Counter({1: 150, 0: 130, 2: 129, 4: 128, 5: 127, 6: 118, 3: 117, 7: 101})
```

Note that a `Counter` object displays the values in descending order of count value. An `OrderedDict` object will display the values in the order in which the keys were created. A `dict` maintains no order.

If we want to impose an order on the keys, we can use:

```
>>> for key in sorted(histogram):
...     print(key, histogram[key])
0 130
1 150
2 129
3 117
4 128
5 127
6 118
7 101
```

See also

- In the *Removing from dictionaries – the pop() method and the del statement* recipe we'll look at how dictionaries can be modified by removing items
- In the *Controlling the order of dict keys* recipe we'll look at how we can control the order of keys in a dictionary

Removing from dictionaries – the pop() method and the del statement

A common use case for a dictionary is as an **associative store**: we can keep 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 value) from the dictionary

We have two common variations on this theme:

- We have the in-memory dictionary, `dict`, and the variations on this theme in the `collections` module. The collection only exists while our program is running.
- We also have persistent storage in the `shelve` and `dbm` modules. The data collection is a persistent file in the file system.

These are very similar, the distinctions between a `shelf.Shelf` and `dict` object are minor. This allows us to experiment with a `dict` and switch to a `Shelf` without making dramatic changes to a program.

A server process will often have multiple, concurrent sessions. When sessions are created, they can be placed into `dict` or `shelf`. When the session exits, the item can be deleted or perhaps archived.

We'll simulate this concept of a service that handles multiple requests. We'll define a service that works in a simulated environment with a single processing thread. We'll avoid concurrency and multi-processing considerations.

Getting ready

In the casino game of *Craps*, a player can (and often does) create and remove multiple bets during a game. The rules can be bafflingly complex, but the core concepts include four kinds of bets a player might make:

- A **pass line** bet: For our purposes, this is how one buys in at the start of a game.
- A **pass line odds** bet: This is not marked on the playing surface in a casino, but it's a real bet. This bet pays off at different odds than the pass line bet, and has some statistical advantages. It can be removed, also.
- A **come line** bet: This can be placed during a game.
- A **come line odds** bet: This, too, is placed during a game. This can be taken down, also.

The best way to understand all of these betting choices is to simulate the game and a player. The game will need to track all of the bets a player places. This can be done using a dictionary where bets are inserted, and removed when they pay off, the player takes them down, or the game ends.

We'll simplify parts of the simulation so that we can focus on using a dictionary properly. This is handled best as a class definition so that we can properly isolate bets and game rules from player rules. For more information on class design, see `Chapter 6`, *Basics of Classes and Objects*.

How to do it...

1. Create the overall dictionary object:

```
working_bets = {}
```

2. Define the key and value for each object we're inserting into the dictionary. For example, the key might be a description of the bet: `come`, `pass`, `come odds`, or `pass odds`. The value might be the amount of the bet. It's common to avoid working in currency, and instead work in units of the table minimum bet. Usually these are simple integer multiples, most often just the integer value one to represent the minimum bet.

3. Enter values as the bets are being placed:

```
working_bets[bet_name] = bet_amount
```

   For a concrete example, we'd have `working_bets["pass"] = 1`.

4. Remove values as bets are paid off or taken down. We can use the `del` statement or the dictionary `pop()` method:

```
del working_bets['come odds']
```

   If the key is not present, this will raise a `KeyError` exception.

The `pop()` method both mutates the dictionary and returns a value associated with the key. If the key doesn't exist, this will raise an `KeyError` exception.

```
amount = working_bets.pop('come odds')
```

It turns out that `pop()` can be given a default value. If the key is not present, it will not raise an exception, but will return the default value instead.

How it works...

Because a dictionary is a mutable object, we can remove keys from a dictionary. This will delete both the key and the value object associated with the key.

If we try to delete a key which does not exist, we'll raise a `KeyError` exception.

We can replace an object in a dictionary with statements like this:

```
working_bets["come"] = 1
working_bets["come"] = None
```

The key—`come`—remains in the dictionary. The old value, `1`, is no longer required and will be replaced by the new value, `None`. This is not the same as deleting an item.

There's more...

We can only remove the keys of a dictionary. As we noted earlier, we can set the value to `None` to remove the value, leaving the key in the dictionary.

When we use a dictionary in a `for` statement, the target variable will be assigned key values. For example:

```
for bet_name in working_bets:
    print(bet_name, working_bets[bet_name])
```

This will print all of the key values, `bet_name`, and the bet amount associated with that bet in the `working_bets` dictionary.

See also

- In the *Creating dictionaries – inserting and updating* recipe we'll look at how we create dictionaries and fill them with keys and values
- In the *Controlling the order of dict keys* recipe we'll look at how we can control the order of keys in a dictionary

Controlling the order of dict keys

In the *Creating dictionaries – inserting and updating* recipe we looked at the basics of creating a dictionary object. In many cases, we'll put items into a dictionary and fetch items from a dictionary individually. The idea of an order to the keys doesn't even enter into the problem.

There are some cases where we might want to display the contents of a dictionary. In this case, we often want to impose some order on the keys. For example, when we work with web services, the messages are often dictionaries encoded in JSON notation. In many cases we'd like to keep the keys in a particular order so that the message is easier to understand when it's displayed in a debugging log.

As another example, when we read data with the `csv` module each row from a spreadsheet can be represented as a dictionary. In this case, we almost always want to keep the keys in a given order so that the dictionary follows the structure of the source file.

Getting ready

A dictionary is a good model for a row from a spreadsheet. This works particularly well when the spreadsheet has a heading row with column titles. Let's say we have some data collected in a spreadsheet that looks like this:

final	**least**	**most**
5	0	6
-3	-4	0
-1	-3	1
3	0	4

This shows the final outcome, the lowest amount the player had, and the highest amount the player had. We can use the `csv` module to read this data for further analysis:

```
>>> from pathlib import Path
>>> import csv
>>> data_path = Path('code/craps.csv')
>>> with data_path.open() as data_file:
...     reader = csv.DictReader(data_file)
...     data = list(reader)
>>> for row in data:
...    print(row)
{'most': '6', 'least': '0', 'final': '5'}
{'most': '0', 'least': '-4', 'final': '-3'}
{'most': '1', 'least': '-3', 'final': '-1'}
```

```
{'most': '4', 'least': '0', 'final': '3'}
```

Each row of the spreadsheet is a dictionary. However, there's something peculiar about each row. It's not obvious, but the order of the keys in the row doesn't match the order of the keys in the original `.csv` file.

Why is that? The default `dict` structure does not guarantee any ordering for the keys. What if we want to show the keys in a specific order?

How to do it...

We have two common ways to force an ordering on the keys of a dictionary:

- Create an `OrderedDict`: This keeps keys in the order they are created
- Use `sorted()` on the keys: This puts the keys into a sorted order

Most of the time, we can simply use `OrderedDict` instead of `dict()` or `{}` to create an empty dictionary. This will allow us to create keys in the required order.

Sometimes, however, we can't easily replace a `dict` instance with an `OrderedDict` instance. We've chosen this example because we can't trivially replace the `dict` class that is created by `csv`.

Here's how we can force the row's `dict` keys to follow the order of the columns in the original `.csv` file:

1. Get the preferred order of keys. In the case of a `DictReader` the `fieldnames` attribute of the reader object has the proper order information.
2. Use a generator expression to create the fields in the proper order. We'll have something like this:

```
((name, raw_row[name]) for name in reader.fieldnames)
```

3. Create an `OrderedDict` from the generator. Here's the whole sequence:

```
>>> from collections import OrderedDict
>>> with data_path.open() as data_file:
...     reader = csv.DictReader(data_file)
...     for raw_row in reader:
...         column_sequence = ((name, raw_row[name])
...             for name in reader.fieldnames)
...         good_row = OrderedDict(column_sequence)
...         print(good_row)
```

```
OrderedDict([('final', '5'), ('least', '0'), ('most', '6')])
OrderedDict([('final', '-3'), ('least', '-4'), ('most', '0')])
OrderedDict([('final', '-1'), ('least', '-3'), ('most', '1')])
OrderedDict([('final', '3'), ('least', '0'), ('most', '4')])
```

This builds dictionaries with keys in a specific order.

As an optimization, we can combine the two steps into a single step:

```
OrderedDict((name, raw_row[name]) for name in reader.fieldnames)
```

This will build an ordered version of the `raw_row` object.

How it works...

The `OrderedDict` class keeps the keys in the order they are created. This class is very handy for assuring a structure remains in an order that's easier to understand.

There's a small performance cost to this, of course. The default `dict` class computes a hash for each key, and the hash values are used to locate a space in the dictionary. This tends to use more memory, but performs extremely quickly.

The `OrderedDict` uses some additional storage to retain the ordering for the keys. This requires some additional time when a key is created. If key creation tends to dominate the algorithm, we'll notice the slowdown. If key retrieval tends to dominate the design, then we won't see much change when using an `OrderedDict`.

There's more...

In some packages—like `pymongo`—there are some alternative ordered dictionary implementations.

See `https://api.mongodb.org/python/current/api/bson/son.html`.

The `bson.son` module includes the `SON` class which is a very handy ordered dictionary. This is focused on the needs of the Mongo database, but it works very nicely for other applications, also.

See also

- In the *Creating dictionaries – inserting and updating* recipe we'll look at how we can create dictionaries.
- In the *Removing from dictionaries – the pop() method and the del statement* recipe we'll look at how dictionaries can be modified by removing items.

Handling dictionaries and sets in doctest examples

We will look at one small aspect of writing a proper test in this recipe. We'll look at testing overall in `Chapter 9`, *Testing*. The data structures in this chapter—`dict` and `set`—both include some complexity when it comes to writing proper tests.

Since `dict` keys (and `set` members) have no order, our test results will have a problem. We need to have a repeatable result, but there's no way to guarantee the order of the collection. This can lead to test results which don't properly match our expectations.

Assume that our test expects the set `{"Poe", "E", "Near", "A", "Raven"}`. Since there's no defined order to a set, Python can display this set in any order:

```
>>> {"Poe", "E", "Near", "A", "Raven"}
{'E', 'Poe', 'Raven', 'Near', 'A'}
```

The elements are the same, but the overall line of output from Python isn't the same. The `doctest` package relies on the literal output from the example being *identical* to the output produced by Python's REPL.

How can we be sure our doctest examples really work?

Getting ready

Let's look at an example that involves a `set` object:

```
>>> words = set(
... '''Beautiful is better than ugly.
... Explicit is better than implicit.
... Simple is better than complex.
... Complex is better than complicated.
... '''.replace('.', ' ').split())
```

```
>>> words
{'complicated', 'Simple', 'ugly', 'implicit', 'Beautiful',
'complex', 'is', 'Explicit', 'better', 'Complex', 'than'}
```

This example is simple. The results, however, will often vary each time we process this example. Indeed, when working on secure algorithms, it's considered important to have the order vary. This is called the **hash randomization** problem—when the hashed values are predictable, it can become a security vulnerability.

When we use the `doctest` module, we need to have examples that are perfectly consistent. As we'll see in `Chapter 9`, *Testing*, the `doctest` module is clever about locating examples, but it's not a genius about assuring that actual results match expected results.

And the problem is—mostly—confined to sets and dictionaries. These are two collections where key ordering cannot be guaranteed because of hash randomization.

How to do it...

When we need to be sure that items in a set or dictionary have a particular order, we can convert the collection to a sorted sequence.

We have two choices:

- Convert a set to a sorted sequence
- Convert a dictionary to a sorted sequence of (key, value) two-tuples

Both of these recipes are similar. Here's what we need to do to force a set into a normalized structure:

```
>>> list(sorted(words))
['Beautiful', 'Complex', 'Explicit', 'Simple', 'better',
'complex', 'complicated', 'implicit', 'is', 'than', 'ugly']
```

For a dictionary, we'll often use this:

```
list(sorted(some_dictionary.items()))
```

This will extract each item in the dictionary as a `(key, value)` two-tuple. The tuples will be sorted into order by the key. The resulting sequence will be turned into a list so that it can be compared with the expected results.

How it works...

When confronted with a collection that fails to impose an order, we have to locate a collection with two properties:

- The same content
- Some kind of consistent order

Python's built-in structures are variations on three themes:

- Sequence
- Set
- Mapping

Since the only one with a guaranteed order is the sequence, we can convert sets and mappings into sequences. This, it turns out, is easy to do with the `sorted()` function.

For sets, we'll sort the items. For mappings, we'll sort the `(key, value)` two-tuples. This assures us that the output from our example is precisely what is required.

There's more...

We'll look at several other kinds of data that has minor variations in `Chapter 9`, *Testing*:

- Floating-point numbers
- Dates
- Object ID's and Tracebacks
- Random sequences

All of these need to be put into a context with a predictable output so that tests will work repeatedly. The two data structures, `set` and `dict`, are the subjects of this chapter. We'll cover other variations in the relevant chapters.

Understanding variables, references, and assignment

How do variables really work? What happens when we assign a mutable object to two variables? We can easily have two variables that share references to a common object; this can lead to potentially confusing results when the shared object is mutable. The rules are simple and the consequences are generally obvious.

We'll focus on this rule: **Python shares references. It doesn't copy data**.

We need to look at what this rule on reference sharing means.

We'll create two data structures, one is mutable and one is immutable. We'll use two kinds of sequences, although we could do something similar with two kinds of sets: Getting ready We'll create two data structures, one is mutable and one is immutable. We'll use 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)
```

The mutable data structure can be changed and shared. The immutable data structure is also shared, but it's much harder to tell that it's being shared.

We can't easily do this with a mapping because Python doesn't offer a handy immutable mapping.

How to do it...

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)`.

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
```

2. Make a change to one of the two references to the collection. For mutable structures, we have methods like `append()` or `add()`:

```
>>> mutable += [mutable[-2] + mutable[-1]]
```

For a list structure, the += assignment is really an internal use of the `extend()` method.

We can do a similar thing with an immutable structure:

```
>>> immutable += (immutable[-2] + immutable[-1],)
```

Since a tuple has no method like `extend()`, the += will build a new tuple object and replace the value of `immutable` with that new object.

3. Look at the other reference to the structure:

```
>>> mutable_b
[1, 1, 2, 3, 5, 8, 13]
>>> mutable is mutable_b
True
>>> immutable_b
(5, 8, 13, 21)
>>> immutable
(5, 8, 13, 21, 34)
```

The two variables `mutable` and `mutable_b` 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.

How it works...

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 the object temporarily.

A variable is a reference to the underlying object. When we assign an object to a variable, we're giving a name to a reference to the underlying object. When we use a variable in an expression, Python locates the object to which the variable refers.

For mutable objects, a method of an object can modify the object's state. All variables that refer to the object will reflect the state change because a variable is just a reference, not a complete copy.

When we use a variable on an assignment statement there are two possible actions:

- For mutable objects that provide definitions for appropriate 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 which previously referred to the object being replaced are not affected, they continue to refer to old objects.

Python tracks the number of places that an object is referenced. When the number of references becomes zero, the object is no longer used anywhere, and can be removed from memory.

There's more...

Languages like C++ or Java have primitive types in addition to objects. In these languages, a += statement leverages a feature of the hardware instructions or the Java Virtual Machine to tweak the value of a primitive type.

Python doesn't have this kind of optimization. Numbers are immutable objects. When we do something like this:

```
>>> a = 355
>>> a += 113
```

We're not tweaking the internal state of the object `355`. This does not rely on the internal `__iadd__` special method. This behaves as if we had written:

```
>>> a = a + 113
```

The expression `a + 113` is evaluated, and 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.

See also

- In the *Making shallow and deep copies of objects* recipe we'll look at ways we can copy mutable structures

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. When we write:

```
a = b
```

we now have two references to the same underlying object. If `b` is a list, both `a` and `b` are references to the same, mutable list.

As we saw in the *Understanding variables, references, and assignment* recipe, a change to the `a` variable changes the list object that both `a` and `b` refer to.

Most of the time, this is the behavior we want. 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

We have to make special arrangements to make a copy of an object. We've seen several kinds of syntax for doing that.

- **Sequences** – `list` and `tuple`: We can use `sequence[:]` to copy a sequence by using an empty slice expression. We can also use `sequence.copy()` to make a copy of a variable named `sequence`.
- **Mappings** – `dict`: We can use `mapping.copy()` to copy a dictionary named `mapping`.
- **Sets** – `set` and `frozenset`: We can use `someset.copy()` to clone a set named `someset`.

What's important is that these are all *shallow* copies.

Shallow means that two collections will contain references to the same underlying objects. If the underlying objects are immutable numbers or strings, this distinction doesn't matter. When we can't mutate items inside the collection, the items are simply replaced.

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()
```

We have to make a shallow copy of the dictionary. The two copies 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 `some_dict`.

```
>>> 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:

```
>>> some_dict['a'].append(5)
>>> another_dict
{'a': [1, 1, 2, 3, 5]}
```

We made a change to a mutable `list` object that's shared between two `dict` objects, `some_dict` and `another_dict`.

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 effect works for `list` collections that contain other `list` objects as items, also:

```
>>> some_list = [[2, 3, 5], [7, 11, 13]]
>>> another_list = some_list.copy()
>>> some_list is another_list
False
>>> some_list[0] is another_list[0]
True
```

We've made a copy of an object, `some_list`, and assigned it to the variable `another_list`. The top-level `list` object is distinct, but the items within the `list` are shared references. We used the `is` operator to show that item zero in each list are both references to the same underlying objects.

Because we can't make a `set` of mutable objects, we don't really have to consider making shallow copies of sets which share items.

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 only 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` library:

```
>>> 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 too see that they're distinct objects.

How it works...

Making a shallow copy is relatively easy. We can write our own version of the algorithm using 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 mutable collection.

For a `list` the conceptual algorithm is something like this:

```
immutable = (numbers.Number, tuple, str, bytes)
def deepcopy_list(some_list:
    copy = []
    for item in some_list:
        if isinstance(item, immutable):
```

```
            copy.append(item)
        else:
            copy.append(deepcopy(item))
```

The actual code doesn't look like this, of course. It's a bit more clever in the way it handles each distinct Python type. This does, however, provide some hints as to how the `deepcopy()` function works.

It turns out that there are some additional considerations. The most import consideration is an object which contains a reference to itself.

We could do this:

```
a = [1, 2, 3]
a.append(a)
```

This is a confusing, but technically valid, Python construct. It will lead to problems when attempting to write a naïve recursive operation to visit all items in the list. In order to overcome this, an internal cache is used so that items are only copied once. After that, an internal reference can be found in the cache.

See also

- In the *Understanding variables, references, and assignment* recipe we'll look at how Python prefers to create references to objects.

Avoiding mutable default values for function parameters

In `Chapter 3`, *Function Definitions*, 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, it could look like this:

```
>>> from collections import Counter
>>> from random import randint, seed
>>> def gather_stats(n, samples=1000, summary=Counter()):
...     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 with two stories. The first story offers no argument collection. The function creates and returns a collection of statistics. Here's the example of this story:

```
>>> seed(1)
>>> s1 = gather_stats(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})
```

The second story allows us to provide an explicit parameter value so that the statistics update a given object. Here's an example of this story:

```
>>> seed(1)
>>> mc = Counter()
>>> gather_stats(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. This makes it easy to confirm that the results are the same if we provide a `Counter` object or use the default `Counter` object. In the second example, we provided an explicit `Counter` object, named `mc` to the function.

The `gather_stats()` function returns a value. When writing a script, we'd simply ignore the returned value. When working Python's interactive REPL the output is printed. We've shown `Counter...` instead of the lengthy output.

The problem arises when we do the following operation after doing the preceding two operations:

```
>>> seed(1)
>>> s3 = gather_stats(2)
>>> s3
Counter({7: 336, 6: 294, 8: 272, 9: 228, 5: 220, 10: 154, 11: 142, 4: 140,
3: 104, 12: 58, 2: 52})
```

Note that the counts are doubled. Something has gone wrong. Since this only happens when we use the default story more than once, it may pass a 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 following:

```
>>> s1 is s3
True
```

This means that two variables, `s1` and `s2`, are both references to the same underlying object. It appears that we've updated some shared collection.

Does that mean the value of `s1` changed?

```
>>> s1
Counter({7: 336, 6: 294, 8: 272, 9: 228, 5: 220, 10: 154, 11: 142, 4: 140,
3: 104, 12: 58, 2: 52})
```

Yes, the default use of this `gather_stats()` function seems to be sharing a single object. How can we avoid this?

How to do it...

There are two approaches to solving this problem:

- 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 the 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(n, samples=1000, summary=None):
```

2. Add an `if` statement to check for an argument value of `None` and replace it with a fresh, new mutable object:

```
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.

There are very few good reasons for providing a mutable object as a default value to a function. In most cases, we should consider changing the design, and not using a mutable object as a default value for a parameter. In the rare case where we really do have a complex algorithm which can update an object or create a fresh new object, we should consider defining two separate functions.

We'd refactor this function to look like this:

```
def create_stats(n, samples=1000):
    return update_stats(n, samples, Counter())
def update_stats(n, samples=1000, summary):
    summary.update(
        sum(randint(1,6) for d in range(n))
            for _ in range(samples))
```

We've created two separate functions. This will separate the two stories so that there's no confusion. The idea of optional mutable arguments is not a good idea in the first place.

How it works...

As we noted earlier, Python prefers to share references. It rarely creates copies of objects. Therefore, default values for function parameter values will be shared objects. Python doesn't easily create fresh, new objects.

The rule is very important and often confuses programmers new to Python.

Don't use mutable defaults for functions. A mutable object (`set`, `list`, `dict`) should not be a default value for a function parameter.

This rule applies to the core language. It doesn't apply throughout the standard library, however. There are cases where there are some clever alternative approaches.

There's more...

In the standard library, there are some examples of a cool technique that shows how we can create fresh default objects. One widely-used example is in the `defaultdict` collection. When we create a `defaultdict` we provide a no-argument function that will be used to create new dictionary entries.

When a key is missing from the dictionary, the given function is evaluated to compute a fresh default value. In the case of `defaultdict(int)` we're using the `int()` function to create an immutable object. As we've seen, a default value of an immutable object doesn't cause any problems because the immutable object has no internal state.

When we do `defaultdict(list)` or `defaultdict(set)` we see the real power of this design pattern. When a key is missing, a fresh, empty `list` (or `set`) is created.

The evaluate-a-function pattern used by `defaultdict` does not apply to the way functions themselves operate. Most of the time the default values we provide for function parameters are immutable objects like numbers, strings, or tuples. Having to wrap an immutable object with a `lambda` is certainly possible, but irksome because it's such a common case.

In order to leverage this technique, we need to modify the design of our example function. We will no longer update an existing counter object in the function. We'll always create a fresh, new object. We can modify what class of object is created.

Here's a function that allows us to plug in a different class in the case where we don't want the default `Counter` class to be used.

```
>>> def gather_stats(n, samples=1000, summary_func=lambda x:Counter(x)):
...     summary = summary_func(
...         sum(randint(1,6) for d in range(n))
...             for _ in range(samples))
...     return summary
```

For this version, we've defined an initialization value to be a function of one argument. The default will apply this one-argument function to a generator function for the random samples. We can override this function with another one-argument function that will collect data. This will build a fresh object using any kind of object that can gather data.

Here's an example using `list()`:

```
>>> seed(1)
>>> gather_stats(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 `Counter` object:

```
>>> seed(1)
>>> gather_stats(2, 12)
Counter({5: 2, 7: 2, 8: 2, 10: 2, 3: 1, 4: 1, 6: 1, 9: 1})
```

In this case, we've used the default value. The function created a `Counter()` object from the random samples.

See also

- See the *Creating dictionaries – inserting and updating* recipe, which shows how `defaultdict` works

5
User Inputs and Outputs

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

- Using features of the print() function
- Using input() and getpass() for user input
- Debugging with "format".format_map(vars())
- Using argparse to get command-line input
- Using cmd for creating command-line applications
- Using the OS environment settings

Introduction

The core value of software is to produce useful output. One simple type of output is a text display of some useful result. Python supports this with the `print()` function.

The `input()` function has a clear parallel with the `print()` function. The `input()` function reads text from a console, allowing us to provide distinct values to our programs.

There are a number of other common ways to provide input. Parsing the command-line is also 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 is distinct and needs to be looked at separately. In this chapter, we'll focus on the fundamentals of `input()` and `print()`.

Using features of the print() function

In many cases, the `print()` function is the first function we learn. The first script is often a variation on the following:

```
print("Hello world.")
```

We quickly learn that the `print()` function can display multiple values, including a helpful space between items.

When we write this:

```
>>> count = 9973
>>> print("Final count", count)
Final count 9973
```

We see that a space is included to separate the two values. 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?

It turns out that there are some more things we can do with `print()`.

Getting ready

We have a spreadsheet that is used to record fuel consumption on a large sailboat. It has rows that look like this:

Date	10/25/13	10/26/13	10/28/13
Engine on	08:24:00	09:12:00	13:21:00
Fuel height on	29	27	22
Engine off	13:15:00	18:25:00	06:25:00
Fuel height off	27	22	14

For more information on this data, refer to the *Removing items from a set – remove(), pop(), and difference* and *Slicing and dicing a list* recipes in `Chapter 4`, *Built-in Data Structures – list, set, dict*. There's no level gauge inside the tank. The depth of fuel has to be read through a sight glass on the side of the tank, which is why the volume of fuel is stated as a depth. The full depth of the tank is about 31 inches, and the volume is about 72 gallons; it's possible to convert depth to volume.

Here's an example of using the CSV data. This function reads the file and returns a list of fields built from each row:

```
>>> from pathlib import Path
>>> import csv
>>> from collections import OrderedDict
>>> def get_fuel_use(source_path):
...     with source_path.open() as source_file:
...         rdr= csv.DictReader(source_file)
...         od = (OrderedDict(
...             [(column, row[column]) for column in rdr.fieldnames])
...             for row in rdr)
...         data = list(od)
...     return data
>>> source_path = Path("code/fuel2.csv")
>>> fuel_use= get_fuel_use(source_path)
>>> fuel_use
[OrderedDict([('date', '10/25/13'), ('engine on', '08:24:00'),
    ('fuel height on', '29'), ('engine off', '13:15:00'),
    ('fuel height off', '27')]),
OrderedDict([('date', '10/26/13'), ('engine on', '09:12:00'),
    ('fuel height on', '27'), ('engine off', '18:25:00'),
    ('fuel height off', '22')]),
OrderedDict([('date', '10/28/13'), ('engine on', '13:21:00'),
    ('fuel height on', '22'), ('engine off', '06:25:00'),
    ('fuel height off', '14')])]
```

We used a `pathlib.Path` object to define the location of the raw data. We defined a function, `get_fuel_use()`, that will open and read the file at a given path. This function creates a list of rows from the source spreadsheet. Each line of data is represented as an `OrderedDict` object.

The function starts by creating a `csv.DictReader` object to parse the raw data. The reader normally returns a built-in `dict` object, which doesn't force a particular ordering on the keys. To force a particular key order, this function uses a generator expression to create an `OrderedDict` object for each row. The `fieldnames` attribute of the reader, `rdr`, is used to force the columns into a specific order. The generator expression uses a nested pair of loops: one loop processes each field of a row and the outer loop processes each row of the data.

The result is a list object that contains the `OrderedDict` objects. This is a consistent source of data that we can use for printing. Each row has five fields based on the column names in the first row.

How to do it...

We have two ways to control the `print()` formatting:

- Set the inter-field separator character, `sep`, which has a space as its default value
- Set the end-of-line character, `end`, which has the `\n` character as its default value

We'll show several examples of changing `sep` and `end`. Each is a kind of one-step recipe.

The default case looks like this. This example has no change to `sep` or `end`:

```
>>> for leg in fuel_use:
...     start = float(leg['fuel height on'])
...     finish = float(leg['fuel height off'])
...     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. The `\n` character at the end of each collection of data items means that each `print()` function produces a separate line.

When preparing data, we might want to use a format that's similar to comma-separated values, perhaps using a column separator that's not a simple comma. Here's an example using `|`:

```
>>> print("date", "start", "end", "depth", sep=" | ")
date | start | end | depth
>>> for leg in fuel_use:
...     start = float(leg['fuel height on'])
...     finish = float(leg['fuel height off'])
...     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.

The most common case seems to be where we want to suppress the separators entirely. This gives us a fine degree of control over the output.

Here's how we might change the default punctuation to emphasize the field name and value. In this case, we've changed the `end` setting:

```
>>> for leg in fuel_use:
...     start = float(leg['fuel height on'])
...     finish = float(leg['fuel height off'])
...     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 end string was changed to `, `, each use of the `print()` function did not produce a separate line. We didn't get a proper end of line until the final `print()` function, which had the default value for `end`.

Clearly, this technique can get quite complex for anything more sophisticated than these simple examples. For something simple, we can tweak the separator or ending. For anything more complex, we need to use the `format()` method of a string.

How it works...

In the general case, the `print()` function is a handy wrapper around `stdout.write()`. This relationship can be changed, as we'll see next.

We can imagine that `print()` has a definition something like this:

```
def print(*args, *, sep=None, end=None, file=sys.stdout):
    if sep is None: sep = ' '
    if end is None: end = '\n'
    arg_iter= iter(args)
    first = next(arg_iter)
    sys.stdout.write(repr(first))
    for value in arg_iter:
        sys.stdout.write(sep)
        sys.stdout.write(repr(value()))
    sys.stdout.write(end)
```

This provides a hint as to how the the separator string and end string are included in the output from the `print()` function. If no value is provided, the default values are space and new line. The function iterates through the argument values, treating the first value as special because it does not have a separator. This approach assures that the separator string, `sep`, appears between values.

The end of line string, `end`, appears after all of the values. It is always written. We can effectively turn it off by setting it to a zero-length string.

There's more...

The `sys` module defines the two standard output files that are always available: `sys.stdout` and `sys.stderr`.

We can use the `file=` keyword argument to write to the standard error file in addition 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 would not be part of the standard output stream.

Generally, we need to be cautious of opening too many output files in a single program. The OS limits are usually more than adequate to open many files. However, it can become confusing when a program creates a large number of files.

It often works out nicely to use OS file redirection techniques. A program's primary output can be written to `sys.stdout`; this is easily redirected at the OS level. A user might enter a command line like this:

```
python3 myapp.py <input.dat >output.dat
```

This will provide the `input.dat` file as the input on `sys.stdin`. When the Python program writes to `sys.stdout`, the output will be redirected by the OS to the `output.dat` object.

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("somefile.dat")
with target_path.open('w', encoding='utf-8') as target_file:
    print("Some output", file=target_file)
    print("Ordinary log")
```

In this example, we've opened a specific path for 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 and 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

- Refer to the *Debugging with "format".format_map(vars())* recipe
- For more information on the input data in this example, refer to the *Removing items from a set – remove(), pop(), and difference* and *Slicing and dicing a list* recipes in `Chapter 4`, *Built-in Data Structures – list, set, dict*
- For more information on file operations in general, refer to `Chapter 8`, *Input/Output, Physical Format, Logical Layout*

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 the user for input.

There are two relatively common situations:

- **Ordinary input**: We use the `input()` function for this. This will provide a helpful echo of the characters being entered.
- **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.

The `input()` and `getpass()` functions are just two implementation choices for reading from the console. It turns out that getting the string of characters is only the first step in processing. We actually have separate tiers of considerations:

1. The initial interaction with the console. This is the basics of writing a prompt and reading input. This must correctly handle data as well as keyboard events, such as backspace for editing. This may also mean handling end-of-file appropriately.
2. Validating the input to see that it belongs in the expected domain of values. We might be looking for digits, yes/no values, or days of the week. In most cases, there are two parts to the validation tier:
    - We check whether the input fits some general domain, for example, numbers.
    - We check whether the input fits some more specific subdomain. For example, this might include a check to see if the number is greater than or equal to zero.
3. Validating the input in some larger context to ensure that it's consistent with other inputs. For example, we can check whether the user's birth date is prior to today.

Above and beyond these techniques, we'll look at some other approaches in the *Using argparse to get command-line input* recipe.

Getting ready

We'll look at a technique for reading a complex structure from a person. In this case, we'll use the year, month, and day as separate items to create a complete date.

Here's a quick example that omits all of the validation issues:

```
from datetime import date

def get_date():
    year = int(input("year: "))
    month = int(input("month [1-12]: "))
    day = int(input("day [1-31]: "))
    result = date(year, month, day)
    return result
```

This illustrates how easy it is to use the `input()` function. We often need to wrap this in additional processing to make it more useful. The calendar is complex, and we'd hate to accept February 32 without warning a user that this is not a proper date.

How to do it...

1. Check whether the input is a password or something equally subject to redaction. If so, then use the `getpass.getpass()` function. This means we need to import the following function:

```
from getpass import getpass
```

   Otherwise, if the redacted input is not required, use the `input()` function.

2. Determine which prompt will be used. This might be as simple as `>>>` or something more complex. In some cases, we might provide a great deal of contextual information. In our example, we provided a field name and a hint about the type of data expected as a prompt string. The prompt string is the argument to the `input()` or `getpass()` function:

```
year = int(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.
4. We might want to restructure our input to look like this:

```
month = None
while month is None:
    month_text = input("month [1-12]: ")
    try:
        month = int(month_text)
        if 1 <= month <= 12:
            pass
        else:
            raise ValueError("Month of range 1-12")
    except ValueError as ex:
        print(ex)
        month = None
```

This applies two validation rules to the input:

- It checks whether the month is a valid integer using the `int()` function
- It checks whether the integer is in the range [1, 12] using an `if` statement that raises a `ValueError` exception

Raising an exception for faulty input is generally the simplest approach. It allows us the most flexibility. There are other exception classes we might use, including defining a customized data validation exception.

Since we'll use nearly identical loops for each field of a complex object, we need to restructure this input and validate the sequence into a separate function. We'll call it `get_integer()`. We'll look at the details here:

1. Validate the composite object. In this case, it also means that our overall input needs to be restructured to allow for a retry in the event of bad input:

```
input_date = None
while input_date is None:
    year = get_integer("year: ", 1900, 2100)
    month = get_integer("month [1-12]: ", 1, 12)
    day = get_integer("day [1-31]: ", 1, 31)
    try:
        result = date(year, month, day)
    except ValueError as ex:
        print(ex)
        input_date = None
# assert input_date is the valid date entered by the user
```

   This overall loop implements the higher level validation of the composite date object.

   Given a year and a month, we can actually determine a slightly more narrow range for the number of days. The complexity is that not only do months have different numbers of days, varying from 28 to 31, but February has a number of days that varies with the type of year.

2. Rather than mimicing the rules, it's easier to use the `datetime` module to compute the first days of two adjacent months, as follows:

```
day_1_date = date(year, month, 1)
if month == 12:
    next_year, next_month = year+1, 1
else:
    next_year, next_month = year, month+1
day_end_date = date(next_year, next_month, 1)
```

This will properly compute the last day of any given month. The algorithm works by computing the first day of a given year and month. It then computes the first day of the next month. It properly changes the year so that January of `year+1` follows December of `year`.

The number of days between these dates is the number of days in the given month. We can use the expression `(day_end_date - day_1_date).days` to extract the number of days from the `timedelta` object.

How it works...

We need to decompose the input problem into several separate, but closely related problems. 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 simply
- `getpass.getpass()`: This prompts and reads passwords without an echo

We expect to be able to edit the current line of input using the *Backspace* character. In some environments, there is a more sophisticated editor available. It's embodied in the Python `readline` module. This module, if present, can add a great deal of editing while preparing a line of input. The principle feature of this module is an OS-level input history—we can use the up arrow key to recover any previous input.

We've decomposed the input validation into several tiers to reflect the kind of programming required to confirm that the input is valid:

- A **general domain** validation should use the simple conversion functions such as `int()` or `float()`. These tend to raise exceptions for invalid data. It's far simpler to use these conversion functions and handle the exceptions than to attempt to write a regular expression that matches valid numeric values.
- Our **subdomain** validation must use an `if` statement to determine whether values fit any additional constraints, such as ranges, that are imposed. For consistency, this should also raise an exception if the data is invalid.

There are a lot of potential kinds of constraints that might be imposed on values. For example, we might want only valid OS process IDs, called PIDs. This requires checking the `/proc/<pid>` path on Nanny Linux systems.

For BSD-based systems such as Mac OS X, the `/proc` file system doesn't exist. Instead, something like the following needs to be done to determine if a PID is valid:

```
import subprocess
status = subprocess.check_output(
    ['ps',PID])
```

For Windows, the command would look like this:

```
status = subprocess.check_output(
    ['tasklist', '/fi', '"PID eq {PID}"'.format(PID=PID)])
```

Either of these two functions would need to be part of input validation to ensure that the user is entering a proper PID value. This can only be applied if the primary domain of integers was assured.

Finally, our overall input function should also raise an exception for invalid input. This can vary quite a bit in complexity. We created a simple date object in the example. In other cases, we might have to do considerably more processing to determine whether a complex input is valid.

There's more...

We have several alternatives for user input that involve slightly different approaches. We'll look at these two topics in detail:

- Input string parsing: This will involve simple use of `input()` with clever parsing
- Interaction via `cmd` module: This involves a more complex class, and somewhat simpler parsing

Input string parsing

A simple date value requires three separate fields. A more complex date-time that includes a timezone offset from UTC will involve seven separate fields. The user experience might be improved by reading and parsing a string rather than individual fields.

For a simple date input, we might use the following:

```
raw_date_str = input("date [yyyy-mm-dd]: ")
input_date = datetime.strptime(raw_date_str, '%Y-%m-%d').date()
```

We've used the `strptime()` function to parse a time string in a given format. We've emphasized the expected date format in the prompt that's provided in the `input()` function.

This style of input requires the user to enter a more complex string. Since it's a single string that includes all of the details for a date, many people find it as easier and more friendly.

Note that both techniques—gathering individual fields and processing a complex string—depend on the underlying `input()` function.

Interaction via the cmd module

The `cmd` module includes the `Cmd` class that can be used to build an interactive interface. This takes a dramatically different approach to the notion of user interaction. It does not rely on using `input()` explicitly.

We'll look at this closely in the *Using cmd for creating command-line applications* recipe.

See also

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`

Specifically, all of these commands that begin with `ck` are for gathering and validating user input. This could be used to define a module of input validation rules:

- `ckdate`: This prompts for and validates a date
- `ckgid`: This prompts for and validates a group ID
- `ckint`: This displays a prompt, verifies, and returns an integer value
- `ckitem`: This builds a menu, prompts for, and returns a menu item
- `ckkeywd`: This prompts for and validates a keyword
- `ckpath`: This displays a prompt, verifies, and returns a pathname
- `ckrange`: This prompts for and validates an integer

- `ckstr`: This displays a prompt, verifies, and returns a string answer
- `cktime`: This displays a prompt, verifies, and returns a time of day
- `ckuid`: This prompts for and validates a user ID
- `ckyorn`: This prompts for and validates yes/no

Debugging with "format".format_map(vars())

One of the most important debugging and design tools available in Python is the `print()` function. There are some kinds of formatting options available; we looked at these in the *Using features of the print() function* recipe.

What if we want more flexible output? We have more flexibility with the `"string".format_map()` method. This isn't all. We can couple this with the `vars()` function to create something that often leads to a wow!

Getting ready

Let's look at a multistep 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 `mean_size`, `std_size`, and `sig1` variables all show elements of the final list comprehension that filters the `size` list. If the result is confusing or even incorrect, it's helpful to know the intermediate steps in the calculation. In this case, because they're floating-point values, we often want to round the results so that they're more meaningful.

How to do it...

1. The `vars()` function builds a dictionary structure from a variety of sources.
2. If no arguments are given, then by default, the `vars()` function will expand all the local variables. This creates a mapping that can be used with the `format_map()` method of a template string.
3. Using a mapping allows us to inject variables using the variable's name into the format template. It looks as follows:

```
>>> print(
...     "mean={mean_size:.2f}, std={std_size:.2f}"
...     .format_map(vars())
... )
mean=1414.77, std=901.10
```

We can put any local variable into the format string. Using `format_map(vars())`, we don't need to have a more complex way to pick which variables are going to be displayed.

How it works...

The `vars()` function builds a dictionary structure from a variety of sources:

- The `vars()` expression will expand all local variables to create a mapping that can be used with the `format_map()` method.
- The `vars(object)` expression will expand all of the items in an object's internal `__dict__` attribute. This allows us to expose attributes of class definitions and objects. When we look at objects in `Chapter 6`, *Basics of Classes and Objects*, we'll see how we can leverage this.

The `format_map()` method expects a single argument, which is a mapping. The format string uses `{name}` to refer to keys in the mapping. We can use `{name:format}` to provide a format specification. We can also use `{name!conversion}` to provide a conversion function using the `repr()`, `str()`, or `ascii()` functions.

For more background on the formatting options, refer to the *Building complex strings with "template".format()* recipe in `Chapter 1`, *Numbers, Strings, and Tuples*.

There's more...

The `format_map(vars())` technique is a simple way to display the values of variables. An alternative is to use `format(**vars())`. This alternative can give us some additional flexibility.

For example, we can use this more flexible format to include additional calculations that aren't simply local variables:

```
>>> print(
...     "mean={mean_size:.2f}, std={std_size:.2f},"
...     " limit2={sig2:.2f}"
...     .format(sig2=mean_size+2*std_size, **vars())
... )
mean=1414.77, std=901.10, limit2=3216.97
```

We've computed a new value, `sig2`, that appears only inside the formatted output.

See also

- Refer to the *Building complex strings with "template".format()* recipe in `Chapter 1`, *Numbers, Strings, and Tuples*, for more of the things that can be done with the format() method
- Refer to the *Using features of the print() function* recipe for other formatting options

Using argparse to get command-line input

In some cases, we want to get the user input from the OS command line without a lot of 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:

```
slott$ python3 ch05_r04.py -r KM 36.12,-86.67 33.94,-118.40
From (36.12, -86.67) to (33.94, -118.4) in KM = 2887.35
```

The OS prompt is `slott$`. We entered a command of `python3 ch05_r04.py`. This command had an optional argument, `-r KM`, and two positional arguments of `36.12,-86.67` and `33.94,-118.40`.

The program parses the command-line arguments and writes the result back to the console. This allows for a very simple kind of user interaction. It keeps the program very simple. It allows the user to write a shell script to invoke the program or merge the program with other Python programs to create a higher level program.

If the user enters something incorrect, the interaction might look like this:

```
slott$ python3 ch05_r04.py -r KM 36.12,-86.67 33.94,-118asd
usage: ch05_r04.py [-h] [-r {NM,MI,KM}] p1 p2
ch05_r04.py: error: argument p2: could not convert string to float:
'-118asd'
```

An invalid argument value of `-118asd` leads to an error message. The program stopped with an error status code. For the most part, the user can hit the up arrow key to get the previous command line back, make a change, and run the program again. The interaction is delegated to the OS command line.

The name of the program—`ch05_r04`—isn't too informative. We could perhaps do better. The positional arguments are two (latitude, longitude) pairs. The output shows the distance between the two in the given units.

How do we parse argument values from the command line?

Getting ready

The first thing we need to do is to refactor our code to create two separate functions:

- A function to get the arguments from the command line. Owing to the way in which the `argparse` module works, this function will almost always return an `argparse.Namespace` object.
- A function which does the real work. This function should be designed so that it makes no reference to the command-line options in any way. This means it can be reused in a variety of contexts.

Here's our *real work* function, `display()`:

```
from ch03_r05 import haversine, MI, NM, KM
def display(lat1, lon1, lat2, lon2, r):
    r_float = {'NM': NM, 'KM': KM, 'MI': MI}[r]
    d = haversine( lat1, lon1, lat2, lon2, r_float )
    print( "From {lat1},{lon1} to {lat2},{lon2}"
          "in {r} = {d:.2f}".format_map(vars()))
```

We've imported the core calculation, `haversine()`, from another module. We've provided argument values to this function and used `format()` to display the final result message.

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`, *Function Definitions*:

$$\alpha = sin2\left(\frac{lat_2 - lat_1}{2}\right) + \cos\left(lat_1\right)\cos\left(lat_2\right)sin^2\left(\frac{lon_2 - lon_1}{2}\right)$$

$$c = 2\arcsin\left(\sqrt{a}\right)$$

The essential calculation yields the central angle, c, between two points, given as (lat_1, lon_1) and (lat_2, lon_2). The angle is measured in radians. We convert it into distance by multiplying it by the Earth's mean radius in some units. If we multiply the angle c by a radius of 3,959 miles, we'll get the distance represented by the angle in miles.

Note that we expect the distance conversion factor, *r*, to be provided as a string. This function will then map the string to an actual floating-point value.

For details on the `format()` method, note that we're using a variation on the *Debugging with "format".format_map(vars())* recipe.

Here's how the function looks when it's used inside Python:

```
>>> from ch05_r04 import display
>>> 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 helpful feature because it lets us decompose the problem. We've separated the user experience from the essential calculation.

How to do it...

1. Define the overall argument parsing function:

```
def get_options():
```

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 distance

   We can use Nautical Miles as a handy default so that sailors get the answers they need:

```
parser.add_argument('-r', 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 two kinds of arguments. The first is the `-r`, argument, which starts with `-` to mark it as optional. Sometimes, a `--` is used with a longer name. In some cases, we'll provide both alternatives, as follows:

```
add_argument('--radius', '-r'....)
```

   The action is to store the value which follows the `-r` on the command-line. We've listed the three possible choices and provided a default. The parser will validate the input and write appropriate errors if the input isn't one of these three values.

The mandatory arguments are provided without a – prefix. We used an action of `store`; this is the default action and doesn't really need to be stated. The function provided as the `type` argument is used to convert the source string to an appropriate Python object. This is also the ideal way to validate complex input values. We'll look at the `point_type()` validation function int this section.

4. Evaluate the `parse_args()` method of the parser object created in step 2:

```
options = parser.parse_args()
```

By default, this uses the values from `sys.argv`, which are the command-line argument values entered by the user. We can provide an explicit argument if we need to modify the user-supplied command-line in some way.

Here's the final function:

```
def get_options():
    parser = argparse.ArgumentParser()
    parser.add_argument('-r', 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()
    return options
```

This relies on the `point_type()` validation function. This is needed because the default input type is defined by the `str()` function. This assures that the values of arguments will be string objects. We've provided the `type` argument so that we can inject a type conversion. We might use `type = int` or `type = float` to convert to a number.

In our example, we used `point_type()` to convert a string to a (*latitude*, *longitude*) two-tuple:

```
def point_type(string):
    try:
        lat_str, lon_str = string.split(',')
        lat = float(lat_str)
        lon = float(lon_str)
        return lat, lon
    except Exception as ex:
        raise argparse.ArgumentTypeError from ex
```

This function parses the input values. First, it separates the two values at the `,` character. It attempts a floating-point conversion on each part. If the `float()` functions both work, we have a valid latitude and longitude that we can return as a pair of floating-point values.

If anything goes wrong, an exception will be raised. From this exception, we'll raise an `ArgumentTypeError` exception. This is used by the `argparse` module to report the error to the user.

Here's the main script that combines the option parser and the output display functions:

```
if __name__ == "__main__":
    options = get_options()
    lat_1, lon_1 = options.p1
    lat_2, lon_2 = options.p2
    r = {'NM': NM, 'KM': KM, "MI": MI}[options.r]
    display(lat_1, lon_1, lat_2, lon_2, r)
```

This main script does a few things to connect the user inputs to the displayed output:

1. Parse the command-line options. These are all present in the options object.
2. Expand the `p1` and `p2` (*latitude, longitude*) two-tuples into four individual variables.
3. Evaluate the `display()` function.

How it works...

The argument parser works in three stages:

1. Define the overall context by creating a parser object as an instance of `ArgumentParser`. We can provide information such as the overall program description. We can also provide a formatter and other options here.
2. Add individual arguments with the `add_argument()` method. These can include optional arguments as well as required arguments. Each argument can have a number of features to provide different kinds of syntax. We'll look at a number of the alternatives in the *There's more...* section.
3. Parse the actual command-line inputs. The parser's `parse()` method will use `sys.argv` automatically. We can provide an explicit value instead of the `sys.argv` values. The most common reason for providing an override value is to allow for more complete unit testing.

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 are provided on the command line, as follows:

```
python3 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 list of files can be processed using an argument defined as follows:

```
parser.add_argument('file', nargs='*')
```

All of the names on the command line that do not start with the – character will be collected into the `file` value in the object built by the parser.

We can then use the following:

```
for filename in options.file:
    process(filename)
```

This will process each file given on the command line.

For Windows programs, the shell doesn't glob, and the application must deal with filenames that have wild card patterns in them. The Python `glob` module can help with this. Also, the `pathlib` module can create `Path` objects, which include globbing features.

We may have to make even more complex argument parsing options. Very complex applications may have dozens of individual commands. As an example, look at the `git` version-control program; this application uses dozens of separate commands such as `git clone`, `git commit`, or `git push`. Each of these commands has unique argument parsing requirements. We can use `argparse` to create a complex hierarchy of these commands and their distinct sets of arguments.

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**: The `-o` or `--option` arguments are often used to enable or disable features of a program. These are often implemented with `add_argument()` parameters of `action='store_true', default=False`.

Sometimes the implementation is simpler if the application uses `action='store_false', default=True`. The choice of default value and stored value may simplify the programming, but it won't change the user's experience.

- **Simple options with non-trivial objects**: The user sees this is as simple `-o` or `--option` arguments. We may want to implement this using a more complex object that's not a simple Boolean constant. We can use `action='store_const', const=some_object, default=another_object`. As modules, classes, and functions are also objects, a great deal of sophistication is available here.
- **Options with values**: We showed `-r unit` as an argument that accepted the string name for the units to use. We implemented this with an `action='store'` assignment to store the supplied string value. We can also use the `type=function` option to provide a function that validates or converts the input into a useful form.
- **Options that increment a counter**: One common technique is to have a debugging log that has multiple levels of detail. We can use `action='count', default=0` to count the number of times a given argument is present. The user can provide `-v` for verbose output and `-vv` for very verbose output. The argument parser treats `-vv` as two instances of the `-v` argument, which means that the value will increase from the initial value of `0` to `2`.
- **Options that accumulate a list**: We might have an option for which the user might want to provide more than one value. We could, for example, use a list of distance values. We could have an argument definition with `action='append', default=[]`. This would allow the user to say `-r NM -r KM` to get a display in both nautical miles and kilometers. This would require a significant change to the `display()` function, of course, to handle multiple units in a collection.
- **Show the help text**: If we do nothing, then `-h` and `--help` will display a help message and exit. This will provide the user with useful information. We can disable this or change the argument string, if we need to. This is a widely used convention, so it seems best to do nothing so that it's a feature of our program.
- **Show the version number**: It's common to have `--Version` as an argument to display the version number and exit. We implement this with `add_argument("--Version", action="version", version="v 3.14")`. We provide an action of `version` and an additional keyword argument that sets the version to display.

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 use simple, widely used argument styles, our users are somewhat more likely to understand how our application works.

There are a few Linux commands, which have even more complex command-line syntax. Some Linux programs, such as `find` or `expr`, have arguments that can't easily be processed by `argparse`. For these edge cases, we would need to write our own parser using the values of `sys.argv` directly.

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 cmd for creating command-line applications

There are several ways of creating 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 how to use `argparse` to create applications with which a user can interact from the OS command line.

We have a third 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.

This is related to material in `Chapter 7`, *More Advanced Class Design*. We'll add features to a class definition to create a unique subclass.

Here's how the interaction will look, we've marked user input like this: "`help`":

```
Starting with 100
Roulette> help

Documented commands (type help <topic>):
========================================
bet  help

Undocumented commands:
======================
done  spin  stake
Roulette> help bet
Bet <name> <amount>
        Name is one of even, odd, red, black, high, or low

Roulette> bet black 1
Roulette> bet even 1
Roulette> spin
Spin ('21', {'red', 'high', 'odd'})
Lose even
Lose black
... more interaction ...
Roulette> done
Ending with 93
```

There's an introductory message from the application. It shows the player's starting stake, that is, how much they have to bet. The application displays a prompt, `Roulette>`. The user can then enter any of the five available commands.

When we enter `help` as a command, we see a display of the available commands. Only two have any documentation. The other three have no further details available.

When we enter `help bet`, we see the detailed documentation for the `bet` command. The description tells us to provide a bet name from the available six choices and a bet amount.

We create two bets—one on black and one on even. We then enter the `spin` command to spin the wheel. This displays the outcome—the number `21`—which is red, high, and odd. Both of our bets are losses.

We've omitted a few more interactions that didn't win very much, either. When we entered the `done` command, the final stake was shown. If the simulation was more detailed, it might also show some aggregate statistics on spins, wins, and losses.

Getting ready

The core feature of the `cmd.Cmd` application is a **read-evaluate-print loop** (**REPL**). This kind of application works well when there are a large number of individual state changes and a large number of commands to make those state changes.

We'll use a simple simulation of a subset of the bets in *Roulette* as an example. The idea is to allow the user to create one or more bets and then spin a simulated *Roulette* wheel. While proper casino *Roulette* has a dizzying array of possible bets, we'll focus on just six:

- red, black
- even, odd
- high, low

An *American Roulette* wheel has 38 bins. The numbers 1 to 36 are colored red and black. There are two other bins, zero and double zero, which are green. These two extra bins are defined as neither even nor odd and neither high nor low. There are only a few ways to bet on the zeroes, but numerous ways to bet on numbers.

We'll represent the *Roulette* wheel using some helper functions that build a collection of bins. Each bin will have a string that shows the number and a set of bet names that are winners.

We can define a generic bin with some simple rules to determine which bets are in the winning set:

```
red_bins = (1, 3, 5, 7, 9, 12, 14, 16, 18,
    21, 23, 25, 27, 28, 30, 32, 34, 36)

def roulette_bin(i):
    return str(i), {
        'even' if i%2 == 0 else 'odd',
        'low'  if 1 <= i < 19 else 'high',
        'red'  if i in red_bins else 'black'
    }
```

The `roulette_bin()` function returns a two-tuple with the string representation for the bin number and a set of three winning propositions.

For `0` and `00`, we'll need something a little different:

```
def zero_bin():
    return '0', set()

def zerozero_bin():
    return '00', set()
```

The `zero_bin()` function returns a string bin number and an empty set. The `zerozero_bin()` function returns a special string to show that it's `00`, plus the empty set to show that none of the defined bets are winners.

We can combine the results of these three functions to create a complete *Roulette* wheel. The whole wheel will be modeled as a list of bin tuples:

```
def wheel():
    b0 = [zero_bin()]
    b00 = [zerozero_bin()]
    b1_36 = [
        roulette_bin(i) for i in range(1,37)
    ]
    return b0+b00+b1_36
```

We've built a simple list that contains the complete set of bins: a zero, a double zero, and the numbers 1 through 36. We can now use the `random.choice()` function to select a bin at random. This will tell us which bets win and which bets lose.

How to do it...

1. Import the cmd module:

```
import cmd
```

2. Define an extension to `cmd.Cmd`:

```
class Roulette(cmd.Cmd):
```

3. Define any initialization required in the `preloop()` method:

```
    def preloop(self):
        self.bets = {}
        self.stake = 100
        self.wheel = wheel()
```

This `preloop()` method is evaluated just once when the processing starts. We've used this to initialize a dictionary for bets and the player's stake. We also created an instance of the wheel collection. The self argument is a requirement for methods within a class. For now, it's a simply required syntax. In `Chapter 6`, *Basics of Classes and Objects*, we'll look at this more closely.

Note that this is indented within the `class` statement.

Initialization can also be done in the `__init__()` method. This is a bit more complex, though, because we have to use `super()` to ensure that the `Cmd` class initialization is done first.

4. For each command, create a `do_command()` method. The name of the method will be the command, prefixed by `do_`. The user's input text after the command will be provided as an argument value to the method. Here are two examples for the `bet` command and the `spin` command:

```
def do_bet(self, arg_string):
    pass
def do_spin(self, arg_string):
    pass
```

5. Parse and validate the arguments to each command. The user's input after the command will be provided as the value of the first positional argument to the method.

If the arguments are invalid, the method should print a message and return. If the arguments are valid, the method can continue past the validation step.

For our example, the `spin` command doesn't require any input. We can ignore the argument string. To be more complete, we might want to display an error if the string is non-empty.

The `bet` command, however, does have a bet, which must be one of the six valid bet names. We might want to check for duplicate bets. We might also want to check for abbreviated bet names. Each of six bets has a unique first letter.

As an extension, a bet can also have an amount. We looked at parsing strings in the *String parsing with regular expressions* recipe in `Chapter 1`, *Numbers, Strings, and Tuples*. For this example, we'll simply handle the name of the bet:

```
def do_spin(self, arg_string):
    if len(self.bets) == 0:
        print("No bets have been placed")
```

```
            return
        # Happy path: more goes here.

    BET_NAMES = set(['even', 'odd', 'high', 'low', 'red', 'black'])

    def do_bet(self, arg_string):
        if arg_string not in BET_NAMES:
            print("{0} is not a valid bet".format(arg_string))
            return
        # Happy path: more goes here.
```

6. Write the happy path processing for each command. For our example, the `spin` command will resolve the bets. The `bet` command will accumulate another bet. Here's the `do_bet()` happy path:

```
self.bets[arg_string] = 1
```

We've added the user's bet to the `self.bets` mapping with the amount. For this example, we'll treat all bets as having the same minimal amount.

7. Here's the `do_spin()` happy path that resolves all of the bets:

```
self.spin = random.choice(self.wheel)
print("Spin", self.spin)
label, winners = self.spin
for b in self.bets:
    if b in winners:
        self.stake += self.bets[b]
        print("Win", b)
    else:
        self.stake -= self.bets[b]
        print("Lose", b)
self.bets= {}
```

First, we spun the wheel to get a winning bet. Then, we examined each of the player's bets to see which of those match the set of winning bets. If the player's bet, `b`, is in the set of winning bets, we'll increase their stake. Otherwise, we'll reduce their stake.

All of the bets in this example pay 1:1. If we want to extend the example to other kinds of bets, we have to provide proper odds for the various bets.

8. Write the main script. This will create an instance of this class and execute the `cmdloop()` method:

```
if __name__ == "__main__":
    r = Roulette()
    r.cmdloop()
```

We've created an instance of our `Roulette` subclass of `Cmd`. When we execute the `cmdloop()` method, the class will write any introductory messages that have been provided, write the prompt, and read a command.

How it works...

The `Cmd` module contains a large number of built-in features for displaying a prompt, reading input from a user, and then locating the proper method based on the user's input.

For example, when we enter `bet black`, the built-in methods of the `Cmd` superclass will strip the first word from the input, `bet`, prefix this with `do_`, and then evaluate the method that implements the command.

If there's no `do_bet()` method, the command processor writes an error message. This is done automatically, we don't need to write any code at all.

Since we wrote a `do_bet()` method, this will be invoked. The text after the command, `black` in this case, will be provided as the positional argument value.

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 can 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_`. They're available in a class-level `__dict__` attribute. Introspection is an advanced topic, one that will be touched on in `Chapter 7`, *More Advanced Class Design*.

There's more...

The `Cmd` class has a number of additional places where we can add interactive features:

- We can define `help_*()` methods that become part of the miscellaneous help topics.
- When any of the `do_*` methods return a value, the loop will end. We might want to add a `do_quit()` method that has `return True` as it's body. This will end the command-processing loop.
- We might provide a method named `emptyline()` to respond to blank lines. One choice is to do nothing quietly. Another common choice is to have a default action that's taken when the user doesn't enter a command.
- The `default()` method is evaluated when the user's input does not match any of the `do_*` methods. This might be used for more advanced parsing of the input.
- 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. This also requires a `do_*` method that returns a value—any non-`False` value—to end the command loop.

Also, there are a number of attributes we can set. These are class-level variables that would be peers to the method definitions:

- The `prompt` attribute is the prompt string to write. For our example, we can do the following:

```
class Roulette(cmd.Cmd):
    prompt="Roulette> "
```

- The `intro` attribute is the introductory message.
- We can tailor the help output by setting `doc_header`, `undoc_header`, `misc_header`, and `ruler` attributes. These will all alter how the help output looks.

The goal is to be able to create a tidy class that handles user interaction in a way that's simple and flexible. This class creates an application that has a lot of features in common with Python's REPL. It also has features in common with many command-line programs that prompt for user input.

One example of these interactive applications is the command-line FTP client in Linux. It has a prompt of `ftp>`, and it parses dozens of individual FTP commands. Entering `help` will show all of the various internal commands that are part of FTP interaction.

See also

- We'll look at class definitions in `Chapter 6`, *Basics of Classes and Objects*, and `Chapter 7`, *More Advanced Class Design*

Using the OS environment settings

There are several ways to look at the span of time for user inputs:

- Interactive data: This is provided by the user in a kind of *right now* time span.
- Command-line arguments provided when the program is started: These values often span one full execution of a program.
- Environment variables set at the OS level: These can be set at the command line, making them almost as interactive as the command that starts an application:
    - They can be configured for a user in a `.bashrc` file or `.profile` file. This makes them more persistent and slightly less interactive than the command-line.
    - In Windows, there's the **Advanced Settings** option that allows someone to set a long-term configuration. These are often inputs to multiple executions of a program.
- Configuration file settings: These vary widely by application. The idea is to edit a file and make these options or arguments available for long periods of time. These might apply to multiple users or even to all users. Configuration files often have the longest time span.

In the *Using input() and getpass() for user input* and *Using cmd for creating command-line applications* recipes, we looked at interaction with the user. In the *Using argparse to get command-line input* recipe, we looked at how to handle command-line arguments. We'll look at configuration files in `Chapter 11`, *Application integration*.

The environment variables are available through the `os` module. How can we have an application's configuration based on these OS-level settings?

Getting ready

We may want to provide information of various types to a program via OS settings. There's a profound limitation here: the OS settings can only be string values. This means that many kinds of settings will require some code to parse the value and create proper Python objects from the string.

When we work with `argparse` to parse command-line arguments, this module can do some data conversions for us. When we use `os` to process environment variables; we'll have to implement the conversion ourselves.

In the *Using argparse to get command-line input* recipe, we wrapped the `haversine()` function in a simple application that parsed command-line arguments.

At the OS level, we created a program that worked like this:

```
slott$ python3 ch05_r04.py -r KM 36.12,-86.67 33.94,-118.40
From (36.12, -86.67) to (33.94, -118.4) in KM = 2887.35
```

After using this for a while, we've found 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.

We'll set set an OS environment variable, `UNITS`, with the distance units. We can set another variable, `HOME_PORT`, with the home point. We want to be able to do the following:

```
slott$ UNITS=NM
slott$ HOME_PORT=36.842952,-76.300171
slott$ python3 ch05_r06.py 36.12,-86.67
From 36.12,-86.67 to 36.842952,-76.300171 in NM = 502.23
```

The units and the home point values are provided to the application via the OS environment. This can be set in a configuration file so that we can make easy changes. It can also be set manually, as shown in the example.

How to do it...

1. Import the `os` module. The OS environment is available through this module:

```
import os
```

2. Import any other classes or object needed for the application:

```
from ch03_r05 import haversine, MI, NM, KM
```

3. Define a function that will use the environment values as defaults for optional command-line arguments. The default set of arguments to parse come from `sys.argv`, so it's important to also import the `sys` module:

```
def get_options(argv=sys.argv):
```

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("Invalid value for UNITS, not KM, NM, or MI")
    default_home_port = os.environ.get('HOME_PORT')
```

The `sys.exit()` function handles the error processing nicely. It will print the message and exit with a non-zero status code.

5. Create the `parser` attribute. Provide any default values for the relevant arguments. This depends on the `argparse` module, which must also be imported:

```
            parser = argparse.ArgumentParser()
    parser.add_argument('-r', 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)
    options = parser.parse_args(argv[1:])
```

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()`:

```
if options.p2 is None:
sys.exit("Neither HOME_PORT nor p2 argument provided.")
```

7. Return the `options` object with the set of valid arguments:

```
return options
```

This will allow the `-r` argument and the second point to be completely optional. The argument parser will use the configuration information to supply default values if these are omitted from the command line.

Use the *Using argparse to get command-line input* recipe for ways to process the options created by the `get_options()` function.

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 is used.

In the instance of the `UNITS` variable, the application uses kilometers as the default if not, then the OS environment variable is set.

This gives us three tiers of interaction:

- We can define a setting in a `.bashrc` file. Alternatively, we can use the Windows **Advanced Settings** option to make a change that is persistent. This value will be used each time we log in or create a new command window.
- We can set the OS environment interactively at the command line. This will last as long as our session lasts. When we logout, or close the command window, this value will be lost.
- We can provide a unique value through the command-line arguments each time the program is run.

Note that there's no built-in or automatic validation of the values retrieved from environment variables. We'll need to validate these strings to ensure that they're meaningful.

Also note that we've repeated the list of valid units in several places. This violates the **Don't Repeat Yourself** (**DRY**) principle. A global variable with this list is a good improvement to make.

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 as follows:

```
['ch05_r06.py', '-r', 'NM', '36.12,-86.67']
```

We have to skip the initial value in `sys.argv[0]` at some point in the processing. We have two choices:

- In this recipe, we discard the extra item as late as possible in the parsing process. The first item is skipped when providing `sys.argv[1:]` to the parser.
- In the previous example, we discarded the value earlier in the processing. The `main()` function used `options = get_options(sys.argv[1:])` to provide the shorter list to the parser.

Generally, the only relevant distinction between the two approaches depends on the number and complexity of the unit tests. This recipe will require a unit test that includes an initial argument string, which will be discarded during parsing.

See also

- We'll look at numerous ways to handle configuration files in `Chapter 11`, *Application integration*

6
Basics of Classes and Objects

In this chapter, we will look at the following recipes:

- Using a class to encapsulate data and processing
- Designing classes with lots of processing
- Designing classes with little unique processing
- Optimizing small objects with __slots__
- Using more sophisticated collections
- Extending a collection – a list that does statistics
- Using properties for lazy attributes
- Using settable properties to update eager attributes

Introduction

The point of computing is to process data. Even when building something like an interactive game, both the game state and the player's actions are the data; the processing computes the next game state and the display update.

Some games can have a relatively complex internal state. When we think of console games with multiple players and sophisticated graphics, there are complex, real-time state changes.

On the other hand, when we think of a casino game, such as *Craps*, the game state is very simple. There may be no point established, or one of the numbers 4, 5, 6, 8, 9, or 10 may be the established point. The transitions are relatively simple, and are often denoted by moving markers and chips around on the casino table. The data includes the current state, player actions, and rolls of the dice. The processing is the rules of the game.

A game such as *Blackjack* has a somewhat more complex internal state change as each card is accepted. In games where the hands can be split, the state of play can become quite complex. The data includes the current game state, the player's commands, and the cards drawn from the deck. Processing is defined by the rules of the game as modified by any house rules.

In the case of *craps*, the player may place bets. Interestingly, the player's input has no effect on the game state. The internal state of the game object is determined entirely by the next throw of the dice. This leads to a class design that's relatively easy to visualize.

In this chapter, we will create classes that implement a number of statistical formulae. The math can be a little daunting at first. Almost everything will be based on the summation of a sequence of values, often shown as $\sum x$. In many cases, this can be implemented using Python's `sum()` function.

Using a class to encapsulate data and processing

The essential idea of computing is to process data. This is exemplified when we write functions that process data. We looked at this in `Chapter 3`, *Function Definitions*.

Often, we'd like to have a number of closely related functions that work with a common data structure. This concept is the heart of object-oriented programming. A class definition will contain a number of methods that all control the internal state of an object.

The unifying concept behind a class definition is often captured as a summary of the responsibilities allocated to the class. How can we do this 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 the casino game of *Craps*. The goal is to use a simulation of results to help invent a better playing strategy. This will save us from losing real money while we try to beat the house edge.

There's an important distinction between the class definition and an instance of the class, called an **object**. We call this idea **object-oriented programming** as a whole. Our focus is on writing class definitions. Our overall application will create instances of the classes. The behavior that emerges from the collaboration of the instances is the overall goal of the design process.

Most of the design effort is on class definitions. Because of this, the name object-oriented programming can be misleading.

The idea of **emergent behavior** is an essential ingredient in object-oriented programming. We don't specify every behavior of a program. Instead, we decompose the program into objects, and define the object's state and behavior via the object's classes. The programming decomposes into class definitions based on their responsibilities and collaborations.

An object should be viewed as a thing—a noun. The behaviors of the class should be viewed as verbs. This gives us a hint as to how we can proceed to design classes that work effectively.

Object-oriented design is often easiest to understand when it relates to tangible real-world things. It's often easier to write software to simulate a playing card than to create software that implements an **Abstract Data Type**.

For this example, we'll simulate the rolling of dice. For some games, such as the casino game of *Craps*, two dice are used. We'll define a class that models the pair of dice. To be sure that the example is tangible, we'll model the pair of dice in the context of simulating a casino game.

How to do it...

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 short sentences, and emphasize the nouns and verbs:
    - 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.
    - The total of the dice changes the state of the *craps* game. Those rules are separate from the dice, however.
    - If the two dice match, the number was rolled the hard way. If the two dice do not match, the number was easy. Some bets depend on this hard-easy distinction.

2. Identify all of the nouns in the sentences. Nouns may identify different classes of objects. These are **collaborators**. Examples include player and game. Nouns may also identify attributes of objects in questions. Examples include face and point value.
3. Identify all the verbs in the sentences. Verbs are generally methods of the class in question. Examples include rolled and match. Sometimes, they are methods of other classes. One example is change the state, which applies to *Craps*.
4. Identify any adjectives. Adjectives are words or phrases that clarify a noun. In many cases, some adjectives will clearly be properties of an object. In other cases, the adjectives will describe relationships among objects. In our example, a phrase such as *the total of the dice* is an example of a prepositional phrase taking the role of an adjective. The *the total of* phrase modifies the noun *the dice*. The total is a property of the pair of dice.
5. Start writing the class with the `class` statement:

```
class Dice:
```

6. Initialize the object's attributes in the `__init__` method:

```
def __init__(self):
    self.faces = None
```

We'll model the internal state of the dice with the `self.faces` attribute. The `self` variable is required to be sure that we're referencing an attribute of a given instance of a class. The object is identified by the value of the instance variable, `self`.

We could also put some other properties here. The alternative is to implement the properties as separate methods. The details of this design decision are the subject of the *Using properties for lazy attributes* recipe later on in this chapter.

7. Define the object's methods based on the various verbs. In our case, we have several methods that must be defined:
    - Here's how we can implement dice are rolled by a player:

```
def roll(self):
    self.faces = (random.randint(1,6), random.randint(1,6))
```

We've updated the internal state of the dice by setting the `self.faces` attribute. Again, the `self` variable is essential for identifying the object to be updated.

Note that this method mutates the internal state of the object. We've elected to not return a value. This makes our approach somewhat like the approach of Python's built-in collection classes. Any method that mutates the object does not return a value.

- This method helps implement the total of the dice changes the state of the *craps* game. The game is a separate object, but this method provides a total that fits the sentence.

```
def total(self):
    return sum(self.faces)
```

These two methods help answer the hardways and easyways questions.

```
def hardway(self):
    return self.faces[0] == self.faces[1]
def easyway(self):
    return self.faces[0] != self.faces[1]
```

It's rare in a casino game to have a rule that has a simple logical inverse. It's more common to have a rare third alternative that has a remarkably bad payoff rule. In this case, we could have defined `easyway` as return `not self.hardway()`.

Here's an example of using the class:

1. First, we'll seed the random number generator with a fixed value so that we can get a fixed sequence of results. This is a way of creating a unit test for this class:

```
>>> import random
>>> random.seed(1)
```

2. We'll create a `Dice` object, `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:

```
>>> from ch06_r01 import Dice
>>> d1 = Dice()
>>> d1.roll()
>>> d1.total()
7
>>> d1.faces
(2, 5)
```

3. We'll create a second `Dice` object, `d2`. We can then set its state with the `roll()` method. We'll look at the result of the `total()` method, as well as the `hardway()` method. We'll examine the state by looking at the `faces` attribute:

```
>>> d2 = Dice()
>>> d2.roll()
>>> d2.total()
4
>>> d2.hardway()
False
>>> d2.faces
(1, 3)
```

4. Since the two objects are independent instances of the `Dice` class, a change to `d2` has no effect on `d1`:

```
>>> d1.total()
7
```

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. Nouns represent things. A good descriptive sentence should focus on tangible, real-world things more than ideas or abstractions.

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 locate an abstract implementation that offers some of the tangible features.

The idea of rolling the dice is an example physical action that we can model with a method definition. Clearly, this action changes the state of the object. In rare cases—one time in 36—the next state will happen to match the previous state.

Adjectives often hold the potential for confusion. The following are descriptions of the most common ways in which adjectives operate:

- Some adjectives, such as first, last, least, most, next, previous, and so on, will have a simple interpretation. These can have a lazy implementation as a method, or an eager implementation as an attribute value.
- Some adjectives are a more complex phrase, such as *the total of the dice*. This is an adjective phrase built from a noun (total) and a preposition (of). This, too, can be seen as a method or an attribute.

- Some adjectives involve nouns that appear elsewhere in our software. We might have a phrase such as *the state of the Craps game*, where *state of* modifies another object, the *Craps* game. This is clearly only tangentially related to the dice themselves. This may reflect a relationship between dice and game.
- We might add a sentence to the problem statement such as *the dice are part of the game*. This can help clarify the presence of a relationship between game and dice. Prepositional phrases, such as *are part of*, can always be reversed to create the statement from the other object's point of view: for example, *The game contains dice*. This can help clarify the relationships among objects.

In Python, the attributes of an object are by default dynamic. We don't specify a fixed list of attributes. We can initialize some (or all) of the attributes in the `__init__()` method of a class definition. Since attributes aren't static, we have considerable flexibility in our design.

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 **S.O.L.I.D.**:

- **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 proper behavior and also adhere to the design principles.

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 7`, *More Advanced Class Design*, we'll look in more depth at class design techniques
- See `Chapter 9`, *Testing*, for recipes on how to write appropriate unit tests for the class

Designing classes with lots of processing

Most of the time, an object will contain all of the data that defines its internal state. However, this isn't always true. There are cases where a class doesn't really need to hold the data, but instead can hold the processing.

Some prime examples of this design are statistical processing algorithms, which are often outside the data being analyzed. The data might be in a `list` or `Counter` object. The processing might be a separate class.

In Python, of course, this kind of processing is often implemented using functions. See `Chapter 3`, *Function Definitions* for more information on this. In some languages, all code must take the form of a class, leading to some extra complexity.

How can we design a class that makes use of Python's array of sophisticated built-in collections?

Getting ready

In `Chapter 4`, *Built-in Data Structures – list, set, dict*, specifically the *Using set methods and operators* recipe, we looked at a statistical process called the **Coupon Collector's Test**. The concept is that each time we perform some process, we save a coupon that describes some aspect or parameter for the process. The question is, how many times do I have to perform the process before I collect a complete set of coupons?

If we have customers assigned to different demographic groups based on their purchasing habits, we might ask how many online sales we have to make before we've seen someone from each of the groups. If the groups are all about the same size, it's trivial to predict the average number of customers we encounter before we get a complete set of coupons. If the groups are different sizes, it's a little more complex to compute the expected time before collecting a full set of coupons.

Let's say we've collected data using a `Counter` object. For more information on the various collections, see `Chapter 4`, *Built-in Data Structures – list, set, dict*, specifically the *Using set methods and operators* and *Avoiding mutable default values for function parameters* recipes. In this case, the customers fall into eight categories with approximately equal numbers.

The data looks like this:

```
Counter({15: 7, 17: 5, 20: 4, 16: 3, ... etc., 45: 1})
```

The key is the number of visits needed to get a full set of coupons. The value is the number of times that it took the given number of visits. In the preceding line of code `15` visits were required seven different times. `17` visits were required five times. This has a long tail. At one point, there were `45` individual visits before a full set of eight coupons was collected.

We want to compute some statistics on this `Counter`. We have two overall strategies for doing this:

- **Extend**: We can extend the `Counter` class definition to add statistical processing. The complexity of this varies with the kind of processing that we want to introduce. We'll cover this in detail in the *Extending a collection – a list that does statistics* recipe, as well as `Chapter 7`, *More Advanced Class Design*.
- **Wrap**: We can wrap the `Counter` object in another class that provides just the features we need. When we do this, though, we'll often have to expose some additional methods that are an important part of Python, but which don't matter much for our application. We'll look at this in `Chapter 7`, *More Advanced Class Design*.

There's a variation on wrapping where we use a statistical computation object to wrap an object from a built-in collection. This often leads to an elegant solution.

We have two ways to design the processing. These two design alternatives apply to both overall architectural choices:

- **Eager**: This means that we'll compute the statistics as soon as possible. The values can then be attributes of the class. While this can improve performance, it also means that any change to the data collection will invalidate the eagerly computed values. We have to examine the overall context to see if this can happen.
- **Lazy**: This means we won't compute anything until it's required via a method function or property. We'll look at this in the *Using properties for lazy attributes* recipe.

The essential math for both designs is the same. The only question is when the computation is done.

We compute the mean using a sum of the expected values. The expected value is the frequency of a value multiplied by the value. The mean, μ, is this:

$$\mu = \sum_{k \in C} f_k \times k$$

Here, k is the key from the `Counter`, C, and f_k is the frequency value for the given key from the `Counter`.

The standard deviation, σ, depends on the mean, μ. This also involves computing a sum of values, each of which is weighted by frequency. The following is the formula:

$$\sigma = \sqrt{\frac{\sum_{k \in C} f_k \times \left(k - \mu\right)^2}{C + 1}}$$

Here, k is the key from the `Counter`, C, and f_k is the frequency value for the given key from the `Counter`. The total number of items in the `Counter` is $C = \sum_{k \in C} f_k$. This is the sum of the frequencies.

How to do it...

1. Define the class with a descriptive name:

```
class CounterStatistics:
```

2. Write the `__init__` method to include the object to which this object will be connected:

```
def __init__(self, raw_counter:Counter):
    self.raw_counter = raw_counter
```

   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.

3. Initialize any other local variables 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()
```

   We've eagerly computed the mean and standard deviation from the `Counter` object, and saved them in two instance variables.

4. Define the required methods for the various values. Here's the calculation of the mean:

```
def compute_mean(self):
    total, count = 0, 0
    for value, frequency in self.raw_counter.items():
        total += value*frequency
        count += frequency
    return total/count
```

5. Here's how we can calculate the standard deviation:

```
def compute_stddev(self):
    total, count = 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.

   Also, this uses `math.sqrt()`. Be sure to add the needed `import math` statement in the Python file.

Here's how we can create some sample data:

```
>>> from ch04_r06 import *
>>> from collections import Counter
>>> def raw_data(n=8, limit=1000, arrival_function=arrival1):
...     expected_time = float(expected(n))
...     data = samples(limit, arrival_function(n))
...     wait_times = Counter(coupon_collector(n, data))
...     return wait_times
```

We've imported functions such as `expected()`, `arrival1()`, and `coupon_collector()` from the `ch04_r06` module. We've also imported the `Counter` collection from the standard library `collections` module.

We defined a function, `raw_data()`, that will generate a number of customer visits. By default, it will be 1,000 visits. The domain will be eight different classes of customers; each class will have an equal number of members. We'll use the `coupon_collector()` function to step through the data, emitting the number of visits required to collect a full set of eight coupons.

This data is then used to assemble a `Counter` object. This will have the number of customers required to get a full set of coupons. Each number of customers will also have a frequency showing how often that number of visits occurred.

Here's how we can analyze the `Counter` object:

```
>>> import random
>>> from ch06_r02 import CounterStatistics
>>> random.seed(1)
>>> data = raw_data()
>>> stats = CounterStatistics(data)
```

```
>>> print("Mean: {0:.2f}".format(stats.mean))
Mean: 20.81
>>> print("Standard Deviation: {0:.3f}".format(stats.stddev))
Standard Deviation: 7.025
```

First, we imported the `random` module so that we could pick a known seed value. This makes it easier to test and demonstrate an application because the random numbers are consistent. We also imported the `CounterStatistics` class from the `ch06_r02` module.

Once we have all of the items defined, we can force the `seed` to a known value, and generate the coupon collector test results. The `raw_data()` function will emit a `Counter` object, which we called data.

We'll use the `Counter` object to create an instance of the `CounterStatistics` class. We'll assign this to the `stats` variable. Creating this instance will also compute some summary statistics. These values are available as the `stats.mean` attribute and the `stats.stddev` attribute.

For a set of eight coupons, the theoretical average is `21.7` visits to collect all coupons. It looks like the results from `raw_data()` show behavior that matches the expectation of random visits. This is sometimes called the **null hypothesis**—the data is random.

How it works...

This class encapsulates two complex algorithms, but doesn't include any data that changes state. This kind of class doesn't need to retain a lot of data. Instead, the design performs all of the computations as soon as possible.

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 is paid once; each time an attribute value is used, no further calculating is required.

The disadvantage of this design is that a change to the underlying `Counter` object makes the `CounterStatistics` object obsolete. Generally, we use this kind of design when the `Counter` isn't going to change. The example creates a single, static `Counter`, which is used to create `CounterStatistics`.

There's more...

If we need to have stateful objects, we can add update methods that can change the `Counter` object. For example, we can introduce a method to add another value by delegating the work to the associated `Counter`. This switches the design pattern from a simple connection between computation and collection to a proper wrapper around the collection.

The method might look like this:

```
    def add(self, value):
        self.raw_counter[value] += 1
        self.mean = self.compute_mean()
        self.stddev = self.compute_stddev()
```

First, we've updated the state of the `Counter`. Then, we recomputed all of the derived values. This kind of processing might create tremendous computation overheads. There needs to be a compelling reason to recompute the mean and standard deviation after every value is changed.

There are considerably more efficient solutions. For example, if we save two intermediate sums and an intermediate count, we can update the sums and counts by computing the mean and standard deviation efficiently.

For this, we might have an `__init__()` method that looks like this:

```
    def __init__(self, counter:Counter=None):
        if counter:
            self.raw_counter = counter
            self.count = sum(self.raw_counter[k] for k in self.raw_counter)
            self.sum = sum(self.raw_counter[k]*k for k in self.raw_counter)
            self.sum2 = sum(self.raw_counter[k]*k**2 for k in
self.raw_counter)
            self.mean = self.sum/self.count
            self.stddev = math.sqrt((self.sum2-
self.sum**2/self.count)/(self.count-1))
        else:
            self.raw_counter = Counter()
            self.count = 0
            self.sum = 0
            self.sum2 = 0
            self.mean = None
            self.stddev = None
```

We've written this method to work either with a `Counter` or without a `Counter`. If no data is provided, it will start with an empty collection, and zero values for the various sums.

When the count is zero, the mean and standard deviation have no meaningful value, so `None` is provided.

If a `Counter` is provided, then a `count`, `sum`, and sum of squares are computed. These can be incrementally adjusted easily, quickly recomputing the `mean` and standard deviation.

When a single new value is added, the following method will incrementally recompute the various derived values:

```
def add(self, value):
    self.raw_counter[value] += 1
    self.count += 1
    self.sum += value
    self.sum2 += value**2
    self.mean = self.sum/self.count
    if self.count > 1:
        self.stddev = math.sqrt(
            (self.sum2-self.sum**2/self.count)/(self.count-1))
```

Updating the `Counter` object, the `count`, the `sum`, and the sum of squares is clearly necessary to be sure that the `count`, `sum`, and sum of squares values match the `self.raw_counter` collection at all times. Since we know the `count` must be at least `1`, the mean is easy to compute. The standard deviation requires at least two values, and is computed from the `sum` and sum of squares.

Here's the formula for this variation on standard deviation:

$$\sigma = \sqrt{\frac{\sum_{k \in C} f_k \times k^2 - \frac{\left(\sum_{k \in C} f_k \times k\right)^2}{C}}{C-1}}$$

This involves computing two sums. One sum involves frequency times the value squared. The other sum involves the frequency and the value, with the overall sum being squared. We've used *C* to represent the total number of values; this is the sum of the frequencies.

See also

- In the *Extending a collection – a list that does statistics* recipe, we'll look at a different design approach where these functions are used to extend a class definition.
- We'll look at different approach in the *Using properties for lazy attributes* recipe. This alternative recipe will use properties and compute the attributes as needed.
- In the *Designing classes with little unique processing* recipe we'll look at a class with no real processing. It acts as a polar opposite of this class.

Designing classes with little unique processing

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, a class can be designed that depends only on built-in Python features and doesn't require any unique method functions.

In many cases, Python's built-in container classes can cover almost all of the various use cases for us. The small problem is that the syntax for a dictionary or a list isn't quite so elegant as the syntax for attributes of an object.

How can we create a class that allows us to use `object.attribute` syntax instead of `object['attribute']`?

Getting ready

There are really only two cases for any kind of class design:

- Is it stateless? Does it embody a number of attributes, but never changes?
- Is it stateful? Will there be state changes for the various attributes?

A stateful design is slightly more general. We can always use a stateful implementation and avoid making any changes to the object to support stateless objects. However, there are some significant storage and performance advantages of using truly stateless objects.

We'll use two kinds of class to illustrate both kinds of design:

- **Stateless**: 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.
- **Stateful**: We'll define a class to describe a player's current state in a game of *Blackjack* where there is a dealer's hand, the player's hand(s), plus an optional insurance bet. There are a number of aspects of play that grow during each hand.

How to do it...

We'll look at stateless objects and then stateful objects. For stateful objects that have no methods, we have two more choices: We can use a new class or we can leverage an existing class. These choices lead to three small recipes.

Stateless objects

1. We'll base stateless objects on `collections.namedtuple`.:

```
from collections import namedtuple
```

2. Define the class name, which will be used twice:

```
Card = namedtuple('Card',
```

3. Define the attributes of the object:

```
Card = namedtuple('Card', ('rank', 'suit'))
```

Here's how we can use this class definition to create `Card` objects:

```
>>> from collections import namedtuple
>>> Card = namedtuple('Card', ('rank', 'suit'))
>>> 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]
8
```

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 rank attribute because it's defined first in the sequence of attribute names.

This kind of class definition is relatively rare. It has a fixed, defined set of attributes. Generally, Python class definitions have dynamic attributes. Also, the 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):
  File
"/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/doctest.py
", line 1318, in __run
    compileflags, 1), test.globs)
  File "<doctest default[0]>", line 1, in <module>
    eight_hearts.suit = '\N{Black Spade Suit}'
AttributeError: can't set attribute
```

We attempted to change the `suit` attribute of the object. This raised an `AttributeError` exception.

Stateful objects with a new class

1. Define the new class:

```
class Player:
    pass
```

2. We've written an empty class definition. An instance of this class is created easily with something like the following:

```
p = Player()
```

We can then add attributes to the object with statements such as the following:

```
p.stake = 100
```

While this can work out well, it's often helpful to add a few more features to a class definition. Generally, we'll add methods, including the `__init__()` method, to initialize the instance variables of the object.

Stateful objects using an existing class

Rather than defining an empty class, we can also use modules in the standard library. We can use the `argparse` module or the `types` module for this:

1. Import the module.

   The `argparse` module includes the class `Namespace`, which can be used instead of an empty class definition:

   ```
   from argparse import Namespace
   ```

   We can also use the `SimpleNamespace` from the types module. It looks like this:

   ```
   from types import SimpleNamespace
   ```

2. Create the class as a reference to the `SimpleNamespace` or `Namespace`:

   ```
   Player = SimpleNamespace
   ```

How it works...

Any of these techniques will define a class that can have an indefinite number of attributes. However, the `SimpleNamespace` has a more flexible constructor than defining our own class:

```
>>> from types import SimpleNamespace
>>> Player = SimpleNamespace
>>> player_1 = Player(stake=100, hand=[], insurance=None, bet=None)
>>> player_1.bet = 10
>>> player_1.stake -= player_1.bet
>>> player_1.hand.append( eight_hearts )
>>> player_1
namespace(bet=10, hand=[Card(rank=8, suit='♡')], insurance=None, stake=90)
```

We've created a new class named `Player`. We don't provide a list of attributes, since they're dynamic.

When we constructed the `player_1` object, we provided a list of attributes that we'd like to create as part of that object. After creating the object, we can then make state changes to it; we set the `player_1.bet` value, updated the `player_1.stake`, and also updated the `player_1.hand`.

When we display the object, all of the attributes are shown. Typically, they're provided in alphabetical order, making it slightly easier to write unit test examples.

When we use a `namedtuple()` function, we're creating a class object. We provide a class name as a string, as well as attribute names that will parallel the positional values for a tuple. The resulting object needs to be assigned to a variable, and it's best practice to make sure that the class name provided as an argument to the `nametuple()` function and the variable name are the same.

The class object created by `namedtuple()` is the same kind of class object that would be created by the `class` statement. Indeed, if you want to see the source, you can use `print(Card._source)` to see exactly what was used to create the class.

A `namedtuple` class is essentially a tuple with the added feature of named attributes. Like all other tuple objects, it's immutable—once built, it cannot be changed.

When we use the `SimpleNamespace`, we're using a very simple class definition that has (almost) no methods. Because attributes are generally dynamic, this class allows us to `set`, `get`, and `delete` attributes freely.

Classes that are not subclasses of `tuple` or that use `__slots__` (a topic we'll look at in the *Optimizing small objects with __slots__* recipe) are very flexible. There are also some very advanced techniques for altering the way attributes behave. These rely on deeper knowledge of how Python's special method names work.

There's more...

In many cases, we'll decompose our application processing into two broad categories of class definitions:

- **Data – collections and items**: We'll use built-in collection classes, collections from the standard library, and perhaps even items based on `namedtuple()`, or `SimpleNamespace`, or other class definitions that seem to focus on generic collections of data.
- **Processing**: We'll define classes in a way similar to the example shown in the *Designing classes with lots of processing* recipe. These processing classes generally depend on data objects.

The idea of cleanly separating the data from the processing fits with several of the S.O.L.I.D. design principles. In particular, it aligns our classes with the Single Responsibility Principle, the Open/Closed Principle, and the Interface Segregation Principle. We can create classes with the kind of narrow focus that makes change (via subclass extension) relatively simple.

See also

- In the *Designing classes with lots of processing* recipe we'll look at a class that is entirely processing and almost no data. It acts as the polar opposite of this class.

Optimizing small objects with __slots__

The general case for an object allows a dynamic collection of attributes, each of which has a dynamic value. There's a special case for an immutable object that's based on the `tuple` class. We looked at both of these in the *Designing classes with little unique processing* recipe.

There's a middle ground—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

Let's look at the idea of a hand of playing cards in the casino game of *Blackjack*. There are two parts to a hand:

- The cards
- The bet

Both have dynamic values. But there are only these two things. It's common to get more cards. It's also possible to raise the bet via a double down play.

The idea of a split will create additional hands. Each split hand is a separate object, with a distinct collection of cards and a unique bet.

How to do it...

We'll leverage the `__slots__` special name when creating the class:

1. Define the class with a descriptive name:

```
class Hand:
```

2. Define the list of attribute names:

```
    __slots__ = ('hand', 'bet')
```

   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.

3. Add an initialization method:

```
def __init__(self, bet, hand=None):
    self.hand= hand or []
    self.bet= bet
```

   Generally, each hand starts as a bet. The dealer then deals two initial cards to the hand. Under some circumstances, though, we might want to rebuild a `Hand` object from a sequence of `Card` instances.We've used a feature of the `or` operator. If the left side operand is not a false-like value (that is, `None`,) then that's the value of an `or` expression. If the left side operand is false-like, then the right side operand is evaluated. For more information on why this is necessary, see the *Designing functions with optional parameters* recipe in `Chapter 3`, *Function Definitions*.

4. Add a method to update the collection. We've called it `deal` because it's used to deal a new card to the `Hand`:

```
def deal(self, card):
    self.hand.append(card)
```

5. Add a `__repr__()` method so that it can be printed easily:

```
def __repr__(self):
    return "{class_}({bet}, {hand})".format(
        class_= self.__class__.__name__,
        **vars(self)
    )
```

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 *Designing classes with little unique processing* recipe:

```
>>> from ch06_r04 import Card, Hand
>>> h1 = Hand(2)
>>> h1.deal(Card(rank=4, suit='♠'))
>>> h1.deal(Card(rank=8, suit='♡'))
>>> h1
Hand(2, [Card(rank=4, suit='♠'), Card(rank=8, suit='♡')])
```

We've imported the `Card` and `Hand` class definitions. We built an instance of a `Hand`, `h1`, with a bet of twice the table minimum. We then added two cards to the hand via the `deal()` method of the `Hand` class. This shows how the `h1.hand` value can be mutated.

This example also displays the instance of `h1` to show the bet and the sequence of cards. The `__repr__()` method produces output that's in Python syntax.

We can also replace the `h1.bet` value when the player doubles down (yes, this is a crazy thing to do when showing 12):

```
>>> h1.bet *= 2
>>> h1
Hand(4, [Card(rank=4, suit='♠'), Card(rank=8, suit='♡')])
```

When we displayed the `Hand` object, `h1`, it showed that the `bet` attribute was changed.

Here's what happens when we try to create a new attribute:

```
>>> h1.some_other_attribute = True
Traceback (most recent call last):
  File
"/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/doctest.py
", line 1318, in __run
    compileflags, 1), test.globs)
  File "<doctest default[0]>", line 1, in <module>
    h1.some_other_attribute = True
AttributeError: 'Hand' object has no attribute 'some_other_attribute'
```

We attempted to create an attribute named `some_other_attribute` on the `Hand` object, `h1`. This raised an `AttributeError` exception. Using `__slots__` means that new attributes cannot be added to the object.

How it works...

When we create a class definition, the behavior is defined in part by the object class and the `type()` function. Implicitly, a class is assigned a special `__new__()` method that handles the internal house-keeping required to create a new object.

Python has three essential paths:

- The default behavior, which builds a `__dict__` attribute in each object. Because the object's attributes are kept in a dictionary, we can add, change, and delete attributes freely. This flexibility requires the use of a relatively large amount of memory for the dictionary object.
- The `__slots__` behavior, which avoids 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 only the defined attributes. This lack of flexibility means that less memory is used for each object.
- The subclass of `tuple` behavior. These are immutable objects. The easiest way to create these is with `namedtuple()`. Once built, they cannot be changed. When measuring memory use, these are the thriftiest of all classes of objects.

The `__slots__` optimization is used infrequently in Python. The default class behavior provides the most flexibility and makes altering a class easy. In some cases, however, a large application might be constrained by the amount of memory used, and switching just one class to `__slots__` can have a dramatic 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 a rather advanced technique because it exposes some more of 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 the object. Once the empty object has been created, then the class `__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 namespace 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 *Designing classes with little unique processing* recipe.

Using more sophisticated collections

Python has a wide variety of built-in collections. In `Chapter 4`, *Built-in Data Structures – list, set, dict*, 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 fold 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 the overview of the alternative collections. They're in three modules.

The `collections` module contains a number of variations on the built-in collections. These include the following:

- `deque`: A double-ended queue. It's a mutable sequence with optimizations for pushing and popping from each end. Note that the class name starts with a lower-case 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 lower-case letter; this is atypical for Python.
- `Counter`: A mapping that is designed to count occurrences of a key. This is sometimes called a multiset or a bag.

- `OrderedDict`: A mapping that retains the order in which keys where created.
- `ChainMap`: A mapping that combines several dictionaries into a single mapping.

The `heapq` module includes a priority queue implementation. This is a specialized sequence that maintains 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.

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 common naive approach is for the producer to accumulate items in a list, and then the consumer processes the items from the list. This approach will tend to build a large intermediate data structure. A change in focus can interleave production and consumption, reducing the amount of memory used:

   - 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. This is often used for optimizing work, including graph search algorithms. We can simulate this by using `list.append()`, `list.sort(key=lambda x:x.priority)`, and `list.pop(-1)`. Since this involves a sort after each insert, it's terribly inefficient. Using the `heapq` module is considerably more efficient.

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 a `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 zero. We can also use `defauldict(list)` and `defauldict(set)` to create dictionary-of-list or dictionary-of-set structures.
    - In some cases, we'll need to to provide a different literal value as the default:

```
lookup = defaultdict(lambda:"N/A")
```

    This uses a lambda object to define a very small function that has no name and always returns the string `N/A`. This will create a default item of `N/A` for missing keys.

    The `defaultdict(int)` used to count items is so common that the `Counter` class does exactly this.

3. How do we want to handle the order of keys in a dictionary?
    - Order doesn't matter; we always set and get items by key. This is the behavior of a built-in `dict` class. Key ordering depends on hash randomization, and is, therefore, unpredictable.
    - We want to preserve the insert order as well as rapidly find items using their key. The `OrderedDict` class provides this unique combination of features. It has the same interface as the built-in `dict` class, but preserves the insert order of the keys.
    - We want the keys sorted into their proper order. While a sorted list does this, the lookup time for a given key is quite slow. We can use the bisect module to provide rapid access to items within a sorted list. This requires a three step algorithm:
        1. Build the list, perhaps via `append()` or `extend()`.
        2. Sort the list. `list.sort()` is all we need for this.
        3. Do retrievals from the sorted list, using the `bisect` module.

4. How will we build the dictionary?
    - We have a simple algorithm to create items. In this case, a built-in dict 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.

```
import json
user = json.load('~/app.json')
system = json.load('/etc/app.json')
application = json.load('/opt/app/default.json')
```

5. How can we combine these?

```
from collections import ChainMap
config = ChainMap(user, system, application)
```

The resulting `config` object will do a sequential search through the various dictionaries. It will look in the user, system, and application dictionaries for a given key.

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 in opposition: anything we do to reduce storage use tends to increase processing time, and anything we do to reduce processing time increases storage use.

The time aspect is formalized via a complexity metric. There's considerable analysis of the complexity of an algorithm:

- Operations that are described as $O(1)$ happen in constant time. In this case, the complexity doesn't change with the volume of data. For some collections, the actual overall long-term average is nearly $O(1)$ with minor exceptions. List `append` operations are an example: they're all about the same complexity. Once in a while, though, a behind the scenes memory management operation will add some time.

- Operations that are described as ***O***(n) happen in linear time. The cost grows as the volume of data grows. Finding an item in a list has this complexity. Finding an item in a dictionary is closer to ***O***(1) because it's (nearly) the same low complexity, no matter how large the dictionary is.
- Operations that are ***O***(*n* log *n*) grow more quickly than the volume of data. The `bisect` module includes search algorithms that have this complexity.
- There are even worse cases: some algorithms have a complexity of $O(n^2)$ or even $O(n!)$. We'd like to avoid these through clever design and smarter data structures.

The various data structures reflect unique 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 each event from a log file. If the event is part of the pattern, save just this event. We might use a `defaultdict` with the 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.

The first algorithm, read everything into memory, is often wildly impractical. On a large web server, the logs might involve hundreds of gigabytes, or perhaps even terabytes, of data. This won't 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` package. We will 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, a framework such as Hadoop is required for this kind of processing.

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 **MapReduce** operations applied to the results of previous MapReduce operations.

See also

- See the *Choosing a data structure* recipe in `Chapter 4`, *Built-in Data Structures – list, set, dict*, for a foundational set of decisions for selecting data structures

Extending a 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?

Getting ready

We'll create a sophisticated list that can compute the sums and averages of the items in the list. This will require that our application only puts numbers in the list; otherwise, there will be `ValueError` exceptions.

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):
```

   This shows the syntax for defining an extension to a built-in class. If we provide a body that consists only of the `pass` statement, then the new `StatsList` class can be used anywhere the `list` class is used.

   When we write this, the `list` class is called the superclass of `StatsList`.

2. Define the additional processing as new methods. The `self` variable will be an object that has inherited all of the attributes and methods from the superclass. Here's a `sum()` method:

```
def sum(self):
    return sum(v for v in self)
```

   We've used a generator expression to make it perfectly clear that the `sum()` function is applied to every item in the list. Using a generator expression allows us to do calculations or introduce filters very easily.

3. Here's another method that we often apply to a list. This counts items:

```
def count(self):
    return sum(1 for v in self)
```

   This will count the items in the list. Rather than use the `len()` function, we opted to use a generator expression in case we want to add filtering features in the future.

4. Here's the `mean` function:

```
def mean(self):
    return self.sum() / self.count()
```

5. Here are some additional methods:

```
        def sum2(self):
            return sum(v**2 for v in self)
        def variance(self):
            return (self.sum2() -
self.sum()**2/self.count())/(self.count()-1)
        def stddev(self):
            return math.sqrt(self.variance())
```

   The `sum2()` method computes the sum of the 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.

The `StatsList` object inherits all the features of a `list` object. It is extended by the methods that we added. Here's an example of using this collection:

```
>>> from ch06_r06 import StatsList
>>> subset1 = StatsList([10, 8, 13, 9, 11])
>>> data = StatsList([14, 6, 4, 12, 7, 5])
>>> data.extend(subset1)
```

We've created two `StatsList` objects from a literal list of objects. We used the `extend()` method 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 which we defined on this object:

```
>>> data.mean()
9.0
>>> data.variance()
11.0
```

We've displayed the results of the `mean()` and `variance()` methods. Of course, all the features of the built-in `list` class are all present in our extension:

```
>>> data.sort()
>>> data[len(data)//2]
9
```

We used the built-in `sort()` method and used the index feature to extract an item from the list. Because there are an odd number of values, this is the median value. Note that this mutates the `list` object, changing the order of the items. This isn't the best possible implementation for this algorithm.

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 which created a subclass that's a specialized kind of list.

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 the S.O.L.I.D. design principles. However, there will always be cases where it's clearly appropriate to extend a built-in collection.

There's more...

The idea of generalization can lead to superclasses that are abstractions. An abstract class is incomplete, and 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`, *Built-in Data Structures – list, set, dict*, there are abstract superclasses for all of the built-in collections. Rather than start 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 Mapping
class MyFancyMapping(Mapping):
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...

Let's say we've collected data using a `Counter` object. For more information on the various collections, see `Chapter 4`, *Built-in Data Structures – list, set, dict*, specifically the *Using set methods and operators* and *Avoiding mutable default values for function parameters* recipes. In this case, the customers fall into eight categories with approximately equal numbers.

The data looks like this:

```
Counter({15: 7, 17: 5, 20: 4, 16: 3, ... etc., 45: 1})
```

In this collection, each key is the number of visits needed to get a full set of coupons. The values are the numbers of times that the visits occurred. In the preceding data that we saw, there were seven occasions where `15` visits were needed to get a full set of coupons. We can see from the sample data that there were five occasions where `17` visits were needed. This has a long tail. At only one point, there were `45` individual visits before a full set of eight coupons was collected.

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 collection – a list that does statistics* recipe, and we will cover this in `Chapter 7`, *More Advanced Class Design*.
- **Wrap**: We can wrap the `Counter` object in another class that provides just the features we need. We'll look at this in `Chapter 7`, *More Advanced Class Design*.

  A common variation on wrapping uses a statistical computation object with a separate data collection object. This variation on wrapping often leads to an elegant solution.

No matter which class architecture we choose, we have two ways to design the processing:

- **Eager**: This means that 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 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.

The mean, μ, is this:

$$\mu = \sum_{k \in C} f_k \times k$$

Here, k is the key from the `Counter`, C, and f_k is the frequency value for the given key from the `Counter`.

The standard deviation, σ, depends on the mean, μ. The formula is this:

$$\sigma = \sqrt{\frac{\sum_{k \in C} f_k \times (k - \mu)^2}{C + 1}}$$

Here, k is the key from the `Counter`, C, and f_k is the frequency value for the given key from the `Counter`. The total number of items in the counter is $C = \sum_{k \in C} f_k$.

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:

```
def __init__(self, raw_counter:Counter):
    self.raw_counter = raw_counter
```

   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.

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):
    return sum(f*v for v, f in self.raw_counter.items())
@property
def count(self):
    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):
    return self.sum / self.count
```

5. Here's how we can calculate the standard deviation:

```
@property
def sum2(self):
    return sum(f*v**2 for v, f in self.raw_counter.items())
@property
def variance(self):
    return (self.sum2 - self.sum**2/self.count)/(self.count-1)
@property
def stddev(self):
    return math.sqrt(self.variance)
```

Note that we've been using `math.sqrt()`. Be sure to add the required `import math` statement in the Python file.

6. Here's how we can create some sample data:

```
>>> from ch04_r06 import *
>>> from collections import Counter
>>> def raw_data(n=8, limit=1000, arrival_function=arrival1):
...     expected_time = float(expected(n))
...     data = samples(limit, arrival_function(n))
...     wait_times = Counter(coupon_collector(n, data))
...     return wait_times
```

We've imported functions such as `expected()`, `arrival1()`, and `coupon_collector()` from the `ch04_r06` module. We've imported the `Counter` collection from the standard library `collections` module.

We defined a function, `raw_data()`, that will generate a number of customer visits. By default, it will be 1,000 visits. The domain will be eight different classes of customers; each class will have an equal number of members. We'll use the `coupon_collector()` function to step through the data, emitting the number of visits required to collect a full set of eight coupons.

This data is then used to assemble a `Counter` object. This will have the number of customers required to get a full set of coupons. Each number of customers will also have a frequency showing how often that number of visits occurred.

7. Here's how we can analyze the `Counter` object:

```
>>> import random
>>> from ch06_r07 import LazyCounterStatistics
>>> random.seed(1)
>>> data = raw_data()
>>> stats = LazyCounterStatistics(data)
>>> print("Mean: {0:.2f}".format(stats.mean))
Mean: 20.81
>>> print("Standard Deviation: {0:.3f}".format(stats.stddev))
Standard Deviation: 7.025
```

First, we imported the `random` module so that we could pick a known `seed` value. This makes it easier to test and demonstrate an application because the random numbers are consistent. We also imported the `LazyCounterStatistics` class from the `ch06_r07` module.

Once we have all of the items defined, we can force the seed to a known value, and generate the coupon collector test results. The `raw_data()` function will emit a `Counter` object, which we called `data`.

We'll use the `Counter` object to create an instance of the `LazyCounterStatistics` class. We'll assign this to 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.

For a set of eight coupons, the theoretical average is 21.7 visits to collect all coupons. It looks like the results from `raw_data()` show behavior that matches the expectation of random visits. This is sometimes called the null hypothesis—the data is random.

In this case, the data really was random. We've validated our approach. We can now use this software on real-world data with some confidence that it behaves correctly.

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.

This shows that being naive about a lazy design may not be optimal in some cases. This is an easy problem to fix, in general. We can always create additional local variables to save intermediate results.

To make this class look like the class that performs eager calculations, we used the `@property` decorator. This makes a method function appear to be an attribute. This can only work for method functions 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 value is used rarely, 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 recomputation that's done. 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:

```
def __init__(self, raw_counter:Counter):
    self.raw_counter = raw_counter
    self._count = None
@property
def count(self):
    if self._count is None:
        self._count = sum(f for v, f in self.raw_counter.items())
    return self._count
```

This technique uses an attribute to save a copy of the count calculation. This value can be computed once and returned as often as needed with no cost for recalculation.

This optimization is only 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 recreate the `LazyCounterStatistics` every time the `Counter` was 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.

Using settable properties to update eager attributes

In several of the previous recipes, we've looked at the important distinction between eager and lazy computation. See the *Designing classes with lots of processing* recipe for an example of eagerly computing a result and setting object attributes. See the *Using properties for lazy attributes* recipe for a way to use properties to lazily compute a result.

When an object is stateful, then attribute values must be changed throughout the object's life. It's common to use methods to eagerly compute attribute changes, but this isn't really necessary.

We have the following choices for stateful objects:

- Set attribute values via methods:
    - Compute results eagerly, putting results in attributes
    - Compute results lazily, using properties that have syntax that looks like a simple attribute
- Set values via attributes:
    - If results are computed lazily via properties, then the new state can be reflected in these calculations

What can we do if we want to use attribute-like syntax for setting a value, but we also want to perform eager calculations?

This gives us another variation: we can use a property setter to have attribute-like syntax. This method can also perform eager calculations of the results.

For example, we'll use a fairly complex looking object that has several attributes that are derived from other attributes. How can we eagerly compute values from attribute changes?

Getting ready

Consider a class that represents a leg of a voyage. It has three principle features—rate, time, and distance. Looking at this in general, it's possible to eagerly compute any one value from a change in the other two.

We can add features to make this quite a bit more complex. For example, if the distance is computed from latitude and longitude, the general approach has to be modified somewhat. If we're using specific points instead of a more flexible distance, then a distance calculation may involve something like rate, time, starting point, and bearing. This involves two interlocked calculations. We won't go quite so far in this example; we'll stick to a simpler rate-time-distance calculation.

Since two attributes must be set to compute the third, the object will have a fairly complex set of internal states:

- No attributes have been set: everything is unknown.
- One item has been set: nothing can be computed yet.
- Two distinct items have been set: now the third can be computed.

After this, it's ideal to support additional attribute changes. The essential rule is to compute appropriate new values based on the most recent two distinct changes:

- If rate, *r*, and time, *t*, are the last two things that were changed, compute the distance, *d*. Use $d = r * t$.
- If rate, *r*, and distance, *d*, are the last two things that were changed, compute the time, *t*. Use $t = d/r$.
- If time, *t*, and distance, *d*, are the last two things that were changed, compute the rate, *r*. Use $r = d/t$.

We'd like the object to behave like this:

```
    leg_1 = Leg()
    leg_1.rate = 6.0 # knots
    leg_1.distance = 35.6 # nautical miles
    print("Cover {leg.distance:.1f}nm at {leg.rate:.2f}kt =
{leg.time:.2f}hr".
        format(leg=leg_1))
```

This has the distinct advantage of offering a very simple interface to the `leg` object. An application merely sets any two attributes and the calculation is performed eagerly to provide a value for the remaining attribute.

How to do it...

We'll break this into two parts. First, the general overview of defining settable properties, then the details of how to track state changes:

1. Define a class with a meaningful name.

2. Provide hidden attributes. These will be exposed as properties:

```
class Leg:
def __init__(self):
    self._rate= rate
    self._time= time
    self._distance= distance.
```

3. For each gettable property, provide a method to compute the property value. In many cases, these will parallel the hidden attributes:

```
@property
def rate(self):
    return self._rate
```

4. For each settable property, provide a method to set the property value:

```
@rate.setter
def rate(self, value):
    self._rate = value
    self._calculate('rate')
```

The setter method has a special property decorator based on the getter method name. In this example, the `@property` decorator on the `rate()` method also creates a `rate.setter` decorator that can be used to define the setter method for this attribute.

Note that the method names for the getter and setter are identical. The `@property` and `@rate.setter` decorations distinguish the two methods from each other.

In this example, we've saved the value into the hidden attribute, `self._rate`. Then, the `_calculate()` method is used to eagerly calculate all of the hidden attributes, if possible.

5. This can be repeated for all other properties. In our case, the code for time and distance are similar:

```
@property
def time(self):
    return self._time
@time.setter
def time(self, value):
    self._time = value
    self._calculate('time')
@property
def distance(self):
    return self._distance
@distance.setter
def distance(self, value):
    self._distance = value
    self._calculate('distance')
```

The details of tracking the state change rely on a feature of the `collections.deque` class. The rule for calculation can be implemented as a two-element bounded queue of distinct changes. As each distinct field is changed, we can enqueue the field name. The two distinct names in the queue are the last two fields changed; the third can be determined from this by set subtraction:

1. Import the `deque` class:

```
from collections import deque
```

2. Initialize the queue in the `__init__()` method:

```
self._changes= deque(maxlen=2)
```

3. Enqueue each distinct change. Determine what's missing from the queue, and compute this:

```
def _calculate(self, change):
if change not in self._changes:
    self._changes.append(change)
compute = {'rate', 'time', 'distance'} - set(self._changes)
if compute == {'distance'}:
    self._distance = self._time * self._rate
elif compute == {'time'}:
    self._time = self._distance / self._rate
elif compute == {'rate'}:
    self._rate = self._distance / self._time
```

If the latest change is not already in the queue, it's appended. Since the queue has a bounded size, the oldest item, the one least recently changed, is silently popped to keep the queue size fixed.

The difference between the set of available properties, and the set of properties recently changed is a single property name. This is the name least recently set; the value for this can be computed from the other two that were set more recently.

How it works...

This works because Python implements a property with a kind of class called a **Descriptor**. A descriptor class can have methods for getting a value, setting a value, and deleting a value. Depending on the context, one of these methods is used implicitly:

- When a descriptor object is used in an expression, the `__get__` method is used
- When a descriptor is on the left side of an assignment statement, the `__set__` method is used
- When a descriptor appears in a `del` statement, the `__delete__` method is used

The `@property` decorator does three things:

- Modifies the following method to be wrapped up in a descriptor object. The method that follows is modified to be the descriptor's `__get__` method. It will compute values when used in an expression.
- Adds a `method.setter` decorator. This decorator will modify the method that follows to be the descriptor's `__set__` method. When the name is used on the left side of an assignment statement, the given method is executed.
- Adds a `method.deleter` decorator. This decorator will modify the method that follows to be the descriptor's `__delete__` method. When the name is used in a `del` statement, the given method is executed.

This allows the building of an attribute name that can be used to provide values, set values, and even delete values.

There's more...

There are a few more refinements we could make to this class. We'll look at two more advanced techniques for initialization and calculation.

Initialization

We can provide a way to properly initialize an instance with some values. This change makes it possible to do the following:

```
>>> from ch06_r08 import Leg
>>> leg_2 = Leg(distance=38.2, time=7)
>>> round(leg_2.rate, 2)
5.46
>>> leg_2.time=6.5
>>> round(leg_2.rate, 2)
5.88
```

The example shows how this helps in planning a voyage by a sailboat. If the distance to cover is `38.2` nautical miles, and the goal is to finish in `7` hours, the boat must reach a speed of `5.46` knots. To shave a half hour off the trip requires a speed of `5.88` knots.

For this to work, the `__init__()` method needs to be changed. The internal `dequeue` object must be built right away. As each attribute is set, the internal `_calculate()` method must be used to track the setting:

```
class Leg:
    def __init__(self, rate=None, time=None, distance=None):
        self._changes= deque(maxlen=2)
        self._rate= rate
        if rate: self._calculate('rate')
        self._time= time
        if time: self._calculate('time')
        self._distance= distance
        if distance: self._calculate('distance')
```

The `dequeue` function is created first. As each individual field value is set, the change is logged in the queue of changed attributes. If two fields are set, the third will be computed.

If all three fields are set, then the last two changes—time and distance, in this case—will compute a value for `rate`. This will overwrite the provided value.

Calculation

Currently, the various calculations are buried inside an `if` statement. This makes changes difficult because a subclass would be forced to supply the entire method rather than simply supplying a calculation change.

We can remove the `if` statement using an introspection technique. The overall design would be better with explicit calculation methods:

```
def calc_distance(self):
    self._distance = self._time * self._rate
def calc_time(self):
    self._time = self._distance / self._rate
def calc_rate(self):
    self._rate = self._distance / self._time
```

The following version of `_calculate()` makes use of these methods:

```
def _calculate(self, change):
    if change not in self._changes:
        self._changes.append(change)
    compute = {'rate', 'time', 'distance'} - set(self._changes)
    if len(compute) == 1:
        name = compute.pop()
        method = getattr(self, 'calc_'+name)
        method()
```

When the value of compute is a singleton set, using the `pop()` method extracts that one value from the set. Prepending `calc_` to this string gives the name of a method that will compute the desired value.

The `getattr()` function does a lookup to find the requested method of the object, `self`. This is then evaluated as a bound function. It can update attributes with the desired result.

Refactoring the calculations into separate methods makes the class more open to extension. We can now create a subclass that includes revised calculations, but preserves the overall features of the class.

See also

- For more information on working with sets, see the *Using set methods and operators* recipe in `Chapter 4`, *Built-in Data Structures – list, set, dict*.
- A `dequeue`, effectively, is a list that's highly optimized for append and pop operations. See the *Deleting from a list – deleting, removing, popping, and filtering* recipe in `Chapter 4`, *Built-in Data Structures – list, set, dict*.

7
More Advanced Class Design

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

- Choosing between inheritance and extension – 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
- Defining an ordered collection
- Deleting from a list of mappings

Introduction

In `Chapter 6`, *Basics of Classes and Objects*, we looked at some recipes that cover the basics of class design. In this chapter, we'll dive a little more deeply into Python classes.

In the *Designing classes with lots of processing* and *Using properties for lazy attributes* recipes in `Chapter 6`, *Basics of Classes and Objects*, we identified a design choice that's central to object-oriented programming, the wrap versus extend choice. It's possible to add features to a class via extension and it's also possible to create a new class that wraps an existing class to add new features. There are a number of extension techniques available in Python, providing a lot of alternatives.

A Python class can inherit features from more than one superclass. This can lead to confusion, but a simple design pattern, the **mixin**, can prevent problems.

A larger application may require some global data that's widely shared by many classes or modules. This can be challenging to manage. We can, however, use a module to manage a global object and create a simple solution.

In `Chapter 4`, *Built-in Data Structures – list, set, dict*, we looked at the core built-in data structures. It's time to combine some features to create more sophisticated objects. This can also include extending built-in data structures to add sophistication.

Choosing between inheritance and extension – the is-a question

In the *Using cmd for creating command-line applications* recipe in `Chapter 5`, *User Inputs and Outputs*, and the *Extending a collection – a list that does statistics* recipe in `Chapter 6`, *Basics of Classes and Objects*, we looked at extending a class. In both cases, our class was a subclass of a built-in class.

The idea of extension is sometimes called the generalization-specialization relationship. It's sometimes also called the **is-a relationship**.

There's an important semantic issue here that we can also summarize as the **wrap versus extend problem**:

- Do we really mean that the subclass is an example of the superclass? This is the is-a relationship. An example in Python is the built-in `Counter`, which extends the base class `dict`.
- Or do we mean something else? Perhaps there's an association, sometimes called the **has-a relationship**. An example of this is in the *Designing classes with lots of processing* recipe in `Chapter 6`, *Basics of Classes and Objects*, where `CounterStatistics` wraps a `Counter` object.

What's a good way to distinguish between these two techniques?

Getting ready

The question is a bit of metaphysical philosophy, specifically focused on the ideas of an **ontology**. An ontology is a way to define categories of being.

When we extend an object, we have to ask the following:

> *"Is this a new class of objects, or a mixture of existing classes of objects?"*

We'll look at two ways to model a deck of playing cards:

- As a new class of objects that extends the built-in `list` class
- As a wrapper that combines the built-in `list` class with some other features

A deck is a collection of cards. The core ingredient, then, is the underlying `Card` object. We'll define this very simply using `namedtuple()`:

```
>>> from collections import namedtuple
>>> Card = namedtuple('Card', ('rank', 'suit'))
>>> SUITS = '\u2660\u2661\u2662\u2663'
>>> Spades, Hearts, Diamonds, Clubs = SUITS
>>> Card(2, Spades)
Card(rank=2, suit='♠')
```

We've created the class definition, `Card`, using `namedtuple()`. This creates a simple class with two attributes—`rank` and `suit`.

We also defined the various suits, `SUITS`, as a string of Unicode characters. To make it easier, to create cards of a specific suit, we also decomposed the string into four individual one character substrings. If your interactive environment doesn't properly display Unicode characters, you may have trouble with this. It might be necessary to change the OS environment variable, `PYTHONIOENCODING`, to `UTF-8`, so that proper encoding is done.

The `\u2660` string is a single Unicode character. You can confirm this with `len(SUITS) == 4`. If the length isn't 4, check for extraneous spaces.

We'll use this `Card` class in the rest of this recipe. In some card games, a single 52-card deck is used. In other games, a dealing shoe is used. A shoe is a box that allows a dealer to shuffle together multiple decks and deal conveniently.

What's important is that the various kinds of collection—deck, shoe, and the built-in list all have considerable overlaps in the kinds of features they support. Are they all more or less related? Or are they fundamentally distinct?

How to do it...

We'll wrap the *Using a class to encapsulate data and processing* recipe in `Chapter 6`, *Basics of Classes and Objects*, with this recipe:

1. Use the nouns and verbs from the original story or problem statement to identify all of the classes.
2. Look for overlaps in the feature sets of various classes. In many cases, the relationships will come directly from the problem statement itself. In our preceding example, a game can deal cards from a deck, or deal cards from a shoe. In this case, we might state one of these two views:
    - A shoe is a specialized deck that starts with multiple copies of the 52-card domain
    - A deck is a specialized shoe with only one copy of the 52-card domain
3. Create a small ontology that clarifies the relationships among the classes. There are several kinds of relationships.

   Some classes are independent of each other. They're linked for the purposes of implementing a user story. In our example a `Card` refers to a string for the suit. The two objects are independent of each other. Many cards will share a common suit string. These are ordinary references between objects and there are no special design considerations:

    - **Aggregation**: Some objects are bound into collections, but the objects have a properly independent existence. Our `Card` objects might be aggregated into a `Hand` collection. When the game ends, the `Hand` objects can be deleted, but the `Card` objects continue to exist. We might create a `Deck` that refers to a built-in `list`.
    - **Composition**: Some objects are bound into collections, but do not have an independent existence. When looking at card games, a `Hand` of cards cannot exist without a `Player`. We might say that a `Player` object is composed—in part—of a `Hand`. If a `Player` is eliminated from a game, then the `Hand` objects must also be removed. While this is important for understanding the relationships among objects, there are some practical considerations that we'll look at in the next section.
    - **Is-a or inheritance**: This is the idea that a `Shoe` is a `Deck` with an extra feature (or two). This may be central to our design. We'll look into this in detail in the *Extending – inheritance* section of this recipe.

We've identified several paths for implementing the associations. The aggregation and composition cases are both wrap techniques. The inheritance case is the extend technique. We'll look at aggregation and composition—both wrapping techniques and extending techniques—separately.

Wrapping – aggregation and composition

Wrapping is a way to understand a collection. It can be a class that is an aggregate of independent objects. It's also a composition that wraps an existing list, meaning that the underlying `Card` objects will be shared by a `list` collection and a `Deck` collection.

1. Define the independent collection. It might be a built-in collection, for example, a `set`, `list`, or `dict`. For this example, it will be a list that contains the cards:

```
domain = [Card(r+1,s) for r in range(13) for s in SUITS]
```

2. Define the aggregate class. For this example, the name has a `_W` suffix. This is not a recommended practice; it's only used here to make the distinctions between class definitions more clear. Later, we'll see a slightly different variation on this design:

```
class Deck_W:
```

3. 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]):
    self.cards = cards.copy()
    self.deal_iter = iter(cards)
```

   This uses a type hint, `List[Card]`. The `typing` module provides the necessary definition of `List`.

4. If needed, provide other methods to either replace the collection, or update the collection. This is rare in Python, since the underlying attribute, `cards`, can be accessed directly. However, it might be helpful to provide a method that replaces the `self.cards` value.

5. Provide the methods appropriate to the aggregate object:

```
def shuffle(self):
    random.shuffle(self.cards)
    self.deal_iter = iter(self.cards)
def deal(self) -> Card:
    return next(self.deal_iter)
```

The `shuffle()` method randomizes the internal list object, `self.cards`. The `deal()` object creates an iterator that can be used to step through the `self.cards` list. We've provided a type hint on `deal()` to clarify that it returns a `Card` instance.

Here's how we can use this class. We'll be sharing 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 SUITS)
>>> 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 the list of objects in the `domain` variable:

```
>>> import random
>>> from ch07_r01 import Deck_W
>>> d = Deck_W(domain)
```

Once the `Deck_W` object is available, it's possible to use the unique features:

```
>>> 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='♢')]
```

We've seeded the random number generator to force the cards to have a defined order. That makes unit testing possible. After that, we shuffled the deck into an order based on the random seed. Once the seed is sown, the results are consistent, making unit testing easy. We can deal five cards from the deck. This shows how the `Deck_W` object, `d`, shares the same pool of objects as the `domain` list.

We can delete the `Deck_W` object, `d`, and create a new deck from the `domain` list. This is because the `Card` objects are not part of a composition. The cards have an independent existence from the `Deck_W` collection.

Extending - inheritance

Here's an approach to defining a class that extends a collection of objects. We'll define a `Deck` as an aggregate that wraps an existing list. The underlying `Card` objects will be shared by a list and a `Deck`:

1. Define the extension class as a subclass of a built-in collection. For this example, the name has a `_X` suffix. This not a recommended practice; it's only used here to make the distinctions between two class definitions in this recipe more clear:

   ```
   class Deck_X(list):
   ```

   This is a clear and formal statement—a `Deck` is a list.

2. Use the `__init__()` method inherited from the `list` class. No code is needed.
3. Use other methods of the `list` class for adding, changing, or removing items from the `Deck`. No code is needed.
4. Provide the appropriate methods to the extended object:

   ```
   def shuffle(self):
       random.shuffle(self)
       self.deal_iter = iter(self)
   def deal(self) -> Card:
       return next(self.deal_iter)
   ```

   The `shuffle()` method randomizes the object as a whole, `self`, because it is an extension of the list. The `deal()` object creates an iterator that can be used to step through the `self.cards` list. We've provided a type hint on `deal()` to clarify that it returns a `Card` instance.

Here's how we can use this class. First, we'll build a deck of cards:

```
>>> from ch07_r01 import Deck_X
>>> d2 = Deck_X(Card(r+1,s) for r in range(13) for s in SUITS)
>>> len(d2)
52
```

We used a generator expression to build individual `Card` objects. We can use the `Deck_X()` class function exactly the way we'd use the `list()` class function. In this case, we built a `Deck_X` object from the generator expression. We could build a `list` similarly.

We did not provide an implementation for the built-in `__len__()` method. This was inherited from the `list` class, and works nicely.

Using the deck-specific features for this implementation looks exactly like the other implementation, `Deck_W`:

```
>>> random.seed(1)
>>> d2.shuffle()
>>> [d2.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='♢')]
```

We've seeded the random number generator, shuffled the deck, and dealt five cards. The extension methods work as well for `Deck_X` as they do for `Deck_W`. The `shuffle()` and `deal()` methods do their jobs.

How it works...

Python's mechanism for finding a method (or attribute) works like this:

1. Search in the class for the method or attribute.
2. If the name is not defined in the immediate class, then search in all of the parent classes for the method or attribute.

This is how Python implements the idea of inheritance. Searching through the parent classes assures two things:

- Any method defined in any superclass is available to all subclasses
- Any subclass can override a method to replace a superclass method

Because of this, a subclass of the `list` class inherits all the features of the parent class. It is a specialized variation 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. The word `private` is used by languages such as C++ and Java. Python doesn't have this, a subclass can override any method.

To explicitly refer to methods from a superclass, we can use the `super()` function to force a search through the superclasses. This allows a subclass to add features by wrapping the superclass version of a method. We use it like this:

```
def some_method(self):
    # do something extra
    super().some_method()
```

In this case, the `some_method()` object will do something extra and then do 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...

When designing a class, we must choose between the several essential techniques:

- **Wrapping**: This technique creates a new class. All of the required methods must be defined. This can be a lot of code to provide the required methods. Wrapping can be decomposed into two broad implementation choices:
    - **Aggregation**: The objects being wrapped have an independent existence from the wrapper. The `Deck_W` example showed how the `Card` objects and even the list of cards were independent from the class. When any `Deck_W` object is deleted, the underlying list will continue to exist.
    - **Composition**: The objects being wrapped don't have any independent existence; they're an essential part of the composition. This involves a subtle difficulty because of Python's reference counting. We'll look at this shortly in some detail.
- **Extension via inheritance**: This is the is-a relationship. When extending a built-in class, then a great many methods are available from the superclass. The `Deck_X` example showed this technique by creating a deck as an extension to the built-in `list` class.

When looking at independent existence of objects, there's an important consideration. We don't really remove objects from memory. Instead, Python uses a technique called reference counting to track how many times an object is used. A statement such as `del deck` doesn't really remove the `deck` object, it removes the `deck` variable, which decrements the reference count for the underlying object. If the reference count is zero, the object is not used and can be removed.

Consider the following example:

```
>>> c_2s = Card(2, Spades)
>>> c_2s
Card(rank=2, suit='♣')
>>> another = c_2s
>>> another
Card(rank=2, suit='♣')
```

At this point, we have an object, `Card(2, Spades)`, and two variables that refer to the object—`c_2s` and `another`.

If we remove one of those variables with the `del` statement, the other variable still has a reference to the underlying object. The object can't be removed from memory until both variables are removed.

This consideration makes the distinction between aggregation and composition mostly irrelevant for Python programmers. In languages that don't use automatic garbage collection or reference counters, then composition becomes important because objects may vanish. In Python, objects can't vanish unexpectedly. We generally focus on aggregation because removal of unused objects is entirely automatic.

See also

- We've looked at the built-in collections in `Chapter 4`, *Built-in Data Structures – list, set, dict*. Also, in `Chapter 6`, *Basics of Classes and Objects*, 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 6`, *Basics of Classes and Objects*, where we put the complex computations into the class as properties; this design relies on extension.

Separating concerns via multiple inheritance

In the *Choosing between inheritance and extension – the is-a question* recipe,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 a rank and a suit. This created a number of 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, such as, *Blackjack* and *Cribbage* assign a point value to each rank. Generally, the face cards have 10 points. For Blackjack, an Ace has two different point values; depending on the total of other cards in the hand, it can be worth one point or ten points.

How can we handle all of the variations in card game rules?

Getting ready

The `Card` class is really a mixture of two feature sets:

1. Some essential features, such as rank and suit.
2. Some game-specific features, such as the number of points. For a game such as *Cribbage*, the points are consistent regardless of any context. For *Blackjack*, however, there's a relationship between a `Hand` and the `Card` objects within a `Hand`.

Python lets us define a class that has multiple parents. A class can have both a `Card` superclass and a `GameRules` superclass.

In order to make sense of this kind of design, we'll often partition the various class hierarchies into two sets of features:

- **Essential features**: This includes `rank` and `suit`
- **Mixin features**: These features are mixed into the class definition

The idea is that a working class definition will have both essential and mixin features.

How to do it...

1. Define the essential class:

```
class Card:
    __slots__ = ('rank', 'suit')
    def __init__(self, rank, suit):
        super().__init__()
        self.rank = rank
        self.suit = suit
    def __repr__(self):
        return "{rank:2d} {suit}".format(
            rank=self.rank, suit=self.suit
        )
```

We've defined a generic `Card` class that is suitable for ranks two to ten. We've included an explicit call to any superclass initialization via `super().__init__()`.

2. Define any subclasses to handle specializations:

```
class AceCard(Card):
    def __repr__(self):
        return " A {suit}".format(
            rank=self.rank, suit=self.suit
        )
class FaceCard(Card):
    def __repr__(self):
        names = {11: 'J', 12: 'Q', 13: 'K'}
        return " {name} {suit}".format(
            rank=self.rank, suit=self.suit,
            name=names[self.rank]
        )
```

We've defined two subclasses of the `Card` class. The `AceCard` class handles the special formatting rules for an Ace. The `FaceCard` class handles other formatting rules for Jack, Queen, and King.

3. Define a mixin superclass that identifies the additional features that will be added. In some cases, the mixins will all inherit features from a common abstract class. In this example, we'll use a concrete class that handles the rules for Ace through 10:

```
class CribbagePoints:
    def points(self):
        return self.rank
```

For the game of *Cribbage*, the points for most cards are equal to the rank of the card.

4. Define concrete mixin subclasses for the various kinds of features:

```
class CribbageFacePoints(CribbagePoints):
    def points(self):
        return 10
```

For the three ranks of face cards, the points are always 10.

5. Create the class definitions that combine the essential classes and the mixin classes. While it's technically possible to add unique method definitions here, that often leads to confusion. The goal is to have two separate sets of features that are simply merged to create the resulting class definition.

```
class CribbageAce(AceCard, CribbagePoints):
    pass

class CribbageCard(Card, CribbagePoints):
    pass

class CribbageFace(FaceCard, CribbageFacePoints):
    pass
```

6. Create a factory function (or factory class) to create the appropriate objects based the on input parameters:

```
def make_card(rank, suit):
    if rank == 1: return CribbageAce(rank, suit)
    if 2 <= rank < 11: return CribbageCard(rank, suit)
    if 11 <= rank: return CribbageFace(rank, suit)
```

7. We can use this function to create a deck of cards like this:

```
>>> from ch07_r02 import make_card, SUITS
>>> import random
>>> random.seed(1)
>>> deck = [make_card(rank+1, suit) for rank in range(13) for suit in
SUITS]
>>> random.shuffle(deck)
>>> len(deck)
52
>>> deck[:5]
[ K ♡,  3 ♡, 10 ♡,  6 ♢,  A ♢]
```

We've seeded the random number generator to assure that the results are the same each time we evaluate the `shuffle()` function. This makes unit testing possible.

We use a list comprehension to generate a list of cards that include all 13 ranks and four suits. This is a collection of 52 individual objects. The objects belong to two class hierarchies. Each object is a subclass of `Card` as well as being a subclass of `CribbagePoints`. This means that both collections of features are available for all of the objects.

For example, 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 three, six, and Ace, so the total points are `30`.

How it works...

Python's mechanism for finding a method (or attribute) works like this:

1. Search in the class for the method or attribute.
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 parent classes are searched in a sequence called appropriately, the **Method Resolution Order** (**MRO**).

The method resolution order is computed when the class is created. The algorithm used is called C3. More information is available at `https://en.wikipedia.org/wiki/C3_linearization`. This algorithm assures that each parent class is searched once. It also assures that the relative ordering of superclasses is preserved so that all subclasses tend to be searched before any of their parent classes.

We can see the method resolution order using the `mro()` method of a class. Here's an example:

```
>>> c = deck[5]
>>> c
10 ♢
>>> c.__class__.__name__
'CribbageCard'
>>> c.__class__.mro()
[<class 'ch07_r02.CribbageCard'>, <class 'ch07_r02.Card'>,
<class 'ch07_r02.CribbagePoints'>, <class 'object'>]
```

We picked a card from the deck, `c`. The card's `__class__` attribute is a reference to the class. In this case, the class name is `CribbageCard`. The `mro()` method of this class shows us the order that's used to resolve names:

1. First search the class itself, `CribbageCard`.
2. If it's not there, search `Card`.
3. Try to find it in `CribbagePoints` next.
4. Use `object` last.

Class definitions generally use internal `dict` objects to store the method definitions. This means that the search is a hash lookup that is extremely fast. The overhead difference is about 3% more time to search `object` (when something's not found in any of the previous classes) than to search `Card`.

If we do one million operations, we see numbers such as the following:

```
Card.__repr__ 1.4413
object.__str__ 1.4789
```

We compared the time to find `__repr__()`, defined in `Card`, against the time to find `__str__()`, defined in `object`. The extra time summed over a million repetitions is 0.03 seconds.

Since the cost is negligible, this capability is an important way to structure the design of a class hierarchy.

There's more...

There are several kinds of concerns, that we can separate like this:

- **Persistence and representation of state**: We might add methods to manage conversion to a consistent external representation.
- **Security**: This may involve a mixin class that performs a consistent authorization check that becomes part of each object.
- **Logging**: A mixin class that creates a logger that's consistent across a variety of classes might be defined.
- **Event signaling and change notification**: In this case, we might have objects that produce state change notifications and objects that will subscribe to those notifications. These are sometimes called the observable and observer design patterns. A GUI widget might observe the state of an object; when the object changes, it notifies the GUI widget so that the display is refreshed.

As a small example, we could add a mixin to introduce logging. We'll define this class so that it must be provided first in the list of superclasses. Since it's early in the MRO list, the `super()` function will find methods defined later in the list of classes.

This class will add the `logger` attribute to each class:

```
class Logged:
    def __init__(self, *args, **kw):
        self.logger = logging.getLogger(self.__class__.__name__)
        super().__init__(*args, **kw)
    def points(self):
        p = super().points()
        self.logger.debug("points {0}".format(p))
        return p
```

Note that we've used `super().__init__()` to perform the `__init__()` method of any other classes defined in the MRO. As we just noted, it's generally the simplest approach to have one class that defines the essential features of an object, and all other mixins simply add features to that object.

We've provided a definition for `points()`. This will search other classes in the MRO list for an implementation of `points()`. Then it will log the results computed by the method from another class.

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 a 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 superclass list that follows it in the class definition.

To make use of these classes, we'd need to make one more small change to an application:

```
def make_logged_card(rank, suit):
    if rank == 1: return LoggedCribbageAce(rank, suit)
    if 2 <= rank < 11: return LoggedCribbageCard(rank, suit)
    if 11 <= rank: return LoggedCribbageFace(rank, suit)
```

We need to use this function instead of `make_card()`. This function will use the other set of class definitions.

Here's how we use this function to build a deck of card instances:

```
deck = [make_logged_card(rank+1, suit)
    for rank in range(13)
        for suit in SUITS]
```

We've replaced `make_card()` with `make_logged_card()` when creating a deck. Once we do this, we now have detailed debugging information available from a number of classes in a consistent fashion.

See also

- When considering multiple inheritance, it's always essential to also consider whether or not a wrapper is a better design. See the *Choosing between inheritance and extension – the is-a question* recipe.

Leveraging Python's duck typing

Most of the time that a design involves inheritance, there's a clear relationship from a superclass to one or more subclasses. In the *Choosing between inheritance and extension – the is-a question* recipe of this chapter as well as the *Extending a collection – a list that does statistics* recipe in `Chapter 6`, *Basics of Classes and Objects*, we've looked at extensions that involve a proper subclass-superclass relationship.

Python doesn't have a formal mechanism for abstract superclasses. The standard library, however, has the `abc` module that supports the creation of abstract classes.

This isn't always necessary, however. Python relies on duck typing to locate methods within a class. 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."*

The quote is originally from James Whitcomb Riley. It's sometimes taken as a summary of **abductive reasoning**: we go from an observation to a more complete theory that includes that observation. In the case of Python class relationships, if two objects have the same methods, and the same attributes, this has the same effect as having a common superclass. It works even if there's no common superclass definition other than the `object` class.

We can call the collection of methods and attributes the signature of a class. The signature uniquely identifies the class's properties and behaviors. In Python, the signature is dynamic, and the matching is simply a matter of doing a lookup for a name within an object's namespace.

Can we exploit this?

Getting ready

It's often easy to create a superclass and be sure that all subclasses extend this class. In some cases, though, this can be awkward. 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 so that it can be included widely.

Instead, it's sometimes easier to avoid a common superclass and simply check that two classes are both equivalent using the duck test—the two classes have the same methods and attributes, therefore, they are effectively members of some superclass that has no formal realization as Python code.

We'll use a simple pair of classes to show how this works. These classes will both simulate rolling a pair of dice. While the problem is simple, we can easily create a variety of implementations.

How to do it...

1. Define a class 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:

```
class Dice1:
    def __init__(self, seed=None):
        self._rng = random.Random(seed)
        self.roll()
    def roll(self):
        self.dice = (self._rng.randint(1,6),
```

```
                self._rng.randint(1,6))
            return self.dice
```

2. Define other classes that have 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:

```
class Die:
    def __init__(self, rng):
        self._rng= rng
    def roll(self):
        return self._rng.randint(1, 6)
class Dice2:
    def __init__(self, seed=None):
        self._rng = random.Random(seed)
        self._dice = [Die(self._rng) for _ in range(2)]
        self.roll()
    def roll(self):
        self.dice = tuple(d.roll() for d in self._dice)
        return self.dice
```

   This class introduces an additional attribute, `_dice`. This change in implementation doesn't change the advertised interface of a single attribute, `dice`, and method, `roll()`.

At this point, the two classes can be interchanged freely:

```
def roller(dice_class, seed=None, *, samples=10):
    dice = dice_class(seed)
    for _ in range(samples):
        yield dice.roll()
```

We can use this function as follows:

```
>>> from ch07_r03 import roller, Dice1, Dice2
>>> 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.

We can, of course, push the envelope and look for the `_dice` attribute as a way to distinguish between the two classes. We can also use `__class__` to distinguish between the classes.

How it works...

When we write an expression of the form `namespace.name`, Python will look up the name within the given namespace. The algorithm works like this:

1. Search the object's `self.__dict__` collection for the name. Some class definitions will save space, using `__slots__`. See the *Optimizing small objects with __slots__* recipe in `Chapter 6`, *Basics of Classes and Objects*, for more on this optimization. This is generally how attribute values are found.
2. Search the object's `self.__class__.__dict__` collection for the name. This is generally how methods are found.
3. As we noted in the *Choosing between inheritance and extension - the is-a question* and *Separating concerns via multiple inheritance* recipes, the search can continue through all of the superclasses of the class. This search is done in the defined method resolution order.

There are two essential outcomes:

- The value is an object that's not callable. This is the value. This is typical of attributes.
- The value of the attribute is a bound method of a class. This is true for both ordinary methods and properties. See the *Using properties for lazy attributes* recipe in `Chapter 6`, *Basics of Classes and Objects*, for more information on properties. The bound method must be evaluated. For simple methods, the arguments are in `()` after the method name. For properties, there are no `()` with method argument values.

We've elided some details about how descriptors are used. For the most common use cases, the presence of the descriptor isn't important.

The essence of this is the search through `__dict__` (or `__slots__`) collections of names. If objects have a common superclass, then we can guarantee that a matching name will be found. If objects do not have a common superclass, then we don't have the same kind of guarantee. We have to rely on disciplined design and good test coverage.

There's more...

When we look at the `decimal` module we see an example of a numeric type that is distinct from all of the other numeric types. In order to make this work out well, the `numbers` module includes a concept of registering a class as a part of the `Number` class hierarchy. This injects a new class into the hierarchy without using inheritance.

A similar technique is used by the `codecs` module to add new data encodings. We can define a new encoding and register it without using any of the classes defined in the `codecs` module.

Previously, we noted that the search for a method of a class involves the concept of a descriptor. Internally, Python uses descriptor objects to create gettable and settable properties of an object.

A descriptor object must implement some combination of the special methods `__get__`, `__set__`, and `__delete__`. When the attribute appears in an expression, then `__get__` will be used to locate the value. When the attribute appears on the left side of an assignment, then `__set__` is used. In a `del` statement, the `__delete__` method is used.

The descriptor object acts as an intermediary so that a simple attribute can be used in a variety of contexts. It's rare to use descriptors directly. We can use the `@property` decorator to build descriptors for us.

See also

- The duck type question is implicit in the *Choosing between inheritance and extension – 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. Since it's very simple to use the mixin design pattern, duck typing is rarely needed.

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 an explicit initialization code.

One example of this is an implicit random number generating object that's part of the `random` module. When we evaluate `random.random()`, we're actually making use of an instance of the `random.Random` class that's an implicit part of the `random` module.

Other examples of this include the following:

- The collection of numeric types available. By default, we only have `int`, `float`, and `complex`. We can, however, add more numeric types, and they will work seamlessly with existing types. There's a global registry of available numeric types.
- The collection of data code/decode methods (codecs) available. The `codecs` module lists the available encoders and decoders. This also involves an implicit registry. We can add encodings and decodings to this registry.
- The `webbrowser` module has a registry of known browsers. For the most part, the operating system default browser is the one preferred by the user and the right one to use, but it's possible for an application to launch a browser other than the user's preferred browser. It's also possible to register a new browser that's unique to an application.

How can we work with this kind of implicit global object?

Getting ready

Generally, an implicit object can cause some confusion. The idea is to provide a suite of features as separate functions rather than methods of an object. The benefit, however, is to allow independent modules to share a common object without having to write any code that explicitly coordinates between the modules.

For a simple example, we'll define a module that will have a global singleton object. We'll look more at modules in `Chapter 11`, *Application integration*.

Our global object will be a counter that we can use to accumulate centralized data from several independent modules or objects. We'll provide an interface to this object using simple functions.

The goal is to be able to write something like this:

```
for row in source:
    count('input')
    some_processing()
print(counts())
```

This implies two functions that will refer to a global counter:

- `count()`: It will increment the counter and return the current value
- `counts()`: It will provide all of the various counter values

How to do it...

There are two ways to do handle global state information. One technique uses a module global variable because modules are singleton objects. The other uses a class level (static) variable because a class definition is a singleton object, also we'll show both techniques.

Module global variable

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:

   ```
   from collections import Counter
   ```

   In some cases, a `types.SimpleNamespace` might be used. In other cases, a more complex class with methods as well as attributes may be necessary.

3. Define the one and only instance of the global singleton object:

   ```
   _global_counter = Counter()
   ```

   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.

4. Define any wrapper functions:

```
def count(key, increment=1):
    _global_counter[key] += increment
def counts():
    return _global_counter.most_common()
```

We've defined two functions that use the global object, `_global_counter`. These functions encapsulate a detail of how the counter is implemented.

Now we can write applications that use the `count()` function in a variety of places. The counted events, however, are fully centralized into this single object.

We might have code that looks like this:

```
>>> from ch07_r04 import count, counts
>>> from ch07_r03 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 a central module. We've also imported the `Dice1` object as a handy object that we can use to create 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.

We can then use the object, `d`, to create random events. For this demonstration, we've categorized the events into two simple buckets, labeled `seven` and `other`. The `count()` function uses an implied global object.

When the simulation is done, we can dump the results using the `counts()` function. This will access the global object defined in the module.

The benefit of this technique is that several modules can all share the global object within the `ch07_r04` module. All that's required is an `import` statement. No further coordination or overheads are necessary.

Class-level static variable

1. Define a class and provide a variable outside the `__init__` method. This variable is part of the class, not part of each individual instance. It's shared by all instances of the class:

```
from collections import Counter
class EventCounter:
    _counts = Counter()
```

   We've given the class-level variable a leading underscore to make it less public. This is a note to anyone using the class that the attribute is an implementation detail that might change. It's not part of the visible interface to the class.

2. Add methods to update and extract data from this variable:

```
def count(self, key, increment=1):
    EventCounter._counts[key] += increment
def counts(self):
    return EventCounter._counts.most_common()
```

   We didn't use `self` in this example to make a point about variable assignment and instance variables. When we use `self.name` on the right side of an assignment statement, the name may be resolved by the object, or the class, or any superclass. This is the ordinary rule for searching a class.

   When we use `self.name` on the left side of assignment, that will create an instance variable. We must use `Class.name` to be sure that a class-level variable is updated instead of creating an instance variable.

The various application components can create objects, but the objects all share a common class level value:

```
>>> from ch07_r04 import EventCounter
>>> c1 = EventCounter()
>>> c1.count('input')
>>> c2 = EventCounter()
>>> c2.count('input')
>>> c3 = EventCounter()
>>> c3.counts()
[('input', 2)]
```

In this example, we've created three separate objects, `c1`, `c2`, and `c3`. Since all three share a common variable, defined in the `EventCounter` class, each can be used to increment that shared variable. These objects could be part of separate modules, separate classes, or separate functions, yet still share a common global state.

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.

We have two ways to share these kinds of global singleton variables:

- Using the module name, explicitly. We could have simply created an instance of `Counter` in the module and shared this via `counter.counter`. This can work, but it exposes an implementation detail.
- Using wrapper functions, as shown in this recipe. This requires a little more code, but it permits a change in implementation without breaking other parts of the application.

The functions provide a way to identify relevant features of the global variable, while encapsulating details of how it's implemented. This gives us the freedom to consider changing the implementation details. As long as the wrapper functions have the same semantics, the implementation can be changed freely.

Since we generally provide only one definition of a class, the Python import mechanism tends to assure us that the class definition is a proper singleton object. If we make the mistake of copying a class definition, and pasting it into two or more modules used by a single application, we would not share a single global object among these class definitions. This is an easy mistake to avoid.

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, three variables share a common `Counter` object. The presence of an implicitly shared global state can be confusing.

There's more...

A shared global state is in a way the opposite of object-oriented programming. One ideal of object-oriented programming is to encapsulate all state changes in individual objects. When we have a shared global state, we have strayed from this ideal:

- Using wrapper functions makes the shared object implicit
- Using a class-level variable conceals the fact that an object is shared

The alternative, of course, is to create a global object explicitly, and make it part of the application in some more obvious way. This might mean providing the object as an initialization parameter to objects throughout the application. This can be a fairly large burden in a complex application.

Having a few shared global objects is more appealing because the application becomes simpler. When these objects are used for pervasive features such as audits, logging, and security, they can be helpful.

This is a technique that is open for abuse. A design that relies on too many global objects can be confusing. It can also harbor subtle bugs because the encapsulation of objects in classes may be difficult to discern. It may also make unit test cases hard to write because of the implicit relationships among objects.

Using more complex structures – maps of lists

In `Chapter 4`, *Built-in Data Structures – list, set, dict*, we looked at the basic data structures available in Python. The recipes generally looked at the various structures in isolation.

We'll look at a common combination structure—the mapping from a key to a list. This is used to accumulate detailed information about an object identified by a given key. This recipe will transform a flat list of details into a structure that for one column contains values taken from other columns.

Getting ready

We'll work with an imaginary web log that's been transformed from the raw web format to a **CSV** (**comma-separated value**) format. This kind of transformation is often done with a regular expression that picks out the various syntactic groups. See the *String parsing with regular expressions* recipe in `Chapter 1`, *Numbers, Strings, and Tuples*, for information on how the parsing might work.

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 line in the file has a timestamp, a severity level, a module name, and some text. After parsing, the data is effectively a flat list of events. It looks like this:

```
>>> data = [
    ('2016-04-24 11:05:01,462', 'INFO', 'module1', 'Sample Message One'),
    ('2016-04-24 11:06:02,624', 'DEBUG', 'module2', 'Debugging'),
    ('2016-04-24 11:07:03,246', 'WARNING', 'module1', 'Something might have
gone wrong')
]
```

We'd like to examine the log, creating a list of all the messages organized by module, instead of sequentially through time. This kind of restructuring can make analysis simpler.

How to do it...

1. Import `defaultdict` from `collections`:

```
from collections import defaultdict
```

2. Use the `list` function as the default value for `defaultdict`:

```
module_details = defaultdict(list)
```

3. Iterate through the data, appending to the list associated with each key. The `defaultdict` object will use the `list()` function to build an empty list for each new key:

```
for row in data:
    module_details[row[2]].append(row)
```

The result of this will be a dictionary that maps from a module to a list of all log rows for that module name. The data will look like the following:

```
    {
        'module1': [
            ('2016-04-24 11:05:01,462', 'INFO', 'module1', 'Sample Message
One'),
            ('2016-04-24 11:07:03,246', 'WARNING', 'module1', 'Something
might have gone wrong')
            ],
        'module2': [
            ('2016-04-24 11:06:02,624', 'DEBUG', 'module2', '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.

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.
- The class `defaultdict` 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. In this case, the function is `list` to create an empty list.

We can imagine using the `set` function to create an empty `set` object for a missing key. This would be suitable for a mapping from a key to a set of objects that share that key.

There's more...

When we think about Python 3.5 and the ability to do type inferencing, we need to have a way to describe this structure:

```
from typing import *
def summarize(data) -> Mapping[str, List]:
    the body of the function.
```

This uses the notation `Mapping[str, List]` to show that the result is a mapping from string keys to a list of string data items.

We can also build a version of this as an extension to the built-in `dict` class:

```
class ModuleEvents(dict):
    def add_event(self, event):
        if event[2] not in self:
            self[event[2]] = list()
        self[event[2]].append(row)
```

We've defined a method that's unique to this class, `add_event()`. This will add the empty list if the key, the module name in `event[2]`, is not currently present in the dictionary. After the `if` statement, a postcondition could be added to assert that the key is now in the dictionary.

This allows us to use code such as the following:

```
module_details = ModuleEvents()
for row in data:
    module_details.add_event(row)
```

The resulting structure is very similar to `defaultdict`.

See also

- In the *Creating dictionaries – inserting and updating* recipe in `Chapter 4`, *Built-in Data Structure – list, set, dict*, we looked at the basics of using a mapping
- In the *Avoiding mutable default values for function parameters* recipe of `Chapter 4`, *Built-in Data Structure – list, set, dict*, we looked at other places where default values are used
- In the *Using more sophisticated collections* recipe of `Chapter 6`, *Basics of Classes and Objects*, we looked at other examples of using the `defaultdict` class

Creating a class that has orderable objects

When simulating card games, it's often essential to be able to sort the `Card` objects into a defined order. When cards form a sequence, sometimes called a straight, this can be an important way to score the hand. This is part of games such as Poker, Cribbage, and even Pinochle.

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 `__hash__()`.

In order to keep items in a sorted collection, we'll need the comparison methods that implement <, >, <=, >=, ==, and `!=`. These comparisons are based on the attribute values of each object.

How do we create comparable objects?

Getting ready

The game of Pinochle generally involves a deck with 48 cards. There are six ranks—9, 10, Jack, Queen, King, and Ace. There are the standard four suits. Each of these 24 cards appears twice in the deck. We have to be careful of using a structure such as a dict or set because cards are not unique in Pinochle; there can be duplicates.

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.

We'll need to add features to those cards to create objects that can be ordered properly. In order to support the *Defining an ordered collection* recipe, we'll look at cards for the game of Pinochle.

Here are the first two elements of the design:

```
from ch07_r02 import AceCard, Card, FaceCard, SUITS
class PinochlePoints:
    _points = {9: 0, 10:10, 11:2, 12:3, 13:4, 14:11}
    def points(self):
        return self._points[self.rank]
```

We've imported the existing `Card` hierarchy. We've also defined a rule for computing the points for each card taken in a trick during play, the `PinochlePoints` class. This has a mapping from card ranks to the potentially confusing points for each card.

A 10 is worth 10 points, and an Ace is worth 11 points, but the King, Jack, and Queen are worth four, three, and two points respectively. This can be confusing for new players.

Because an Ace ranks above a King for purposes of identifying a straight, we've made the rank of the Ace 14. This slightly simplifies the processing.

In order to use a sorted collection of cards, we need to add yet another feature to the cards. We'll need to define the comparison operations. There are six special methods used for object comparison.

How to do it...

1. We're using a mixin design. Therefore, we'll create a new class to hold the comparison features:

   ```
   class SortedCard:
   ```

   This class will join a member of the `Card` hierarchy plus the `PinochlePoints` to create the final composite class definition.

2. Define the six comparison methods:

   ```
   def __lt__(self, other):
       return (self.rank, self.suit) < (other.rank, other.suit)

   def __le__(self, other):
       return (self.rank, self.suit) <= (other.rank, other.suit)

   def __gt__(self, other):
       return (self.rank, self.suit) > (other.rank, other.suit)

   def __ge__(self, other):
       return (self.rank, self.suit) >= (other.rank, other.suit)

   def __eq__(self, other):
       return (self.rank, self.suit) == (other.rank, other.suit)

   def __ne__(self, other):
       return (self.rank, self.suit) != (other.rank, other.suit)
   ```

   We've written all six comparisons out in full. We've converted the relevant attributes of a `Card` into a tuple, and relied on Python's built-in tuple comparison to handle the details.

3. Write the composite class definitions, built from an essential class and two mixin classes to provide additional features:

```
class PinochleAce(AceCard, SortedCard, PinochlePoints):
    pass

class PinochleFace(FaceCard, SortedCard, PinochlePoints):
    pass

class PinochleNumber(Card, SortedCard, PinochlePoints):
    pass
```

   The final class contains elements with three separate, and largely independent feature sets: the essential `Card` features, the mixin comparison features, and the mixin Pinochle specific features.

4. Create a function that will create individual card objects from the classes defined previously:

```
def make_card(rank, suit):
    if rank in (9, 10):
        return PinochleNumber(rank, suit)
    elif rank in (11, 12, 13):
        return PinochleFace(rank, suit)
    else:
        return PinochleAce(rank, suit)
```

   Even though the point rules are dauntingly complex, the complexity is hidden in the `PinochlePoints` class. 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:

```
>>> from ch07_r06a import make_card
>>> c1 = make_card(9, '♡')
>>> c2 = make_card(10, '♡')
>>> c1 < c2
True
>>> c1 == c1
True
>>> c1 == c2
False
>>> c1 > c2
False
```

Here's a function that builds the 48-card deck:

```
SUITS = '\u2660\u2661\u2662\u2663'
Spades, Hearts, Diamonds, Clubs = SUITS
def make_deck():
    return [make_card(r, s) for _ in range(2)
        for r in range(9, 15)
        for s in SUITS]
```

The value of `SUITS` is four Unicode characters. We could have set each suit string separately, but this seems slightly simpler. The generator expression inside the `make_deck()` function builds two copies of each card. There are only six ranks and four suits.

How it works...

Python uses special methods for a vast number of things. Almost every visible behavior in the language is due to some special method name. In this recipe, we've leveraged the six comparison operators.

Write the following:

```
c1 <= c2
```

The preceding code is evaluated as if we'd written the following:

```
c1.__le__(c2)
```

This kind of transformation happens for all of the expression operators.

Careful study of *Section 3.3* of the *Python Language Reference* shows that the special methods can be organized into several distinct groups:

- Basic customization
- Customizing attribute access
- Customizing class creation
- Customizing instance and subclass checks
- Emulating callable objects
- Emulating container types
- Emulating numeric types
- With statement context managers

In this recipe, we've looked at only the first of these categories. The others follow some similar design patterns.

Here's how it looks when we create instances of this class hierarchy. The first example will create a 48-card Pinochle deck:

```
>>> from ch07_r06a import make_deck
>>> deck = make_deck()
>>> len(deck)
48
```

If we look at the first eight cards, we can see how they're built from all the combinations of rank and suit:

```
>>> deck[:8]
[ 9 ♣,  9 ♡,  9 ♢,  9 ♠, 10 ♣, 10 ♡, 10 ♢, 10 ♠]
```

If we look at the second half of the deck, we can see that it has the same cards as the first half of the deck:

```
>>> deck[24:32]
[ 9 ♣,  9 ♡,  9 ♢,  9 ♠, 10 ♣, 10 ♡, 10 ♢, 10 ♠]
```

Since the `deck` variable is a simple list, we can shuffle the list object and pick a dozen cards.

```
>>> import random
>>> random.seed(4)
>>> random.shuffle(deck)
>>> sorted(deck[:12])
[ 9 ♠, 10 ♠,  J ♣,  J ♢,  J ♢,  Q ♣,  Q ♠,  K ♣,  K ♣,  K ♠, A ♡,  A ♠]
```

The important part is the use of the `sorted()` function. Because we've defined proper comparison operators, we can sort the `Card` instances, and they are presented in the expected order.

There's more...

A little formal logic suggests that we really only need to implement two of the comparisons. With any two, all the others can be derived. For example, if we could only do the operations for less than (`__lt__()`) and equal to (`__eq__()`), we could compute the missing three fairly easily:

$a \le b \equiv a < b \vee a = b$

$a \ge b \equiv a > b \vee a = b$

$a \neq b \equiv \neg(a = b)$

Python emphatically does not do any of this kind of advanced algebra for us. We need to do the algebra carefully, or if we're unsure of the logic, we can write out all six comparisons in full.

We've assumed that each `Card` is compared against another card. Try this:

```
>>> c1 = make_card(9, '♡')
>>> c1 == 9
```

We'll get an `AttributeError` exception.

If we need this feature, we'll have to modify the comparison operators to handle two kinds of comparison:

- `Card` against `Card`
- `Card` against `int`

This is done by using the `isinstance()` function to discriminate between the argument types.

Each of our comparison methods would be changed to look like this:

```
def __lt__(self, other):
    if isinstance(other, Card):
        return (self.rank, self.suit) < (other.rank, other.suit)
    else:
        return self.rank < other
```

This handles the `Card` against the `Card` case using rank and suit comparisons. For all other cases, Python's ordinary rules are used to compare the rank against the other value. If, for some obscure reason, the value of the other was `float`, then a `float()` conversion would be used on `self.rank`.

See also

- See the *Defining an ordered collection* recipe that relies on sorting these cards into order

Defining an ordered collection

When simulating card games, the player's hand can be modeled as a set of cards or a list of cards. With most conventional single-deck games, a set works out nicely because there's only one instance of any given card, and the set class can do very fast operations to confirm that a given card is (or is not) in the set.

When modeling Pinochle, however, we have a challenging problem. The Pinochle deck is 48 cards; it has two of 9, 10, Jack, Queen, King, and Ace. A simple set won't work well for this; we would need a multiset or bag. This is a set that permits duplicate items.

The operations are still limited to membership tests. For example, we can add the object `Card(9,'♢')` object more than once, and then also remove it more than one time.

We have a number of ways to create a multiset:

- We can use a list. Appending an item has a nearly fixed cost, characterized as $O(1)$. Searching for an item has a bad performance problem. The complexity of testing for membership tends to grow with the size of the collection. It becomes $O(n)$.
- We can use a mapping; the value can be an integer count of the number of times a duplicated element shows up. This only requires that the default `__hash__()` method is available for each object in the mapping. We have three ways of implementing this:
    - Define our own subclass of dict.
    - Use a `defaultdict`. See the *Using more complex structures - maps of lists* recipe, which uses `defaultdict(list)` to create a list of values for each key. The `len()` of this list is the number of times the key occurred. In effect, this is a kind of multiset.
    - Use a `Counter`. This can be very simple. We've looked at `Counter` in a number of recipes. See the *Avoiding mutable default values for function parameters* recipe in `Chapter 4`, *Built-in Data Structures – list, set, dict*, also the *Designing classes with lots of processing* and *Using properties for lazy attributes* recipes in `Chapter 6`, *Basics of Classes and Objects*, and the *Managing global and singleton objects* recipe of this chapter for other examples.
- We can use a sorted list. Inserting an item that maintains this sort sequence is slightly more expensive than inserting into a list, $O(n \log_2 n)$. Searching, however, is less expensive than an unsorted list; it's $O(\log_2 n)$. The `bisect` module provides a set of functions that do this nicely. This, however, requires objects with a full set of comparison methods.

How can we build a sorted collection of objects? How can we build a multiset or bag using a sorted collection?

Getting ready

In the *Creating a class that has orderable objects* recipe, we defined cards that could be sorted. This is essential for using `bisect`. The algorithms in this module require a full set of comparisons among objects.

We'll define a multiset to keep 12-card Pinochle hands. Because of the duplication, there will be more than one card of a given rank and suit.

In order to view a hand as a kind of set, we'll also need to define some set operators on hand objects. The idea is to define set membership and subset operators.

We'd like to have Python code that's equivalent to the following:

$c \in H$

This is for a card, c, and a hand of cards, $H = \{c_1, c_2, c_3, ...\}$

We'd also like code equivalent to this:

$\{J, Q\} \subset H$

This is for a specific pair of cards, called the Pinochle, and a hand of cards, H.

We'll need to import two things:

```
from ch07_r06a import *
import bisect
```

The first import brings in our orderable card definitions from the *Creating a class that has orderable objects* recipe. The second import brings in the various bisect functions that we'll use to maintain an ordered set with duplicates.

How to do it...

1. Define a class with an initialization that can load the collection from any iterable source of data:

```
class Hand:
    def __init__(self, card_iter):
        self.cards = list(card_iter)
        self.cards.sort()
```

We can use this to build a `Hand` from a list or possibly a generator expression. If the list is non-empty, we'll need to sort the items into order. The `sort()` method of the `self.cards` list will rely on the various comparison operators implemented by the `Card` objects.

Technically, we only care about objects that are subclasses of `SortedCard`, since that is where the comparison methods are defined.

2. Define a method to add cards to a hand:

```
def add(self, aCard: Card):
    bisect.insort(self.cards, aCard)
```

We've used the `bisect` algorithm to assure that the card is properly inserted into the `self.cards` list.

3. Define a method to find the position of a given card in a hand:

```
def index(self, aCard: Card):
    i = bisect.bisect_left(self.cards, aCard)
    if i != len(self.cards) and self.cards[i] == aCard:
        return i
    raise ValueError
```

We've used the `bisect` algorithm for locating a given card. The additional `if` test is recommended in the documentation for `bisect.bisect_left()` to properly handle an edge case in the processing.

4. Define the special method that implements the `in` operator:

```
def __contains__(self, aCard: Card):
    try:
        self.index(aCard)
        return True
    except ValueError:
```

```
            return False
```

When we write `card in some_hand` in Python, it's evaluated as if we had written `some_hand.__contains__(card)`. We've used the `index()` method to either find the card or raise an exception. The exception is transformed into a return value of `False`.

5. Define an iterator over the hand. This is a simple delegation to the `self.cards` collection:

```
def __iter__(self):
    return iter(self.cards)
```

When we write `iter(some_hand)` in Python, it's evaluated as if we had written `some_hand.__iter__()`.

6. Define a subset operation between two hand instances:

```
def __le__(self, other):
    for card in self:
        if card not in other:
            return False
    return True
```

Python doesn't have the $a \subset b$ or $a \subseteq b$ symbols, so < and <= are pressed into service for comparing sets. When we write `pinochle <= some_hand` to see if the hand contains a specific combination of cards, it's evaluated as if we'd written `pinochle.__le__(some_hand)`. The subset is the `self` instance variable, and the target `Hand` is the other parameter value.

The in operator is implemented by the `__contains__()` method. This shows how the simple Python syntax is implemented by the special methods.

We can use this `Hand` class like this:

```
>>> from ch07_r06b import make_deck, make_card, Hand
>>> import random
>>> random.seed(4)
>>> deck = make_deck()
>>> random.shuffle(deck)
>>> h = Hand(deck[:12])
>>> h.cards
[ 9 ♠, 10 ♠,  J ♣,  J ♢,  J ♢,  Q ♣,  Q ♠,  K ♣,  K ♣,  K ♠, A ♡,  A ♠]
```

The cards are properly sorted in the hand. This is a consequence of the way the hand was created.

Here's an example of using the subset operator, `<=`, to compare a specific pattern to the hand as a whole:

```
>>> pinochle = Hand([make_card(11,'♢'), make_card(12,'♣')])
>>> pinochle <= h
True
```

A `Hand` is a collection, and supports iteration. We can use generator expressions that reference the `Card` objects within the overall `Hand`:

```
>>> sum(c.points() for c in h)
56
```

How it works...

Our `Hand` collection works by wrapping an internal `list` object and applying an important constraint to that object. The items are kept in sorted order. This increases the cost to insert a new item, but reduces the cost to search for an item.

The core algorithms for locating the position for an item are part of the `bisect` module, saving us from having to write (and debug) them. The algorithms aren't really very complex. But it seems more efficient to leverage existing code.

The module's name comes from the idea of bisecting the sorted list to look for an item. The essence is this:

```
while lo < hi:
    mid = (lo+hi)//2
    if x < a[mid]: hi = mid
    else: lo = mid+1
```

This searches a list, `a`, for a given value, `x`. The value of `lo` is initially zero and the value of `hi` is initially the size of the list, `len(a)`.

First, the midpoint is identified. If the target value, `x`, is less than the midpoint value, `a[mid]`, then it must be in the first half of the list: the value of `hi` is shifted so that only the first half is considered.

If the target value, `x`, is greater than or equal to the midpoint value, `a[mid]`, then `x` must be in the second half of the list: the value of `lo` is shifted so that only the second half is considered.

Since the list is chopped in half at each operation, it requires $O(\log_2 n)$ steps to have the values of `lo` and `hi` converge on the position that should have the target value.

If we have a hand with 12 cards, then the first comparison discards six. The next comparison discards three more. The next comparison discards one of the final three. The fourth comparison will locate the position the card should occupy.

If we use an ordinary list, with cards stored in the random order of arrival, then finding a card will take an average of six comparisons. The worst possible case means it's the last of 12 cards, requiring all 12 to be examined.

With `bisect` the number of comparisons is always $O(\log_2 n)$. That's the average as well as the worst case.

There's more...

The `collections.abc` module defines abstract base classes for various collections. If we want our `Hand` to behave more like other kinds of sets, we can leverage these definitions.

We can add numerous set operators to this class definition to make it behave more like the built-in `MutableSet` abstract class definition.

A `MutableSet` is an extension to `Set`. The `Set` class is a composite built from three class definitions: `Sized`, `Iterable`, and `Container`. This means that it must define the following methods:

- `__contains__()`
- `__iter__()`
- `__len__()`
- `add()`
- `discard()`

We'll also need to provide some other methods that are part of being a mutable set:

- `clear()`, `pop()`: These will remove items from the set.
- `remove()`: Unlike `discard()`, this will raise an exception when attempting to remove a missing item.

In order to have unique set-like features, it also needs a number of additional methods. We provided an example of a subset, based on `__le__()`. We also need to provide the following subset comparisons:

- `__le__()`
- `__lt__()`
- `__eq__()`
- `__ne__()`
- `__gt__()`
- `__ge__()`
- `isdisjoint()`

These are generally not trivial one-line definitions. In order to implement the core set of comparisons, we'll often write two and then use logic to build the remainder based on those two.

Since `__eq__()` is simple, let's assume we have definitions for the == and <= operators. The others would be defined as follows:

$x \neq y \equiv \neg(x = y)$

$x < y \equiv (x \leq y) \wedge \neg(x = y)$

$x > y \equiv \neg(x \leq y)$

$x \geq y \equiv \neg(x < y) \equiv \neg(x \leq y) \vee (x = y)$

In order to do set operations, we'll need to provide the following:

- `__and__()` and `__iand__()`. These methods implement the Python `&` operator and the `&=` assignment statement. Between two sets, this is a set intersection, or $a \cap b$.
- `__or__()` and `__ior__()`. These methods implement the Python `|` operator and the `|=` assignment statement. Between two sets, this is a set union, or $a \cup b$.

- `__sub__()` and `__isub__()`. These methods implement the Python – operator and the -= assignment statement. Between sets, this is a set difference, often written as *a* - *b*.
- `__xor__()` and `__ixor__()`. These methods implement the Python ^ operator and the ^= assignment statement. When applied between two sets, this is the symmetric difference, often written as $a \Delta b$.

The abstract class permits two versions of each operator. There are two cases:

- If we provide `__iand__()`, for example, then the statement `A &= B` will be evaluated as `A.__iand__(B)`. This might permit an efficient implementation.
- If we do not provide `__iand__()`, then the statement `A &= B` will be evaluated as `A = A.__and__(B)`. This might be somewhat less efficient because we'll create a new object. The new object is given the label `A`, and the old object will be removed from memory.

There are almost two dozen methods that would be required to provide a proper replacement for the built-in set class. On one the hand, it's a lot of code. On the other hand, Python lets us extend the built-in classes in a way that's transparent and uses the same operators with the same semantics.

See also

- See the *Creating a class that has orderable objects* recipe for the companion recipe that defines Pinochle cards

Deleting from a list of mappings

Removing items from a list has an interesting consequence. Specifically, when item `list[x]` is removed, one of two other things will happen:

- Item `list[x+1]` takes the place of `list[x]`
- Item `x+1 == len(list)` takes the place of `list[x]` because `x` was the last index in the list

These are side-effects that happen in addition to removing an item. Because things can move around in a list, it makes deleting more than one item at a time 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 it becomes more challenging to remove multiple items from the list.

How can we delete multiple items from a list?

Getting ready

We'll work with a list-of-dict structure. In this case, we've got some data that includes a song name, the writers, and a duration. The data looks like this:

```
>>> source = [
...     {'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'}
... ]
```

To work with this kind of data structure, we'll need the `pprint` function:

```
>>> from pprint import pprint
```

We can easily traverse the list of values with the `for` statement. The problem is, how do we delete selected items?

```
>>> data = source.copy()
>>> for item in data:
...     if 'Lake' in item['writer']:
...         print("remove", item['title'])
remove Stones of Years
remove Mass
remove Battlefield
```

We can't simply use the statement `del item` here, because it has no effect on the source collection, `data`. This statement would only delete the local variable copy of the item in the original list by deleting the `item` variable and the associated object.

To properly delete items from a list, we must work with index positions in the list. Here's a naïve approach that emphatically does not work:

```
>>> data = source.copy()
>>> for index in range(len(data)):
...     if 'Lake' in data[index]['writer']:
...         del data[index]
Traceback (most recent call last):
  File
"/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/doctest.py
", line 1320, in __run
    compileflags, 1), test.globs)
  File "<doctest __main__.__test__.chapter[5]>", line 2, in <module>
    if 'Lake' in data[index]['writer']:
IndexError: list index out of range
```

We can't simply use `range(len(data))` based on the original size of the list. As items are removed, the list gets smaller. The value of the index will be set to a value that's too large.

When removing simple items that have simple equality tests, we would use something like this:

```
while x in list:
    list.remove(x)
```

The problem is that we don't have an implementation of `__contains__()` that identifies items with `Lake` in `item['writer']`. We could use a subclass of dict that implements `__eq__()` as a string parameter value in `self['writer']`. This clearly violates the semantics of equality because it only checks a single field.

We can't extend the built-in features of these classes. The use case here is very specific to the problem domain, not a general feature of the list of dict structure.

To parallel the basic `while in...remove` loop, we need to write something like this:

```
>>> def index(data):
...     for i in range(len(data)):
...         if 'Lake' in data[i]['writer']:
...             return i
>>> data = source.copy()
>>> position = index(data)
>>> while position:
...     del data[position] # or data.pop(position)
...     position = index(data)
```

We've written a function, `index()`, that locates the first instance of the target value. The result of this function is a single value that provides two kinds of information:

- When the value returned is not `None`, the item exists in the list
- The return value is the proper index for the item within the list

The `index()` function is wordy and inflexible. If we have alternate rules, we need to either write multiple `index()` functions or we need to make the test more flexible.

More importantly, consider when a target value occurs x times in a list of n items. There will be x trips through this loop. Each trip through the loop examines an average of $\boldsymbol{O}(x \times n/2)$ trips through the list. The worst case is that the items are all at the end of the list, leading to just under $\boldsymbol{O}(x \times n)$ processing iterations.

We can do better. Our preferred solution builds on the ideas in the *Designing a while statement which terminates properly* recipe in `Chapter 2`, *Statements and Syntax*, to design a proper loop for removing complex items from a list structure.

How to do it...

1. Initialize an index value to zero. This establishes a variable that will traverse the data collection:

```
i = 0
```

2. The terminating condition must show that every item in the list has been examined. Additionally, the body of the loop needs to remove all of the items that match the target criteria. This leads to an invariant condition that `item[i]` has not yet been examined. After the item is examined, it may be preserved, which means the index, `i`, must be incremented to reset the not yet examined invariant. If the item is removed, then items will shift forward and `item[i]` will automatically meet the not yet examined invariant:

```
if 'Lake' in data[i]['writer']:
    del data[i] # Remove
else:
    i += 1 # Preserve
```

When removing an item, the list becomes one shorter, and the index value, `i`, will point to a new, unexamined item. When preserving an item, the index value, `i`, is advanced to the next unexamined item.

3. The terminating condition is used to wrap the processing body:

```
while i != len(data):
```

   At the end of the `while` statement, the value of `i` will indicate that all items have been examined.

   This leads to the following:

```
>>> i = 0
>>> while i != len(data):
...     if 'Lake' in data[i]['writer']:
...         del data[i]
...     else:
...         i += 1
>>> pprint(data)
[{'time': '2:43', 'title': 'Eruption', 'writer': ['Emerson']},
 {'time': '1:16', 'title': 'Iconoclast', 'writer': ['Emerson']},
 {'time': '1:49', 'title': 'Manticore', 'writer': ['Emerson']},
 {'time': '3:54', 'title': 'Aquatarkus', 'writer': ['Emerson']}]
```

This makes exactly one pass through the data and removes the requested items without raising index errors, or skipping items that should have been deleted.

How it works...

The goal is to examine each item exactly once and either remove it or step over it. The loop design reflects the way that the Python list item removal works. When an item is removed, all of the subsequent items are shuffled forward in the list.

A naïve process based on the `range()` and `len()` functions will have two problems:

- Items will be skipped when the items shift forward and the next value is produced by the range object
- The index can go beyond the end of the list structure after items are removed because the `len()` was used once to get the original size, not the current size

Because of these two problems, the design of the invariant condition in the body of the loop is important. This reflects the two possible state changes:

- If an item is removed, the index must not change. The list itself will change.
- If an item is preserved, the index must change.

We can argue that the loop makes one trip through the data, and has a complexity of $O(n)$. What's not considered in this is the relative cost of each deletion. Deleting item `0` from a list means that each remaining item is shuffled forward one position. The cost of each deletion is effectively $O(n)$. Therefore the complexity is more like $O(n \times x)$, where x items are removed from a list of n items.

Even this algorithm isn't the fastest way to remove items from a list.

There's more...

If we give up on the idea of deleting, we can do even better. Making a shallow copy of items is much faster than removing items from a list, but uses more storage. This is a common time versus memory tradeoff.

We can use a generator expression like the following:

```
>>> data = [item for item in source if not('Lake' in item['writer'])]
```

This will create a shallow copy of the items in the list that we want to keep. 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`, *Built-in Data Structures – list, set, dict*.

We can also use a higher-order function such as this:

```
>>> data = list(filter(lambda item: not('Lake' in item['writer']), source))
```

The `filter()` function has two arguments: a `lambda` object, and the original set of data. The `lambda` object is a kind of degenerate case for a function: it has arguments and a single expression. In this case, the single expression is used to decide which items to pass. Items for which the lambda is `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. A `for` statement is one way to process all results from a generator. The `list()` and `tuple()` functions will also consume all items from a generator.

The third way we can implement this is to write our own generator function that embodies the filter concept. This will use more statements than the generator or the `filter()` function, but it might be more clear.

Here's a generator function definition:

```
def writer_rule(iterable):
    for item in iterable:
        if 'Lake' in item['writer']:
            continue
        yield item
```

We've used a `for` statement to examine each item in the source list. If the item has `'Lake'` in the list of writers, we'll continue the `for` statement process effectively rejecting this item. If `'Lake'` is not in the list of writers, we'll yield the item.

When we call this function, it will yield the interesting list. We can use the function `writer_rule()` like this:

```
>>> from ch07_r07 import writer_rule
>>> data = list(writer_rule(source))
>>> pprint(data)
[{'time': '2:43', 'title': 'Eruption', 'writer': ['Emerson']},
 {'time': '1:16', 'title': 'Iconoclast', 'writer': ['Emerson']},
 {'time': '1:49', 'title': 'Manticore', 'writer': ['Emerson']},
 {'time': '3:54', 'title': 'Aquatarkus', 'writer': ['Emerson']}]
```

This will accumulate the interesting rows into a new structure. Since it's a shallow copy, it doesn't waste vast amounts of storage.

See also

- This is based on the *Designing a while statement which terminates properly* recipe in `Chapter 2`, *Statements and Syntax*
- We've also leveraged two other recipes: *Making shallow and deep copies of objects* and *Slicing and dicing a list* in `Chapter 4`, *Built-in Data Structures – list, set, dict*

8
Input/Output, Physical Format, and Logical Layout

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

- Using pathlib to work with filenames
- Reading and writing files with context managers
- Replacing a file while preserving the previous version
- Reading delimited files with the CSV module
- Reading complex formats using regular expressions
- Reading JSON documents
- Reading XML documents
- Reading HTML documents
- Upgrading CSV from DictReader to namedtuple reader
- Upgrading CSV from DictReader to namespace reader
- Using multiple contexts for reading and writing files

Introduction

The term **file** is overloaded with many meanings:

- The **operating system** (**OS**) uses a file as a way to organize bytes of data. The bytes might represent an image, some sound samples, words, or even an executable program. All of the wildly different kinds of content are reduced to a collection of bytes. Application software makes sense of the bytes.

There are two common kinds of OS files:

  - Block files exist on devices such as disks or **solid state drives** (**SSD**). These files can be read in blocks of bytes. The OS can seek any specific byte within the file at any time.
  - Character files are a way to manage a device like a network connection, or a keyboard attached to a computer. The file is viewed as a stream of individual bytes, which arrive at seemingly random points of time. There's no way to seek forward or backwards in the stream of bytes.

- The word *file* also defines a data structure used by the Python runtime. The Python file abstraction wraps the various OS file implementations. When we open a file, there is a binding between a Python abstraction, an OS implementation, and the underlying collection of bytes on a disk or other device.
- A file can also be interpreted as a collection of Python objects. From this viewpoint, the bytes of the file represent Python objects such as strings or numbers. Files of text strings are very common and easy to work with. The Unicode characters are often encoded to bytes using the UTF-8 encoding scheme, but there are are many alternatives. Python provides modules such as `shelve` and `pickle` to encode more complex Python objects as bytes.

Often, we'll talk about how an object is serialized. When an object is written to a file, the Python object state information is transformed to a series of bytes. Deserialization is the reverse process of recovering a Python object from the bytes. We can also call this idea the representation of state because we generally serialize the state of each individual object separate from the class definition.

When we process data from files, we'll often need to make two distinctions:

- **The physical format of the data**: This answers the fundamental question of what Python data structure is encoded by the bytes in a file. The bytes might be Unicode text. The text could represent **comma-separated values** (**CSV**) or JSON documents. The physical format is commonly handled by Python libraries.
- **The logical layout of the data**: The layout looks at the details of the various CSV columns, or JSON fields within the data. In some cases, the columns may be labeled, or there may be data that must be interpreted by position. This is something that is often the responsibility of our application.

Both the physical format and logical layout 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 logical layout.

Using pathlib to work with filenames

Most operating systems use a hierarchical path to identify a file. Here's an example filename:

```
/Users/slott/Documents/Writing/Python Cookbook/code
```

This full pathname has the following elements:

- The leading `/` means the name is absolute. It starts from the root of the filesystem. In Windows, there can be an extra letter in front of the name, such as `C:`, to distinguish the filesystems on each individual storage device. Linux and Mac OS X treat all of the devices as a single, large filesystem.
- The names such as `Users`, `slott`, `Documents`, `Writing`, `Python Cookbook`, and `code` represent the directories (or folders) of the filesystem. There must be a top-level `Users` directory. It must contain the `slott` subdirectory. This is true for each name in the path.
- In Windows, the OS uses `\` to separate items on the path. Python uses `/`. Python's standard `/` is converted to the Windows path separator character gracefully; we can generally ignore the Windows `\`.

There is no way to tell what kind of object the name `code` represents. There are many kinds of filesystem objects. The name `code` might be a directory that names other files. It could be an ordinary data file, or a link to a stream-oriented device. There is additional directory information that shows what kind of filesystem object this is.

A path without the leading `/` is relative to the current working directory. In Mac OS X and Linux, the `cd` command sets the current working directory. In Windows, the `chdir` command does this job. The current working directory is a feature of the login session with the OS. It's made visible by the shell.

How can we work with pathnames in a way that's independent of the specific operating system? How can we simplify common operations to make them as uniform as possible?

Getting ready

It's important to separate two concepts:

- The path that identifies a file
- The contents of the file

The path provides an optional sequence of directory names and the final filename. It may provide some information about the file contents via the filename extension. The directory includes the files name, information about when the file was created, who owns it, what the permissions are, how big it is, and other details. The contents of the file are separate from the directory information and the name.

Often, a filename has a suffix that can provide a hint as to what the physical format is. A file 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.

It's possible for the contents of a file to have more than one name. Multiple paths can link to a single file. The directory entries that provide additional names for the file's content are created with the link (`ln`) command. Windows uses `mklink`. This is called a **hard link** because it's a low-level connection between names and content.

In addition to hard links, we can also have **soft links** or **symbolic links** (or junction points). A soft link is a different kind of file, the link is easily seen as a reference to another file. The GUI presentation of the OS may show these as a different icon and call it an alias or shortcut to make it clear.

In Python, the `pathlib` module handles all of the 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 and refer to an actual file

This distinction allows us to create pure paths for files that our application will likely create or refer to. We can also create concrete paths for those files that actually exist on the OS. An application can resolve a pure path to create a concrete path.

The `pathlib` module also makes a distinction between Linux path objects and Windows path objects. This distinction is rarely needed; most of the time, we don't want to care about the OS-level details of the path. An important reason for using `pathlib` is because we want processing that is identical irrespective of the underlying OS. The cases where we might want to work with a `PureLinuxPath` object are rare.

All of the mini recipes in this section will leverage the following:

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

We rarely need any of the other class definitions from `pathlib`.

We'll presume that `argparse` 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 5`, *User Inputs and Outputs*. We'll use the `options` variable, which has the `input` filename or directory name that the recipe works with.

For demonstration purposes, a mock argument parsing is shown by providing the following `Namespace` object:

```
>>> from argparse import Namespace
>>> options = Namespace(
...     input='/path/to/some/file.csv',
...     file1='/Users/slott/Documents/Writing/Python
Cookbook/code/ch08_r09.py',
...     file2='/Users/slott/Documents/Writing/Python
Cookbook/code/ch08_r10.py',
... )
```

This `options` object has three mock argument values. The `input` value is a pure path: it doesn't necessarily reflect an actual file. The `file1` and `file2` values reflect concrete paths that exist on the author's computer. This object behaves the same as the options created by the `argparse` module.

How to do it...

We'll show a number of common pathname manipulations as separate mini recipes. This will include the following manipulations:

- Making the output filename from the input filename
- Making a number of sibling output files
- Creating a directory and a number of files
- Comparing file dates to see which is newer
- Removing a file
- Finding all files that match a given pattern

Making the output filename by changing the input suffix

Perform the following steps to make the output filename by changing the input suffix:

1. Create the `Path` object from the input filename string. The `Path` class will properly parse the string to determine the elements of the path:

```
>>> input_path = Path(options.input)
>>> input_path
PosixPath('/path/to/some/file.csv')
```

   In this example, the `PosixPath` class is displayed because the author is using Mac OS X. 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. The `with_suffix()` method saves us from manually parsing the text of the filename.

Making a number of sibling output files with distinct names

Perform the following steps for making a number of sibling output files with distinct names:

1. Create a `Path` object from the input filename string. The `Path` class will properly parse the string to determine the elements of the path:

```
>>> input_path = Path(options.input)
>>> input_path
PosixPath('/path/to/some/file.csv')
```

   In this example, the `PosixPath` class is displayed because the author uses Linux. On a Windows machine, the class would be `WindowsPath`.

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 filename. An input file of `file.csv` will produce an output of `file_pass.csv`:

```
>>> output_stem_pass = input_stem+"_pass"
>>> output_stem_pass
'file_pass'
```

4. Build the complete `Path` object:

```
>>> 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 this in parentheses to be sure that it's performed first and creates a new `Path` object. The `input_directory` variable has the parent `Path` object, and the `output_stem_pass` is a simple string. After assembling a new path with the `/` operator, the `with_suffix()` method is used to assure a specific suffix is used.

Creating a directory and a number of files

The following steps are for creating a directory and a number of files:

1. Create the `Path` object from the input filename string. The `Path` class will properly parse the string to determine the elements of the path:

```
>>> input_path = Path(options.input)
>>> input_path
PosixPath('/path/to/some/file.csv')
```

In this example, the `PosixPath` class is displayed because the author uses Linux. On a Windows machine, the class would be `WindowsPath`.

2. Create the `Path` object for the output directory. In this case, we'll create an `output` directory as a subdirectory with the same parent directory as the source file:

```
>>> output_parent = input_path.parent / "output"
>>> output_parent
PosixPath('/path/to/some/output')
```

3. Create the output filename using the output `Path` object. In this example, the output directory will contain a file that has the same name as the input with a different suffix:

```
>>> input_stem = input_path.stem
>>> output_path = (output_parent / input_stem).with_suffix('.src')
```

We've used the `/` operator to assemble a new `Path` object from the parent `Path` and a string based on the stem of a filename. Once a `Path` object has been created, we can use the `with_suffix()` method to set the desired suffix for the file.

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)
      >>> file1_path
      PosixPath('/Users/slott/Documents/Writing/Python
Cookbook/code/ch08_r09.py')
      >>> file2_path = Path(options.file2)
      >>> file2_path
      PosixPath('/Users/slott/Documents/Writing/Python
Cookbook/code/ch08_r10.py')
```

2. Use the `stat()` method of each `Path` object to get timestamps for the file. This method returns a `stat` object, within that `stat` object, the `st_mtime` attribute of that object provides the most recent modification time for the file:

```
>>> file1_path.stat().st_mtime
1464460057.0
>>> file2_path.stat().st_mtime
1464527877.0
```

The values are timestamps measured in seconds. We can easily compare the two values to see which is newer.

If we want a timestamp that's sensible to people, we can use the `datetime` module to create a proper `datetime` object from this:

```
>>> import datetime
>>> mtime_1 = file1_path.stat().st_mtime
>>> datetime.datetime.fromtimestamp(mtime_1)
datetime.datetime(2016, 5, 28, 14, 27, 37)
```

We can use the `strftime()` method to format the `datetime` object or we can use the `isoformat()` method to provide a standardized display. Note that the time will have the local time zone offset implicitly applied to the OS timestamp; depending on the OS configuration(s) a laptop may not show the same time as the server that created it because they're in different time zones.

Removing a file

The Linux term for removing a file is **unlinking**. Since a file may have many links, the actual data isn't removed until all links are removed:

1. Create the `Path` object from the input filename string. The `Path` class will properly parse the string to determine the elements of the path:

```
>>> input_path = Path(options.input)
>>> input_path
PosixPath('/path/to/some/file.csv')
```

2. Use the `unlink()` method of this `Path` object to remove the directory entry. If this was the last directory entry for the data, then the space can be reclaimed by the OS:

```
>>> try:
...     input_path.unlink()
... except FileNotFoundError as ex:
...     print("File already deleted")
File already deleted
```

If the file does not exist, a `FileNotFoundError` is raised. In some cases, this exception needs to be silenced with the `pass` statement. In other cases, a warning message might be important. It's also possible that a missing file represents a serious error.

Additionally, we can rename a file using the `rename()` method of a `Path` object. We can create new soft links using the `symlink_to()` method. To create OS-level hard links, we need to use the `os.link()` function.

Finding all files that match a given pattern

The following are steps to find all files that match a given pattern:

1. Create the `Path` object from the input directory name. The `Path` class will properly parse the string to determine the elements of the path:

```
>>> directory_path = Path(options.file1).parent
>>> directory_path
PosixPath('/Users/slott/Documents/Writing/Python Cookbook/code')
```

2. Use the `glob()` method of the `Path` object to locate all files that match a given pattern. By default, this will not recursively walk the entire directory tree:

```
>>> list(directory_path.glob("ch08_r*.py"))
[PosixPath('/Users/slott/Documents/Writing/Python
Cookbook/code/ch08_r01.py'),
 PosixPath('/Users/slott/Documents/Writing/Python
Cookbook/code/ch08_r02.py'),
 PosixPath('/Users/slott/Documents/Writing/Python
Cookbook/code/ch08_r06.py'),
 PosixPath('/Users/slott/Documents/Writing/Python
Cookbook/code/ch08_r07.py'),
 PosixPath('/Users/slott/Documents/Writing/Python
Cookbook/code/ch08_r08.py'),
 PosixPath('/Users/slott/Documents/Writing/Python
Cookbook/code/ch08_r09.py'),
 PosixPath('/Users/slott/Documents/Writing/Python
Cookbook/code/ch08_r10.py')]
```

How it works...

Inside the OS, a path is a sequence of directories (a folder is a depiction of a directory). In a name such as `/Users/slott/Documents/writing`, the root directory, `/`, contains a directory named `Users`. This contains a subdirectory `slott`, which contains `Documents`, which contains `writing`.

In some cases, a simple string representation is used to summarize the navigation from root to directory through to the final target directory. The string representation; however, makes many kinds of path operations into complex string parsing problems.

The `Path` class definition simplifies many operations on pure paths. A pure `Path` may or may not reflect actual filesystem resources. Operations on `Path` include the following examples:

- Extract the parent directory, as well as a sequence of all enclosing directory names.
- Extract the final name, the stem of the final name, and the suffix of the final name.
- Replace the suffix with a new suffix or replace the entire name with a new name.
- Convert a string to a `Path`. And also convert a `Path` to a string. Many OS functions and parts of Python prefer to use filename strings.
- Build a new `Path` object from an existing `Path` joined with a string using the `/` operator.

A concrete `Path` represents an actual filesystem resource. For concrete `Paths`, we can do a number of additional manipulations of the directory information:

- Determine what kind of directory entry this 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, this includes information such as timestamps, permissions, ownership, size, and so on. We can also modify these things.
- We can unlink (or remove) the directory entry.

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` or `os.path` module.

There's more...

When we look at other file-related recipes in the rest of this chapter, we'll use `Path` objects to name the files. The objective is to avoid trying to use strings to represent paths.

The `pathlib` module makes a small distinction between Linux pure `Path` objects, and Windows pure `Path` objects. Most of the time, we don't care about the OS-level details of the path.

There are two cases where it can help to produce pure paths for a specific operating system:

- If we do development on a Windows laptop, but deploy web services on a Linux server, it may be necessary to use `PureLinuxPath`. This allows us to write test cases on the Windows development machine that reflects actual intended use on a Linux server.
- If we do development on a Mac OS X (or Linux) laptop, but deploy exclusively to Windows servers, it may be necessary to use `PureWindowsPath`.

We might have something like this:

```
>>> from pathlib import PureWindowsPath
>>> home_path = PureWindowsPath(r'C:\Users\slott')
>>> name_path = home_path / 'filename.ini'
>>> name_path
PureWindowsPath('C:/Users/slott/filename.ini')
>>> str(name_path)
'C:\\Users\\slott\\filename.ini'
```

Note that the `/` characters are normalized from Windows to Python notation when displaying the `WindowsPath` object. Using the `str()` function retrieves a path string appropriate for the Windows OS.

If we try this using the generic `Path` class, we'll get an implementation appropriate to the user's environment, which may not be Windows. By using `PureWindowsPath`, we've bypassed the mapping to the user's actual OS.

See also

- In the *Replacing a file while preserving the previous version* recipe, we'll look at how to leverage the features of a `Path` 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 5`, *User Inputs and Outputs*, we'll look at one very common way to get the initial string that will be used to create a `Path` object

Reading and writing files with context managers

Many programs will access external resources such as database connections, network connections, and OS files. It's important for a reliable, well-behaved program to release all external entanglements reliably and cleanly.

A program that raises an exception and eventually crashes can still properly release resources. This includes closing a file and being sure that any buffered data is properly written to the file.

This is particularly important for long-running servers. A web server may open and close many files. If the server did not close each file properly, then data objects might be left in memory, reducing the amount of room that can be used for ongoing web services. The loss of working memory appears like a slow leak. Eventually the server needs to be restarted, reducing availability.

How can we be sure that resources are acquired and released properly? How can we avoid resource leaks?

Getting ready

One common example of expensive and important resources is an external file. A file that has been opened for writing is also a precious resource; after all, we run programs to create useful output in the form of files. It's essential that the OS-level resources associated with a file be cleanly released by the Python application. We want to be sure that buffers are flushed and the file is properly closed no matter what happens inside the application.

When we use a context manager, we can be sure that the files being used by our application are handled properly. Specifically, the file will always be closed even when exceptions are raised during processing.

As an example, we'll use a script to collect some basic information about files in a directory. This can be used to detect file changes, the technique is often used to trigger processing when a file has been replaced.

We'll write a summary file that has the filename, modification date, size, and a checksum computed from the bytes in the file. We can then examine the directory and compare it with the previous state from the summary file. The description of a single file's details can be prepared by this function:

```
from types import SimpleNamespace
import datetime
from hashlib import md5

def file_facts(path):
    return SimpleNamespace(
        name = str(path),
        modified = datetime.datetime.fromtimestamp(
            path.stat().st_mtime).isoformat(),
        size = path.stat().st_size,
        checksum = md5(path.read_bytes()).hexdigest()
    )
```

This function gets a relative filename from the given `Path` object in the `path` parameter. We could also use the `resolve()` method to get the absolute pathname. The `stat()` method of a `Path` object returns a number of OS status values. The `st_mtime` value of the status is the last modified time. The expression `path.stat().st_mtime` gets the modification time for the file. This is used to create a complete `datetime` object. The `isoformat()` method then provides a standardized display of the modification time.

The value of `path.stat().st_size` is the file's current size. The value of `path.read_bytes()` is all of the bytes in the file, these are passed to the `md5` class to create a checksum using the MD5 algorithm. The `hexdigest()` function of the resulting `md5` object gives us a value that is sensitive enough to detect any single-byte change in the file.

We want to apply this to a number of files in a directory. If the directory is being used for example, files are being written frequently then it's possible that our analysis program might crash with an I/O exception while trying to read a file that's being written by a separate process.

We'll use a context manager to make sure the program provides good output even in the rare case that it crashes.

How to do it...

1. We'll be working with file paths, so it's important to import the `Path` class:

```
from pathlib import Path
```

2. Create a `Path` that identifies the output file:

```
summary_path = Path('summary.dat')
```

3. The `with` statement creates the `file` object, and assigns it to a variable, `summary_file`. It also uses this `file` object as the context manager:

```
with summary_path.open('w') as summary_file:
```

   We can now use the `summary_file` variable as an output file. No matter what exceptions are raised inside the `with` statement, the file will be properly closed, and all OS resources released.

The following statements will write information about files in the current working directory to the open summary file. These are indented inside the `with` statement:

```
base = Path(".")
for member in base.glob("*.py"):
    print(file_facts(member), file=summary_file)
```

This creates a `Path` for the current working directory and saves the object in the `base` variable. The `glob()` method of a `Path` object will generate all filenames that match the given pattern. The `file_facts()` function shown previously produces a namespace object that has useful information. We can print each summary to the `summary_file`.

We've omitted converting the facts to a more useful notation. It can slightly simplify subsequent processing if the data is serialized in JSON notation.

When the `with` statement finishes, the file will be closed. This will happen irrespective of any exception that might have been raised.

How it works...

The context manager object and the `with` statement work together to manage valuable resources. In this case, the file connection is a relatively expensive resource because it binds OS resources with our application. It's also precious because it's the useful output from the script.

When we write `with x:`, the object `x` is the context manager. A context manager object responds to two methods. These two methods are invoked by the `with` statement on the object provided. The significant events are as follows:

- `x.__enter__()` is evaluated at the beginning of the context.
- `x.__exit__(*details)` is evaluated at the end of the context. The `__exit__()` is guaranteed irrespective of any exceptions that might have been raised within the context. The exception details are provided to the `__exit__()` method. The context manager might want to behave differently if there was an exception.

File objects and several other kinds of objects are designed to work with this object manager protocol.

Here's the sequence of events that describe how the context manager is used:

1. Evaluate `summary_path.open('w')` to create a file object. This is saved to `summary_file`.
2. Evaluate `summary_file.__enter__()` as the with context starts.
3. Do the processing inside the `with` statement context. This will write several lines to the given file.
4. At the end of the `with` statement, evaluate `summary_file.__exit__()`. This will close the output file, and release all OS resources.
5. If an exception was raised inside the `with` statement and not handled, then reraise that exception now that the file is properly closed.

The file close operations are handled automatically by the `with` statement. They're always performed, even if there's an exception raised. This guarantee is essential to preventing resource leaks.

Some people like to quibble about the word *always*: they like to search for the very few situations where the context manager will not work properly. For example, there is a remote possibility that the entire Python runtime environment crashes; this will invalidate all of the language guarantees. If the Python context manager doesn't close the file properly, the OS will close the file, but the final buffer of data may be lost. There's an even more remote possibility that the entire OS crashes, or the hardware stops, or the computer is destroyed during a zombie apocalypse; the context manager won't close the files in these situations, either.

There's more...

Many database connections and network connections also work as context managers. The context manager guarantees that the connection is closed properly and the resources released.

We can use context managers for input files, also. The best practice is to use a context manager for all file operations. Most of the recipes in this chapter will use files and context managers.

In rare cases, we'll need to add context management capabilities to an object. The `contextlib` includes a function, `closing()`, which will call an object's `close()` method.

We can use this to wrap a database connection that lacks appropriate context manager capabilities:

```
from contextlib import closing
with closing(some_database()) as database:
    process(database)
```

This assumes that the `some_database()` function creates a connection to a database. This connection can't be directly used as a context manager. By wrapping the connection in the `closing()` function, we've added the necessary features to make this into a proper connection manager object so that we can be assured that the database is properly closed.

See also

- For more information on multiple contexts, see the *Using multiple contexts for reading and writing files* recipe

Replacing a file while preserving the previous version

We can leverage the power of `pathlib` to support a variety of filename manipulations. In the *Using pathlib to work with filenames* recipe, we looked at a few of the most common techniques of 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 yesterday's use of the `long_complex.py` application.
2. At time t_1 we start running the `long_complex.py` application. It begins overwriting the `output.csv` file. It is expected to finish normally at time t_3.
3. At time t_2, the application crashes. The partial `output.csv` file is useless. Worse, the valid file from time t_0 is not available either, since it was overwritten.

Clearly, we can make backup copies of files. This introduces an extra processing step. We can do better. What's a good approach to creating files that are fail-safe?

Getting ready

Fail-safe file output generally means that we don't overwrite the previous file. Instead, the application will create a new file using a temporary name. If the file was created successfully, then the old file can be replaced using a rename operation.

The goal is to create files in such a way that at any time prior to the rename, a crash will leave the original file in place. At any time subsequent to the rename, the new file is in place and is valid.

There are several ways to approach this. We'll show a variation that uses three separate files:

- The output file that will be overwritten eventually: `output.csv`.
- A temporary version of the file: `output.csv.tmp`. 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. Sometimes it will be in the `/tmp` filesystem.
- The previous version of the file: `name.out.old`. Any previous `.old` file will be removed as part of finalizing the output.

How to do it...

1. Import the `Path` class:

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

2. For demonstration purposes, we'll mock the argument parsing by providing the following `Namespace` object:

```
>>> from argparse import Namespace
>>> options = Namespace(
...     target='/Users/slott/Documents/Writing/Python
Cookbook/code/output.csv'
... )
```

   We've provided a mock value for the `target` command-line argument. This `options` object behaves like the options created by the `argparse` module.

3. Create the pure `Path` for the desired output file. This file doesn't exist yet, which is why this is a pure path:

```
>>> output_path = Path(options.target)
>>> output_path
PosixPath('/Users/slott/Documents/Writing/Python
Cookbook/code/output.csv')
```

4. Create the pure `Path` for a temporary output file. This will be used to create output:

```
>>> output_temp_path = output_path.with_suffix('.csv.tmp')
```

5. Write content to the temporary file. This is of course the heart of the application. It's often quite complex. For this example, we've shortened it to writing just one literal string:

```
>>> output_temp_path.write_text("Heading1,Heading2\r\n355,113\r\n")
```

Any failure here has no impact on the original output file; the original file hasn't been touched.

6. Remove any prior `.old file`:

```
>>> output_old_path = output_path.with_suffix('.csv.old')
>>> try:
...     output_old_path.unlink()
... except FileNotFoundError as ex:
...     pass # No previous file
```

Any failure at this point has no impact on the original output file.

7. If there's an existing file, rename it to become the `.old file`:

```
>>> output_path.rename(output_old_path)
```

Any failure after this will leave the `.old` file in place. This extra file can be renamed as part of a recovery process.

8. Rename the temporary file to be the new output file:

```
>>> output_temp_path.rename(output_path)
```

9. At this point, the file has been overwritten by renaming the temporary file. An `.old` file will be left around in case there's a need to roll back the processing to the previous state.

How it works...

This process involves three separate OS operations, an unlink, and two renames. This leads to a situation in which the `.old` file needs to 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 1 (the previous contents) and version 2 (the revised contents):

Time	Operation	.csv.old	.csv	.csv.tmp
t_0		version 0	version 1	
t_1	writing	version 0	version 1	in-process
t_2	close	version 0	version 1	version 2
t_3	unlink `.csv.old`		version 1	version 2
t_4	rename `.csv` to `.csv.old`	version 1		version 2
t_5	rename `.csv.tmp` to `.csv`	version 1	version 2	

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 backup copy, which can be used for recovery

Since none of these operations involved actually copying the files, they're all extremely fast and very reliable.

There's more...

In many cases, the output files involve optionally creating a directory based on timestamps. This can be handled gracefully by the `pathlib` module, also. We might, for example, have an archive directory that we'll put old files in:

```
archive_path = Path("/path/to/archive")
```

We may want to create date-stamped subdirectories for keeping temporary or working files:

```
import datetime
today = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
```

We can then do the following to define a working directory:

```
working_path = archive_path / today
working_path.mkdir(parents=True, exists_ok=True)
```

The `mkdir()` method will create the expected directory. Including the `parents=True` argument that assures that all parent directories will also be created. This can be handy the very first time an application is executed. The `exists_ok=True` is handy so that the existing directory can be reused without raising an exception.

The `parents=True` is not the default. With the default of `parents=False`, when a parent directory doesn't exist, the application will crash because the required file doesn't exist.

Similarly, the `exists_ok=True` is not the default. By default, if the directory exists, a `FileExistsError` exception is raised. Including options that make the operation silent when the directory exists.

Also, it's sometimes 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, we looked at the fundamentals of the `Path` class
- In `Chapter 9`, *Testing*, we'll look at some techniques for writing unit tests that can assure that parts of this will behave properly

Reading delimited files with the CSV module

One commonly used data format is CSV. We can easily generalize this to think of the comma as simply one of many candidate separator characters. We might have a CSV file that uses the `|` character as the separator between columns of data. This generalization makes CSV files particularly powerful.

How can we process data in one of the wide varieties of CSV formatting?

Getting ready

A summary of a file's content is called a schema. It's essential to distinguish between two aspects of the schema:

- **The Physical Format of the file**: For CSV, this means the file contains text. The text is organized into rows and columns. There will be a row separator character (or characters); there will also be a column separator character. Many spreadsheet products will use `,` as the column separator and the `\r\n` sequence of characters as the row separator. Other formats are possible, though, and it's easy to change the punctuation that separates columns and rows. The specific combination of punctuation is called the CSV dialect.
- **The Logical Layout of the data in the file**: This is the sequence of data columns that are present. There are several common cases for handling the logical layout in CSV files:
    - The file has one line of headings. This is ideal, and fits nicely with the way the CSV module works. The best headings are proper Python variable names.
    - 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.
    - If the file has no headings and the column positions aren't fixed, this is generally a serious problem. It can't easily be solved. Additional schema information is required; a separate list of column definitions, for example, can make the file useable.
    - The file has multiple lines of headings. In this case, we have to write special processing to skip past these lines. We will also have to replace complex headings with something more useful in Python.
    - An even more difficult case is where the file is not in proper **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. See the *Slicing and dicing a list* recipe in `Chapter 4`, *Built-in Data Structures – list, set, dict*, and *Using stacked generator expressions* recipe in online `Chapter 12`, *Functional And Reactive Programming Features* (link provided in `Preface`), for other recipes that work on normalizing data structures.

We'll look at a relatively simple 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 has four columns that need to be reformatted to create more useful information.

How to do it...

1. Import the `csv` module and the `Path` class:

   ```
   import csv
   ```

2. From `pathlib` import `PathExamine` from the data to confirm the following features:
   - The column separator characters: `','` are the default.
   - The row separator characters: `'\r\n'` are widely used in both Windows and Linux. This may be a feature of Excel, but it's quite common. Python's universal newlines feature means that the Linux standard `'\n'` will work just as well as a row separator.
   - The presence of a single-row heading. If not present, this information can be provided separately.

3. Create a `Path` object that identifies the file:

   ```
   data_path = Path('waypoints.csv')
   ```

4. Use the `Path` object to open the file in a `with` statement:

   ```
   with data_path.open() as data_file:
   ```

   For more information on the with statement, see the *Reading and writing files with context managers* recipe.

5. Create the CSV reader from the open file object. This is indented inside the `with` statement:

   ```
   data_reader = csv.DictReader(data_file)
   ```

6. Read (and process) the various rows of data. This is properly indented inside the `with` statement. For this example, we'll just print them:

```
for row in data_reader:
    print(row)
```

The output is a series of dictionaries that looks as follows:

```
{'date': '2012-11-27',
 'lat': '32.8321666666667',
 'lon': '-79.9338333333333',
 'time': '09:15:00'}
```

Since the row was transformed into a dictionary, the column keys are not in the original order. If we use `pprint()` from the `pprint` module the keys tend to get sorted into alphabetical order. We can now process the data by referring to `row['date']`. Using the column names is more descriptive than referring to the column by position: `row[0]` is hard to understand.

How it works...

The `csv` module handles physical format work of separating the rows from each other, and separating the columns within each row. The default rules assure that each input line is treated as a separate row, and 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, it is doubled. The second example row shows how the value `"blowing "like stink""` is encoded by doubling the quote characters when they are used inside a quoted column. These quoting rules mean that a CSV file can represent any combination of characters, including the row and column separator characters.

The values in a CSV file are always strings. A string value like `7331` may look like a number to us, but it's merely text when processed by the `csv` module. This makes the processing simple and uniform, but it can be awkward for a human user.

Some CSV data is exported from software such as databases or web servers. This data tends to be the easiest to work with because the various rows tend to be organized consistently.

When data is saved from a manually prepared spreadsheet, the data may reveal quirks of the desktop software's internal rules for data display. It's surprisingly common, for example, to have a column of data that is displayed as a date on the desktop software, but shows up as a simple 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 as a string. Ideally, this is done using ISO rules so that the date is represented in 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, but they're generally either Jan 1, 1900 or Jan 1, 1904.

There's more...

As we saw in the *Combining map and reduce transformations* recipe, there's often a pipeline of processing that includes cleansing and transformation of the source data. 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.

To transform the data into a more useful form, we'll use a two-part design. First, we'll define a row-level cleansing function. In this case, we'll update the row-level dictionary object by adding additional column-like values:

```
import datetime
def clean_row(source_row):
    source_row['lat_n']= float(source_row['lat'])
    source_row['lon_n']= float(source_row['lon'])
    source_row['ts_date']= datetime.datetime.strptime(
        source_row['date'],'%Y-%m-%d').date()
    source_row['ts_time']= datetime.datetime.strptime(
        source_row['time'],'%H:%M:%S').time()
    source_row['timestamp']= datetime.datetime.combine(
        source_row['ts_date'],
        source_row['ts_time']
        )
    return source_row
```

We've created new column values, `lat_n` and `lon_n`, which have proper floating-point values instead of strings. We've also parsed the date and time values to create `datetime.date` and `datetime.time` objects. We've also 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 of data. We can use `map(clean_row, reader)` or we can write a function that embodies this processing loop:

```
def cleanse(reader):
    for row in reader:
         yield clean_row(row)
```

This can be used to provide more useful data from each row:

```
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)
```

We've injected the `cleanse()` function to create a very small stack of transformation rules. The stack starts with the `data_reader`, and only has one other item in it. This is a good beginning. As the application software is expanded to do more computations, the stack will expand.

These cleansed and enriched rows look as follows:

```
{'date': '2012-11-27',
 'lat': '32.8321666666667',
 'lat_n': 32.8321666666667,
 'lon': '-79.9338333333333',
 'lon_n': -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)}
```

We've added columns such as `lat_n` and `lon_n`, which have proper numeric values instead of strings. We've also added `timestamp`, which has a full date-time value that can be used for simple computations of elapsed time between waypoints.

See also

- See the *Combining map and reduce transformations* recipe for more information on the idea of a processing pipeline or stack
- See the *Slicing and dicing a list* recipe in `Chapter 4`, *Built-in Data Structures – list, set, dict* and *Using stacked generator expressions* recipe in online `Chapter 12`, *Functional And Reactive Programming Features* (link provided in `Preface`), for more information on processing a CSV file that isn't in a proper 1NF

Reading complex formats using regular expressions

There are many file formats that 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 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 online `Chapter 12`, *Functional And Reactive Programming Features* (link provided in `Preface`), 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 used in this file. The `csv` module can't handle this complexity.

How can we process this kind of data with the elegant simplicity of a CSV file? Can we transform these irregular rows to a more regular data structure?

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's slightly easier to create a small class that behaves like the `DictReader` class.

The core feature of the reader is a function that will transform one line of text into a dict or tuple of individual field values. 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`, *Numbers, Strings, and Tuples*.

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'\[(\d+-\d+-\d+ \d+:\d+:\d+,\d+)\]'
...     '\s+(\w+)'
...     '\s+in'
...     '\s+([\w_\.]+):'
...     '\s+(.*)')
>>> pattern = re.compile(pattern_text)
```

The date-time stamp is various kinds of digits, hyphens, colons, and a comma; it's surrounded by `[` and `]`. We've had to use `\[` and `\]` to escape the normal meaning of `[` and `]` in a regular expression. The date stamp is followed by a severity level, which is a single run of characters. The characters `in` can be ignored; there are no `()`'s to capture the matching data. The module name is a sequence of letter characters, summarized by the character class `\w`, and also including _ and `.`. There's an extra `:` character after the module name that can also be ignored. Finally, there's a message that extends to the end of the line. We've wrapped the interesting data strings in `()` to capture each of these as part of the regular expression processing.

Note that we've also included the `\s+` sequence to quietly skip any number of space-like characters. It appears that the sample data all use a single space as the separator. However, when absorbing whitespace, using `\s+` seems to be a slightly more general approach because it permits extra spaces.

Here's how this pattern works:

```
>>> sample_data = '[2016-05-08 11:08:18,651] INFO in ch09_r09: Sample
Message One'
>>> match = pattern.match(sample_data)
>>> match.groups()
('2016-05-08 11:08:18,651', 'INFO', 'ch09_r09', 'Sample Message One')
```

We've provided a line of sample data. The match object, `match`, has a `groups()` method that returns each of the interesting fields. We can make this into a dictionary with named fields by using `(?P<name>...)` for each capture instead of simply `(...)`.

How to do it...

This recipe has two parts-defining a parse function for a single line, and using the parse function for each line of input.

Defining the parse function

Perform the following steps for defining the parse function:

1. Define the compiled regular expression object:

```
import re
pattern_text = (r'\[(?P<date>\d+-\d+-\d+ \d+:\d+:\d+,\d+)\]'
    '\s+(?P<level>\w+)'
    '\s+in\s+(?P<module>[\w_\.]+):'
    '\s+(?P<message>.*)')
pattern = re.compile(pattern_text)
```

   We've used the `(?P<name>...)` regular expression construct to provide names for each group that's captured. The resulting dictionary will be identical with the results of `csv.DictReader`.

2. Define a function that accepts a line of text as an argument:

```
def log_parser(source_line):
```

3. Apply the regular expression to create a match object. We've assigned it to the `match` variable:

```
match = pattern.match(source_line)
```

4. If the match object is `None`, the line did not match the pattern. This line may be skipped silently. In some applications, it should be logged in some way to provide information useful for debugging or enhancing the application. It may also make sense to raise an exception for an input line that cannot be parsed:

```
if match is None:
    raise ValueError(
        "Unexpected input {0!r}".format(source_line))
```

5. Return a useful data structure with the various pieces of data from this input line:

```
return match.groupdict()
```

This function can be used to parse each line of input. The text is transformed into a dictionary with field names and values.

Using the parse function

1. Import the `csv` module and the `Path` class:

```
import csv
```

2. From `pathlib` import `PathCreate`, the `Path` object that identifies the file:

```
data_path = Path('sample.log')
```

3. Use the `Path` object to open the file in a `with` statement:

```
with data_path.open() as data_file:
```

For more information on the `with` statement, see the *Reading and writing files with context managers* recipe.

4. Create the log file parser from the open file object, `data_file`. In this case, we'll use `map()` to apply the parser to each line from the source file:

```
data_reader = map(log_parser, data_file)
```

5. Read (and process) the various rows of data. For this example, we'll just print them:

```
for row in data_reader:
    pprint(row)
```

The output is a series of dictionaries that looks as follows:

```
{'date': '2016-05-08 11:08:18,651',
 'level': 'INFO',
 'message': 'Sample Message One',
 'module': 'ch09_r09'}
{'date': '2016-05-08 11:08:18,651',
```

```
 'level': 'DEBUG',
 'message': 'Debugging',
 'module': 'ch09_r09'}
{'date': '2016-05-08 11:08:18,652',
 'level': 'WARNING',
 'message': 'Something might have gone wrong',
 'module': 'ch09_r09'}
```

We can do more meaningful processing on these dictionaries 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 typical of files that are in First Normal Form. 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, the format requires a complex regular expression to parse.

In our log file example, the timestamp has 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 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 the format and number of attributes, breaking one of the First Normal Form assumptions.

In the case of inconsistent data, we'll have to create more sophisticated parsers. This may include complex filtering rules to separate out the various kinds of information that may appear in a web server log file. It may involve parsing part of the line to determine which regular expression must be used to parse the rest of the line.

We've relied on using the `map()` higher-order function. This applies the `log_parse()` function to each line of the source file. The direct simplicity of this provides some assurance that the number of data objects created will precisely match the number of lines in the log file.

We've generally followed the design pattern from the *Reading delimited files with the cvs module* recipe, so that reading a complex log is nearly identical with reading a simple CSV file. Indeed, we can see that the primary difference lies in one line of code:

```
data_reader = csv.DictReader(data_file)
```

As compared to:

```
data_reader = 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.

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 the data in CSV format for later processing.

Some of this is similar to the *Using multiple contexts for reading and writing files* recipe, which also shows multiple open contexts. We'll read from one file and write to another file.

The file writing process looks as follows:

```
import csv
data_path = Path('sample.log')
target_path = data_path.with_suffix('.csv')
with target_path.open('w', newline='') as target_file:
    writer = csv.DictWriter(
        target_file,
        ['date', 'level', 'module', 'message']
        )
    writer.writeheader()

    with data_path.open() as data_file:
        reader = map(log_parser, data_file)
        writer.writerows(reader)
```

The first portion of this script defines a CSV writer for a given file. The path for the output file, `target_path`, is based on the input name, `data_path`. The suffix changed from the original filename's suffix to `.csv`.

This 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. A sequence of column headings is provided. These must match the keys used to write each row to the file. We can see that these headings match the `(?P<name>...)` parts of the regular expression that produces the data.

The `writeheader()` 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 be a kind of 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 function to the `writerows()` method of the writer. The `writerows()` method will consume all of the data produced by the `reader()` function. This will, in turn, consume all of the rows produced by the open file.

We don't need to write any explicit `for` statements to assure that all of the input rows are processed. The `writerows()` function makes this guarantee.

The output file looks as follows:

```
    date,level,module,message
    "2016-05-08 11:08:18,651",INFO,ch09_r09,Sample Message One
    "2016-05-08 11:08:18,651",DEBUG,ch09_r09,Debugging
    "2016-05-08 11:08:18,652",WARNING,ch09_r09,Something might have gone
wrong
```

The file has been transformed from the rather complex input format to a simpler CSV format.

See also

- The *Writing generator functions with the yield statement* recipe in online `Chapter 12`, *Functional And Reactive Programming Features* (link provided in `Preface`), shows other processing of this log format
- In the *Reading delimited files with the CSV module* recipe, we look at other applications of this general design pattern
- In the *Upgrading CSV from Dictreader to namedtuple reader* and *Upgrading CSV from Dictreader to namespace reader* recipes we'll look at even more sophisticated processing techniques

Reading JSON documents

The JSON notation for serializing data is very popular. For details, see `http://json.org`. Python includes the `json` module to serialize and deserialize data in this notation.

JSON documents are used widely by JavaScript applications. It's common to exchange data between Python-based servers and JavaScript-based clients using documents in JSON notation. These two tiers of the application stack communicate via JSON documents sent via the HTTP protocol. Interestingly, a data persistence layer may also use HTTP protocol and JSON notation.

How do we use the `json` module to parse JSON data in Python?

Getting ready

We've gathered some sailboat racing results in `race_result.json`. This file has information on teams, legs, and the orders in which the various teams finish the legs of the race.

In many cases, there are null values when a boat did not start, did not finish, or was disqualified from the race. In those cases, the finish position is assigned a score of one more than the last position. If there are seven boats, then the team is given eight points. This is a hefty penalty.

The data has the following schema. There are two fields within the overall document:

- `legs`: Array of strings that show starting port and ending port.
- `teams`: Array of objects with details about each team. Within each team object, there are several fields of data:
    - `name`: String team name.
    - `position`: Array of integers and nulls with position. The order of items in this array matches the order of 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. There were a total of seven teams in this particular race.

The JSON-formatted data looks 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.

Not all JSON structures are simply Python objects. Interestingly, the JSON document has a null item, which maps to Python's `None` object. The meaning is similar, but the syntax is different.

Also, 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.

How to do it...

1. Import the `json` module:

```
>>> import json
```

2. Define a `Path` object that identifies the file to be processed:

```
>>> from pathlib import Path
>>> source_path = Path("code/race_result.json")
```

   The `json` module doesn't currently work directly with `Path` objects. Consequently, we'll read the content as a big block of text and process that text object.

3. Create a Python object by parsing the JSON document:

```
>>> document = json.loads(source_path.read_text())
```

We've used `source_path.read_text()` to read the file named by the `Path`. We provided this string to the `json.loads()` function for parsing.

Once we've parsed the document to create a Python dictionary, we can see the various pieces. For example, the field `teams` has all of the results for each team. It's an array, and item 0 in that array is the first team.

The data for each team will be a dictionary with two keys: `name` and `position`. We can combine the various keys to get the name of the first team:

```
>>> document['teams'][0]['name']
'Abu Dhabi Ocean Racing'
```

We can look inside the `legs` field to see the names of each leg of the race:

```
>>> document['legs'][5]
'ITAJAÍ – NEWPORT'
```

Note that the JSON source file included a `'\u00cd'` Unicode escape sequence. This was parsed properly and the Unicode output shows the proper Í character.

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"}`. Unlike Python, JSON only uses `"` for string quotation marks. JSON notation is intolerant of an extra , at the end of the dictionary value. Other than this, the two notations are similar.
- **Arrays that map to Python lists**: JSON syntax uses `[item, ...]`, which looks like Python. JSON is intolerant of extra , at the end of the array value.
- **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.

There is no provision for any other kinds of data. This means that Python programs must convert complex Python objects to 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 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, JSON may not be the best notation for preserving a huge volume of information.

There are two additional problems that we often have to tackle:

- Serializing complex objects so that we can write them to files
- Deserializing complex objects from the text that's read from a file

When we represent a Python object's state as a string of text characters, we've serialized the object. Many Python objects need to be saved in a file or transmitted to another process. These kinds of transfers require a representation of object state. We'll look at serializing and deserializing separately.

Serializing a complex data structure

We can also create JSON documents from Python data structures. Because Python is extremely sophisticated and flexible, we can easily create Python data structures that cannot possibly be represented in JSON.

The serialization to JSON works out the best if we create Python objects that are limited to simple `dict`, `list`, `str`, `int`, `float`, `bool`, and `None` values. If we're careful, we can build objects that serialize rapidly and can be used widely by a number of programs written in different languages.

None of these types of values involve Python `sets`, or other class definitions. This means that we're often forced to convert complex Python objects into dictionaries to represent them in a JSON document.

As an example, let's assume we've analyzed some data and created a resulting `Counter` object:

```
>>> import random
>>> random.seed(1)
>>> from collections import Counter
>>> colors = (["red"]*18)+(["black"]*18)+(["green"]*2)
>>> data = Counter(random.choice(colors) for _ in range(100))
Because this data is - effectively - a dict, we can serialie this very
easily into JSON:
>>> print(json.dumps(data, sort_keys=True, indent=2))
{
  "black": 53,
  "green": 7,
  "red": 40
}
```

We've dumped the data in JSON notation, with the keys sorted into order. This assures consistent output. The indent of two will show each `{}` object and each `[]` array indented visually to make it easier to see the document's structure.

We can write this to a file with a relatively simple operation:

```
output_path = Path("some_path.json")
output_path.write_text(
    json.dumps(data, sort_keys=True, indent=2))
```

When we reread this document, we will not get a `Counter` object from the JSON load operation. We'll only get a dictionary instance. This is a consequence of JSON's reduction to very simple values.

One commonly-used data structure that doesn't serialize easily is a `datetime.datetime` object. Here's what happens when we try:

```
>>> import datetime
>>> example_date = datetime.datetime(2014, 6, 7, 8, 9, 10)
>>> document = {'date': example_date}
```

We've created a simple document that has a single field. The value of the field is a `datetime` instance. What happens when we try to serialize this in JSON?

```
>>> json.dumps(document)
Traceback (most recent call last):
  ...
TypeError: datetime.datetime(2014, 6, 7, 8, 9, 10) is not JSON serializable
```

This shows that objects that cannot be serialized will raise a `TypeError` exception. Avoiding this exception can done in one of two ways. We can either convert the data before building the document, or we can add a hook to the JSON serialization process.

One technique is to convert the `datetime` object into a string prior to serializing it as JSON:

```
>>> document_converted = {'date': example_date.isoformat()}
>>> json.dumps(document_converted)
'{"date": "2014-06-07T08:09:10"}'
```

This uses the ISO format for dates to create a string that can be serialized. An application that reads this data can then convert the string back into a `datetime` object.

The other technique for serializing complex data is to provide a default function that's used automatically during serialization. This function must convert a complex object to something that can be safely serialized. Often it will create a simple dictionary with string and numeric values. It might also create a simple string value:

```
>>> def default_date(object):
...     if isinstance(object, datetime.datetime):
...         return example_date.isoformat()
...     return object
```

We've defined a function, `default_date()`, which will apply special conversion rules to `datetime` objects. These will be massaged into string objects that can be serialized by the `json.dumps()` function.

We provide this function to the `dumps()` function using the `default` parameter, as follows:

```
>>> document = {'date': example_date}
>>> print(
...     json.dumps(document, default=default_date, indent=2))
{
  "date": "2014-06-07T08:09:10"
}
```

In any given application, we'll need to expand this function to handle any of the more complex Python objects that we might want to serialize in JSON notation. If there are a large number of very complex data structures, we often want a somewhat more general solution than meticulously converting each object to something serializable. There are a number of design patterns for including type information along with serialized details of an object's state.

Deserializing a complex data structure

When deserializing JSON to create Python objects, there's another hook that can be used to convert data from a JSON dictionary into a more complex Python object. This is called the `object_hook` and it is used during `json.loads()` processing to examine each complex object to see if something else should be created from that dict.

The function we provide will either create a more complex Python object, or it will simply leave the dict alone:

```
>>> def as_date(object):
...     if 'date' in object:
...         return datetime.datetime.strptime(
...             object['date'], '%Y-%m-%dT%H:%M:%S')
...     return object
```

This function will check each object that's decoded to see if the object has a field named `date`. If it does, the value of the entire object is replaced with a `datetime` object.

We provide a function to the `json.loads()` function as follows:

```
>>> source= '''{"date": "2014-06-07T08:09:10"}'''
>>> json.loads(source, object_hook=as_date)
datetime.datetime(2014, 6, 7, 8, 9, 10)
```

This parses a very small JSON document that meets the criteria for containing a date. The resulting Python object is built from the string value found in the JSON serialization.

In a larger context, this particular example of handling dates isn't ideal. The presence of a single `'date'` field to indicate a date object could lead to problems with more complex objects being de-serialized using this `as_date()` function.

A more general approach would either look for something unique and non-Python like, such as `'$date'`. An additional feature would confirm that the special indicator was the only key for the object. When these two criteria were met, then the object could be processed specially.

We may also want to design our application classes to provide additional methods to help with serialization. A class might include a `to_json()` method that will serialize the objects in a uniform way. This method might provide class information. It can avoid serializing any derived attributes or computed properties. Similarly, we might need to provide a static `from_json()` method that can be used to determine if a given dictionary object is actually an instance of the given class.

See also

- The *Reading HTML documents* recipe will show how we prepared this data from an HTML source

Reading XML documents

The XML markup language is widely used to organize data. 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 `<tag>` and `</tag>` constructs that define the structure of the data. The overall file includes the content plus the XML markup text.

Because the markup is intermingled with our text, there are some additional syntax rules that must be used. In order to include the < character in our data, we'll use XML character entity references to avoid confusion. We use `<` to be able to include < in our text. Similarly, `>` is used instead of >, `&` is used instead of `&`, and `"` is also used to embed a `"` in an attribute value.

A document, then, will have items as follows:

```
<team><name>Team SCA</name><position>...</position></team>
```

Most XML processing allows additional `\n` and space characters in the XML to make the structure more obvious:

```
<team>
    <name>Team SCA</name>
    <position>...</position>
</team>
```

In general, content is surrounded by the tags. The overall document forms a large, nested collection of containers. Viewed another way, the document forms a tree with a root tag that contains all of the other tags and their embedded content. Between tags, there is additional content entirely whitespace in this example that will be ignored.

It's very, very difficult to parse this with regular expressions. We need more sophisticated parsers to handle the nested syntax.

There are two binary libraries that are available for parsing XML-SAX and Expat. Python includes `xml.sax` and `xml.parsers.expat` to exploit these two modules.

In addition to these, there's a very sophisticated set of tools in the `xml.etree` package. We'll focus on using the `ElementTree` module to parse and analyze XML documents.

How do we use the `xml.etree` module to parse XML data in Python?

Getting ready

We've gathered some sailboat racing results in `race_result.xml`. This file has information on teams, legs, and the orders in which the various teams finished each leg.

In many cases, there are empty values when a boat did not start, did not finish, or was disqualified from the race. In those cases, the score will be one more than the number of boats. If there are seven boats, then the team is given eight points. This is a hefty penalty.

The root tag is the `<results>` document. This has the following schema:

- The `<legs>` tag contains individual `<leg>` tags that name each leg of the race. The leg names 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.

The data looks as follows:

```
<?xml version="1.0"?>
<results>
    <teams>
            <team>
                    <name>
                            Abu Dhabi Ocean Racing
                    </name>
                    <position>
                            <leg n="1">
                                    1
                            </leg>
                            <leg n="2">
                                    3
                            </leg>
                            <leg n="3">
                                    2
                            </leg>
                            <leg n="4">
                                    2
                            </leg>
                            <leg n="5">
                                    1
                            </leg>
                            <leg n="6">
                                    2
                            </leg>
                            <leg n="7">
                                    5
                            </leg>
                            <leg n="8">
                                    3
                            </leg>
                            <leg n="9">
```

```
                                        5
                                    </leg>
                                </position>
                            </team>
                            ...
                        </teams>
                        <legs>
                        ...
                        </legs>
                    </results>
```

We've only shown the first team. There were a total of seven teams in this particular race.

In XML notation, the application data shows up in two kinds of places. Between tags; for example, `<name>Abu Dhabi Ocean Racing</name>`. The tag is `<name>`, the text between `<name>` and `</name>` is the value of this tag.

Also, data shows up as an attribute of a tag. For example, in `<leg n="1">`. The tag is `<leg>`; the tag has an attribute, `n`, with a value of `1`. A tag can have an indefinite number of attributes.

The `<leg>` tags include the leg number given as an attribute, `n`, and the position in the leg given as the text inside the tag. The general approach is to put important data inside the tags, and supplemental, or clarifying data in the attributes. The line between the two is very blurry.

XML permits a **mixed content model**. This reflects the case where XML is mixed in with text, there will be text inside and outside XML tags. Here's an example of mixed content:

```
<p>This has <strong>mixed</strong> content.</p>
```

Some of the text is inside the `<p>` tag, and some of the text is inside the `<strong>` tag. The content of the `<p>` tag is a mixture of text and tags with more text.

We'll use the `xml.etree` module to parse the data. This involves reading the data from a file and providing it to the parser. The resulting document will be rather complex.

We have not provided a formal schema definition for our sample data, nor have we provided a **Document Type Definition** (**DTD**). This means that the XML defaults to mixed content mode. Furthermore, the XML structure can't be validated against the schema or DTD.

How to do it...

1. We'll need two modules—`xml.etree` and `pathlib`:

```
>>> import xml.etree.ElementTree as XML
>>> from pathlib import Path
```

   We've changed the `ElementTree` module name name to `XML` to make it slightly easier to type. It's also common to rename this to something like `ET`.

2. Define a `Path` object that locates the source document:

```
>>> source_path = Path("code/race_result.xml")
```

3. Create the internal `ElementTree` version of the document by parsing the source file:

```
>>> source_text = source_path.read_text(encoding='UTF-8')
>>> document = XML.fromstring(source_text)
```

The XML parser doesn't readily work with `Path` objects. We've elected to read the text from the `Path` object and then parse that text.

Once we have the document, we can then search it for the relevant pieces of data. In this example, we'll use the `find()` method to locate the first instance of a given tag:

```
>>> teams = document.find('teams')
>>> name = teams.find('team').find('name')
>>> name.text.strip()
'Abu Dhabi Ocean Racing'
```

In this case, we located the `<teams>` tag, and then found the first instance of the `<team>` tag inside that list. Within the `<team>` tag, we located the first `<name>` tag to get the value of the team's name.

Because XML is a mixed content model, all of the `\n`, `\t`, and space characters in the content are perfectly preserved in the data. We rarely want any of this whitespace, and it makes sense to use the `strip()` method to remove all extraneous characters before and after the meaningful content.

How it works...

The XML parser modules transform XML documents into fairly complex objects based on the document object model. In the case of the `etree` module, the document will be built from `Element` objects that generally represent tags and text.

XML also includes processing instructions and comments. These are commonly ignored by many XML processing applications.

Parsers for XML often have two levels of operation. At the bottom level, they recognize events. The events that are found by the parser include element starts, element ends, comment starts, comment ends, runs of text, and similar lexical objects. At the higher level, the events are used to build the various `Elements` of the document.

Each `Element` instance has a tag, text, attributes, and a tail. The tag is the name inside the `<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.

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:

```
<name>Team SCA</name>
<position>...</position>
```

There's a `\n` character after the closing `</name>` tag and before the opening of the `<position>` tag. This is the tail of the `<name>` tag. The tail values can be important when working with a mixed content model. The tail values are generally whitespace when working in a non-mixed content model.

There's more...

Because we can't trivially translate an XML document to a Python dictionary, we need a handy way to search through the document content. The `ElementTree` module 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 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
```

We've looked for the top-level `<teams>` tags. Within that tag, we want `<team>` tags. Within those tags, we want the `<name>` tags. This will search for all instances of this nested tag structure.

We can search for attribute values, also. This can make it handy to find how all teams did on a particular leg of the race. The data is found in the `<leg>` tag within the `<position>` tag for each team.

Furthermore, each `<leg>` has an attribute value of n that shows which of the race legs it represents. Here's how we can use this to extract specific data from the XML document:

```
>>> for tag in document.findall("teams/team/position/leg[@n='8']"):
...     print(tag.text.strip())
3
5
7
4
6
1
2
```

This shows us the finish position of each team on leg 8 of the race. We're looking for all tags with `<leg n="8">` and displaying the text within that tag. We have to match these values with the team names to see that Team SCA finished first, and Dongfeng Race Team finished last on this leg.

See also

- The *Reading HTML documents* recipe 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 markup. A browser renders the data very nicely. How can we parse this data to extract the meaningful content from the displayed web page?

We can use the standard library `html.parser` module, but it's not helpful. It only provides low-level lexical scanning information, but doesn't provide a high-level data structure that describes the original web page.

We'll use the Beautiful Soup module to parse HTML pages. This is available from the **Python Package Index** (**PyPI**). See `https://pypi.python.org/pypi/beautifulsoup4`.

This must be downloaded and installed to be useful. Generally, the `pip` command does this job very nicely.

Often, this is as simple as the following:

```
pip install beautifulsoup4
```

For Mac OS X and Linux users, the `sudo` command is required to escalate the user's privileges:

```
sudo pip install beautifulsoup4
```

This will prompt for the user's password. The user must be able to elevate themselves to have root privileges.

In the rare case that you have multiple versions of Python, be sure to use the matching version of pip. In some cases, we might have to use the following:

```
sudo pip3.5 install beautifulsoup4
```

Use the `pip` that goes with Python 3.5.

Getting ready

We've gathered some sailboat racing results in `Volvo Ocean Race.html`. This file has information on teams, legs, and the order in which the various teams finished each leg. It's scraped from the Volvo Ocean Race website, and it looks wonderful when opened in a browser.

HTML notation is very similar to XML. The content is surrounded by `<tag>` marks that show the structure and presentation of the data. HTML predates XML, and the XHTML standard reconciles the two Browsers; however, must be tolerant of older HTML and even improperly structured HTML. The presence of damaged HTML can make it difficult to analyze data from the World Wide Web.

HTML pages include a great deal of overhead. There are often vast code and style sheet sections, as well as invisible metadata. The content may be surrounded by advertising and other information. Generally, an HTML page has the following overall structure:

```
<html>
    <head>...</head>
    <body>...</body>
</html>
```

Within the `<head>` tag there will be links to JavaScript libraries, and links to **Cascading Style Sheet** (**CSS**) documents. These are generally used to provide interactive features and define the presentation of the content.

The bulk of the content is in the `<body>` tag. Many web pages are very busy and provide a tremendously complex mix of content. The design of web pages is a sophisticated art, and the content is designed to look good on most browsers. It can be difficult to track down the relevant data on a web page, because the focus is on how people see it more than how automated tools can process it.

In this case, the race results are in an HTML `<table>` tag, making them easy to find. What we see is the following overall structure to the relevant content in the page:

```
<table>
    <thead>
        <tr>
            <th>...</th>
            ...
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>...</td>
```

```
                ...
            </tr>
            ...
        </tbody>
    </table>
```

The <thead> tag includes the column titles for the table. There's a single table row tag, <tr>, with table heading, <th>, tags that include the content. The content has two parts; the essential display is a number for each leg of the race. This is the content of the tag. In addition to the displayed content, there's also an attribute value that's used by a JavaScript function. This attribute value is displayed when the cursor hovers over a column heading. The JavaScript function pops up the leg name.

The <tbody> tag includes the team name and the results for each race. The table row (<tr>) contains the details for each team. The team name (and graphic and overall finish rank) is shown in the first three columns of table data, <td>. The remaining columns of table data contain the finish position for a given leg of the race.

Because of the relative complexity of sailboat racing, there are additional notes in some of the table data cells. These are included as attributes that are used to provide supplemental data on the reason for the cell's value. In some cases, teams did not start a leg, or did not finish a leg, or retired from a leg.

Here's a typical <tr> row from the HTML:

```
    <tr class="ranking-item">
        <td class="ranking-position">3</td>
        <td class="ranking-avatar">
            <img src="..."> </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-></td>
        <td class="ranking-number">1</td>
        <td class="ranking-number">4</td>
        <td class="ranking-number">7</td>
        <td class="ranking-number">4</td>
        <td class="ranking-number total">33<span
class="asterix">*</span></td>
    </tr>
```

The `<tr>` tag has a class attribute that defines the style for this row. The CSS provides the style rules for this class of data. The `class` attribute on this tag helps our data gathering application locate the relevant content.

The `<td>` tags also have class attributes that define the style for the individual cells of data. In this case, class information clarifies what the content of the cell means.

One of the cells has no content. That cell has an attribute of `data-title`. This is used by a JavaScript function to display additional information in the cell.

How to do it...

1. We'll need two modules: bs4 and pathlib:

```
>>> from bs4 import BeautifulSoup
>>> from pathlib import Path
```

   We've only imported the `BeautifulSoup` class from the `bs4` module. This class will provide all of the features required to parse and analyze HTML documents.

2. Define a `Path` object that names the source document:

```
>>> source_path = Path("code/Volvo Ocean Race.html")
```

3. Create the soup structure from the HTML content. We'll assign it to a variable, `soup`:

```
>>> with source_path.open(encoding='utf8') as source_file:
...     soup = BeautifulSoup(source_file, 'html.parser')
```

We've used a context manager to access the file. As an alternative we could simply read the content with `source_path.read_text(encodig='utf8')`. This works as well as providing an open file to the `BeautifulSoup` class.

The soup structure in the variable `soup` can then be processed to locate the various pieces of content. For example, we can extract the leg details as follows:

```
def get_legs(soup)
    legs = []
    thead = soup.table.thead.tr
    for tag in thead.find_all('th'):
        if 'data-title' in tag.attrs:
            leg_description_text = clean_leg(tag.attrs['data-title'])
            legs.append(leg_description_text)
```

```
        return legs
```

The expression `soup.table.thead.tr` will find the first `<table>` tag. Within that, the first `<thead>` tag; and within that, the first `<tr>` tag. We assigned this `<tr>` tag to a variable named, perhaps misleadingly, `thead`. We can then do a `findall()` to locate all `<th>` tags within this container.

We'll check each tag's attributes to locate the `data-title` attribute values. This will have the leg name information. The leg name content looks as follows:

```
<th tooltipster data->LEG 1</th>
```

The `data-title` attribute value includes some additional HTML markup within the value. This is not a standard part of HTML and the `BeautifulSoup` parser doesn't look for this HTML within an attribute value.

We have a small bit of HTML to parse, so we can create a small `soup` object just to parse that piece of text:

```
def clean_leg(text):
    leg_soup = BeautifulSoup(text, 'html.parser')
    return leg_soup.text
```

We create a small `BeautifulSoup` object from just the value of the `data-title` attribute. This soup will have information about the tag, `<strong>`, and the text. We used the text attribute to get all of the text without any tag information.

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.

Generally, we're interested in the tags that contain the string content of the web page. These are objects of the `Tag` and `NavigableString` classes.

Each `Tag` instance has a name, string, and attributes. The name is the word inside the < and >. 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. This means that a tag can contain child tags in addition to navigable text. The text is mixed, it can be inside as well as outside any of the child tags. When looking at the children of a given tag, there will be a sequence of tags and text freely intermixed.

One of the most common features of HTML are small blocks of navigable text that contain only newline characters. When we have a soup like this:

```
<tr>
    <td>Data</td>
</tr>
```

There are three children within the `<tr>` tag. Here's a display of the children of this tag:

```
>>> example = BeautifulSoup('''
...     <tr>
...         <td>data</td>
...     </tr>
... ''', 'html.parser')
>>> list(example.tr.children)
['\n', <td>data</td>, '\n']
```

The two newline characters are peers to the `<td>` tag, and are preserved by the parser. This is navigable text that surrounds the child tag.

The `BeautifulSoup` parser depends on another, lower-level process. The lower-level process can be the built-in `html.parser` module. There are alternatives that can be installed, also. The `html.parser` is easiest to use and covers the most common use cases. There are alternatives available, the Beautiful Soup documentation lists the other low-level parsers that can be used to solve particular web parsing problems.

The lower-level parser recognizes events; these include element starts, element ends, comment starts, comment ends, runs of text, and similar lexical objects. At the higher level, the events are used to build the various objects of the Beautiful Soup document.

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:

- All tags except a special root `[document]` container will have a parent. The top `<html>` tag will often be the only child of the root document container.
- The `parents` attribute is a generator for all parents of a tag. It's a path through the hierarchy to a given tag.
- All `Tag` objects can have children. A few tags such as `<img/>` and `<hr/>` have no children. The `children` attribute is a generator that yields the children of a tag.
- A tag with children may have multiple levels of tags under it. The overall `<html>` tag, for example, has the entire document as descendants. The `children` attribute has the immediate children; the `descendants` attribute generates all children of children.
- A tag can also have siblings, which are other tags within the same container. Since the tags have a defined order, there's a `next_sibling` and `previous_sibling` attribute to help step through the peers of a tag.

In some cases, a document will have a generally straight-forward organization and a simple search by the `id` attribute or `class` attribute will find the relevant data. Here's a typical search for a given structure:

```
>>> 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`. 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 any other tabular data on a web page.

Here's the parents of this `<table>` tag:

```
>>> list(tag.name for tag in ranking_table.parents)
['section', 'div', 'div', 'div', 'div', 'body', 'html', '[document]']
```

We've displayed just the tag name for each parent above the given `<table>`. Note that there are four nested `<div>` tags that wrap the `<section>` that contains the `<table>`. Each of these `<div>` tags likely has a different class attribute to properly define the content and the style for the content.

The `[document]` is the overall `BeautifulSoup` container that holds the various tags that were parsed. This is displayed distinctively to emphasize that it's not a real tag, but a container for the top-level `<html>` tag.

See also

- The *Reading JSON documents* and *Reading XML documents* recipes both use similar data. The example data was created for them by scraping the HTML page using these techniques.

Upgrading CSV from DictReader to namedtuple reader

When we read data from a CSV format file, we have two general choices for the resulting data structure:

- When we use `csv.reader()`, each row becomes a simple list of column values.
- When we use `csv.DictReader`, each row becomes a dictionary. By default, the contents of the first row become the keys for the row dictionary. The alternative is to provide a list of values that will be used as the keys.

In both cases, referring to data within the row is awkward because it involves rather complex-looking syntax. When we use a `csv` reader, we must use `row[2]`: the semantics of this are completely obscure. When we use a `DictReader`, we can use `row['date']`, which is less obscure, but is still a lot of typing.

In some real-world spreadsheets the column names are impossibly long strings. It's hard to work with `row['Total of all locations excluding franchisees']`.

What can we do to replace complex syntax with something simpler?

Getting ready

One way to improve the readability of programs that work with spreadsheets is to replace a list of columns with a `namedtuple` object. This provides easy-to-use names defined by the `namedtuple` instead of the possibly haphazard column names in the `.csv` file.

More importantly, it permits much nicer syntax for referring to the various columns. In addition to `row[0]`, we can also use `row.date` to refer to a column named `date`.

The column names (and the data types for each column) are part of the schema for a given file of data. In some CSV files the first line of the column titles is a schema for the file. This schema is limited, it provides only attribute names; the data types aren't known and have to be treated as strings.

This points to two reasons for imposing an external schema on the rows of a spreadsheet:

- We can supply meaningful names
- We can perform data conversions where necessary

We'll look at a relatively simple CSV file that has some real-time data recorded from the log of a sailboat. This is the `waypoints.csv` file, and 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 data has four columns. Two of the columns are the latitude and longitude of the waypoint. It has a column with the date and the time as separate values. This isn't ideal, and we'll look at various data cleansing steps separately.

In this case, the column titles happen to be valid Python variable names. This is rare, but it can lead to a slight simplification. We'll look at the alternatives in the following section.

The most important step is to gather the data as `namedtuples`.

How to do it...

1. Import the modules and definitions required. In this case, they will be from `collections`, `csv`, and `pathlib`:

```
from collections import namedtuple
from pathlib import Path
import csv
```

2. Define the `namedtuple` that matches the actual data. In this case, we've called it `Waypoint` and provided names for the four columns of data. In this example, the attributes happen to match the column names; it's not a requirement that the names match:

```
Waypoint = namedtuple('Waypoint', ['lat', 'lon', 'date', 'time'])
```

3. Define the `Path` object that refers to the data:

```
waypoints_path = Path('waypoints.csv')
```

4. Create the processing context for the open file:

```
with waypoints_path.open() as waypoints_file:
```

5. Define a CSV reader for the data. We'll call this a raw reader. In the long run, we'll follow the *Using stacked generator expressions* recipe, and *Use a stack of generator expressions* to cleanse and filter the data recipe in online `Chapter 12`, *Functional And Reactive Programming Features* (link provided in `Preface`):

```
raw_reader = csv.reader(waypoints_file)
```

6. Define a generator that builds `Waypoint` objects from tuples of input data:

```
waypoints_reader = (Waypoint(*row) for row in raw_reader)
```

We can now process rows using the `waypoints_reader` generator expression:

```
for row in waypoints_reader:
    print(row.lat, row.lon, row.date, row.time)
```

The `waypoints_reader` object will also provide the heading row, which we want to ignore. We'll look at filtering and conversion in the following section.

The expression `(Waypoint(*row) for row in raw_reader)` expands each value of the `row` tuple to be a positional argument value for the `Waypoint` function. This works because the column order in the CSV file matches the column order in the `namedtuple` definition.

This construction can also be performed using the `itertools` module, also. The `starmap()` function can be used as `starmap(Waypoint, raw_reader)`. This will also expand each tuple from the `raw_reader` to be a positional argument to the `Waypoint` function. Note that we can't use the built-in `map()` function for this. The `map()` function assumes that the function takes a single argument value. We don't want each four-item `row` tuple to be used as the only argument to the `Waypoint` function. We need to split the four items into four positional argument values.

How it works...

There are several parts to this recipe. Firstly, we've used the `csv` module for the essential parsing of rows and columns of data. We've leveraged the *Reading delimited files with the cvs module* recipe to process the physical format of the data.

Secondly, we've defined a `namedtuple()` that provides a minimal schema for our data. This is not very rich or detailed. It provides a sequence of column names. It also simplifies the syntax for accessing a particular column.

Finally, we've wrapped the `csv` reader in a generator function to build `namedtuple` objects for each row. This is a tiny change to the default processing, but it leads to a nicer style for the subsequent programming.

Instead of `row[2]` or `row['date']`, we can now use `row.date` to refer to a specific column. This is a small change that can simplify the presentation of complex algorithms.

There's more...

The initial example of processing the input has two additional problems. Firstly, the header row is mixed in with the useful rows of data; this header row needs to be rejected by a filter of some kind. Secondly, the data is all strings, and some conversion is necessary. We'll solve each of these by extending the recipe.

There are two common techniques for discarding the unneeded header row:

- We can use an explicit iterator and discard the first item. The general idea is as follows:

```
with waypoints_path.open() as waypoints_file:
    raw_reader = csv.reader(waypoints_file)
    waypoints_iter = iter(waypoints_reader)
    next(waypoints_iter)  # The header
    for row in waypoints_iter:
        print(row)
```

This snippet shows how to create an iterator object, `waypoints_iter`, from the raw CSV reader. We can use the `next()` function to skip a single item from this reader. The remaining items can be used to build useful rows of data. We can also use the `itertools.islice()` function for this.

- We can write a generator or use the `filter()` function to exclude selected rows:

```
with waypoints_path.open() as waypoints_file:
    raw_reader = csv.reader(waypoints_file)
    skip_header = filter(lambda row: row[0] != 'lat', raw_reader)
    waypoints_reader = (Waypoint(*row) for row in skip_header)
    for row in waypoints_reader:
        print(row)
```

This example shows how to create filtered generator, `skip_header`, from the raw CSV reader. The filter uses a simple expression, `row[0] != 'lat'`, to determine if a row is a header or has useful data. Only the useful rows are passed by this filter. The header row is rejected.

The other thing we'll need to do is to convert the various data items to more useful values. We'll follow the example of the *Simplifying complex algorithms with immutable data structures* recipe in online `Chapter 12`, *Functional And Reactive Programming Features* (link provided in `Preface`), and build a new `namedtuple` from the raw input data:

```
    Waypoint_Data = namedtuple('Waypoint_Data', ['lat', 'lon',
'timestamp'])
```

At this point in most projects, it becomes clear that the original name of the `Waypoint namedtuple` was poorly chosen. The code will need to be refactored to change the names to clarify the role of the original `Waypoint` tuple. This renaming and refactoring will occur several times as the design evolves. It's important to rename things as needed. We won't do the renaming here: we'll leave it for the reader to redesign the names.

To do the conversions, we need a function to handle the individual fields of a single `Waypoint`. This will create more useful values. It will involve using `float()` on the latitude and longitude values. It also requires some careful parsing of the date values.

Here's the first part of working with the separate date and time. These are two lambda objects-small functions with only a single expression that convert date or time strings to date or time values:

```
    import datetime
    parse_date = lambda txt: datetime.datetime.strptime(txt, '%Y-%m-
%d').date()
    parse_time = lambda txt: datetime.datetime.strptime(txt,
```

```
'%H:%M:%S').time()
```

We can use these to build a new `Waypoint_data` object from the original `Waypoint` object:

```
    def convert_waypoint(waypoint):
        return Waypoint_Data(
            lat = float(waypoint.lat),
            lon = float(waypoint.lon),
            timestamp = datetime.datetime.combine(
                parse_date(waypoint.date),
                parse_time(waypoint.time)
            )
        )
```

We've applied a series of functions that build a new data structure from an existing data structure. The latitude and longitude values were converted with the `float()` function. The date and time values were converted to a `datetime` object using the `parse_date` and `parse_time` lambdas with the `combine()` method of the `datetime` class.

This function allows us to build a more complete stack of processing steps for the source data:

```
    with waypoints_path.open() as waypoints_file:
        raw_reader = csv.reader(waypoints_file)
        skip_header = filter(lambda row: row[0] != 'lat', raw_reader)
        waypoints_reader = (Waypoint(*row) for row in skip_header)
        waypoints_data_reader = (convert_waypoint(wp) for wp in
waypoints_reader)
        for row in waypoints_data_reader:
            print(row.lat, row.lon, row.timestamp)
```

The original reader has been supplemented with a filter function to skip the header, a generator to create `Waypoint` objects, and another generator to create `Waypoint_Data` objects. Within the body of the `for` statement, we have a simple and easy-to-use data structure with pleasant names. We can refer to `row.lat` instead of `row[0]` or `row['lat']`.

Note that each generator function is lazy, it doesn't fetch any more input than is minimally required to produce some output. This stack of generator functions uses very little memory and can process files of unlimited size.

See also

- The *Upgrading CSV from dict reader to namespace reader* recipe does this with mutable `SimpleNamespace` data structure

Upgrading CSV from a DictReader to a namespace reader

When we read data from a CSV format file, we have two general choices for the resulting data structure:

- When we use `csv.reader()`, each row becomes a simple list of column values.
- When we use `csv.DictReader`, each row becomes a dictionary. By default, the contents of the first row become the keys for the row dictionary. We can also provide a list of values that will be used as the keys.

In both cases, referring to data within the row is awkward because it involves rather complex-looking syntax. When we use a reader, we must use `row[0]`, the semantics of this are completely obscure. When we use a `DictReader`, we can use `row['date']`, which is less obscure, but is a lot of typing.

In some real-world spreadsheets, the column names are impossibly long strings. It's hard to work with `row['Total of all locations excluding franchisees']`.

What can we do to replace complex syntax with something simpler?

Getting ready

The column names (and the data types for each column) are a schema for our data. The column titles are a schema that's embedded in the first row of the CSV data. This schema provides only attribute names; the data types aren't known and have to be treated as strings.

This points up two reasons for imposing an external schema on the rows of a spreadsheet:

- We can supply meaningful names.
- We can perform data conversions where necessary.

We can also use a schema to define data quality and cleansing processing. This can become quite sophisticated (and complicated). We'll limit our use of schema to providing column names and data conversions.

We'll look at a relatively simple CSV file that has some real-time data recorded from the log of a sailboat. This is the `waypoints.csv` file. The data looks like the following:

```
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 spreadsheet has four columns. Two of them are the latitude and longitude of the waypoint. It has a column with the date and the time as separate values. This isn't ideal, and we'll look at various data cleansing steps separately.

In this case, the column titles are valid Python variable names. This leads to an important simplification in the processing. In the cases where there are no column names, or the column names aren't Python variables, we'll have to apply a mapping from column name to preferred attribute name.

How to do it...

1. Import the modules and definitions required. In this case, it will be from `types`, `csv`, and `pathlib`:

```
from types import SimpleNamespace
from pathlib import Path
```

2. Import `csv` and define a `Path` object that refers to the data:

```
waypoints_path = Path('waypoints.csv')
```

3. Create the processing context for the open file:

```
with waypoints_path.open() as waypoints_file:
```

4. Define a CSV reader for the data. We'll call this a raw reader. In the long run, we'll follow the *Using stacked generator expressions* recipe in online `Chapter 12`, *Functional And Reactive Programming Features* (link provided in `Preface`), and use multiple generator expressions to cleanse and filter the data:

```
raw_reader = csv.DictReader(waypoints_file)
```

5. Define a generator that will convert these dictionaries into `SimpleNamespace` objects:

```
ns_reader = (SimpleNamespace(**row) for row in raw_reader)
```

This uses the generic `SimpleNamespace` class. When we need to use a more specific class, we can replace the `SimpleNamespace` with an application-specific class name. That class `__init__` must use keyword parameters that match the spreadsheet column names.

We can now process rows from this generator expression:

```
for row in ns_reader:
    print(row.lat, row.lon, row.date, row.time)
```

How it works...

There are several parts to this recipe. Firstly, we've used the `csv` module for the essential parsing of rows and columns of data. We've leveraged the *Reading delimited files with the cvs module* recipe to process the physical format of the data. The idea of the CSV format is to have columns of text that are comma separated in each row. There are rules for using quotes to allow the data within a column to contain a comma. The rules are all implemented within the `csv` module, saving us from writing a parser for this.

Secondly, we've wrapped the `csv` reader in a generator function to build a `SimpleNamespace` object for each row. This is a tiny extension to the default processing, but it leads to a nicer style for the subsequent programming. Instead of `row[2]` or `row['date']`, we can now use `row.date` to refer to a specific column. This is a small change that can simplify the presentation of complex algorithms.

There's more...

We may have two additional problems to solve. Whether or not these are needed depends on the data and the use for the data:

- How do we handle spreadsheet names that aren't proper Python variables?
- How can we convert data from text to a Python object?

It turns out that both of these needs can be handled elegantly with a function that does row by row conversion of data, and also handles any necessary renaming of columns:

```
def make_row(source):
    return SimpleNamespace(
        lat = float(source['lat']),
        lon = float(source['lon']),
        timestamp = make_timestamp(source['date'], source['time']),
```

```
        )
```

This function is in effect the schema definition for the original spreadsheet. Each line in this function provides several important pieces of information:

- The attribute name in the `SimpleNamespace`
- The conversion from the source data
- The source column names that were mapped to the final result

The goal is to define any helper or support functions required to be sure that each line of the conversion function is similar to the ones shown. Each line of this function is complete specification for a result column. As a bonus benefit, each line is written in Python notation.

This function can replace `SimpleNamespace` in the `ns_reader` statement. All of the conversion work is now focused into a single place:

```
ns_reader = (make_row(row) for row in raw_reader)
```

This row transformation function relies on a `make_timestamp()` function. This function converts two source columns to one resulting `datetime` object. The function looks like the following:

```
import datetime
make_date = lambda txt: datetime.datetime.strptime(
    txt, '%Y-%m-%d').date()
make_time = lambda txt: datetime.datetime.strptime(
    txt, '%H:%M:%S').time()

def make_timestamp(date, time):
    return datetime.datetime.combine(
            make_date(date),
            make_time(time)
        )
```

The `make_timestamp()` function breaks the timestamp creation into three parts. The first two parts are so simple that a lambda object was all that was needed. These are conversions from text to make `datetime.date` or `datetime.time` objects. Each conversion use the `strptime()` method to parse the date or time strings and return the appropriate class of object.

The third part could also have been a lambda, since it's also a single expression. However, it's a long expression, and it seemed slightly more clear to wrap it as a `def` statement. This expression uses the `combine()` method of `datetime` to combine a date and time into a single object.

See also

- The *Upgrading CSV from dict reader to namedtuple reader* recipe does this with an immutable `namedtuple` data structure instead of a `SimpleNamespace`

Using multiple contexts for reading and writing files

It's common to need to convert data from one format to another. For example, we might have a complex web log that we'd like to convert to a simpler format.

See the *Reading complex formats using regular expressions* recipe for a complex web log format. We'd like to do this parsing just one time.

After that, we'd like to work with a simpler file format, more like the format shown in the *Upgrading CSV from dict reader to namedtuple reader* or *Upgrading CSV from dict reader to namespace reader* recipe. A file that's in CSV notation can be read and parsed with the `csv` module, simplifying the physical format considerations.

How can we convert from one format to another?

Getting ready

Converting a file of data from one format to another means that the program will need to have two open contexts: one for reading and one for writing. Python makes this easy. The use of `with` statement contexts assures that the files are properly closed and all of the related OS resources are completely released.

We'll look at a common problem of summarizing many web log files. The source is in a format that we've seen in the *Writing generator functions with the yield statement* recipe in online `Chapter 12`, *Functional And Reactive Programming Features* (link provided in `Preface`), and also *Reading complex formats using regular expressions* recipe in this chapter. The rows look like the following:

```
    [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
```

These are difficult to process. The regular expression required to parse them is complex. For large volumes of data, it's also rather slow.

Here's the regular expression pattern for the various elements of the line:

```
import re
pattern_text = (r'\[(?P<date>\d+-\d+-\d+ \d+:\d+:\d+,\d+)\]'
    '\s+(?P<level>\w+)'
    '\s+in\s+(?P<module>[\w_\.]+):'
    '\s+(?P<message>.*)')
pattern = re.compile(pattern_text)
```

There are four parts to this complex regular expression:

- The date-time stamp is surrounded with `[ ]` and has a variety of digits, hyphens, colons, and a comma. It will be captured and assigned the name `date` by the `?P<date>` prefix on the `()` group.
- The severity level, which is a run of characters. This is captured and given the name level by the `?P<level>` prefix on the next `()` group.
- The module is a sequence of characters including _ and `.`. It's sandwiched between `in` and a `:`. The is assigned the name `module`.
- Finally, there's a message that extends to the end of the line. This is assigned to the message by the `?P<message>` inside the final `()`.

The pattern also includes runs of whitespace, `\s+`, which are not captured in any `()` groups. They're quietly ignored.

When we create a `match` object using this regular expression, the `groupdict()` method of that `match` object will produce a dictionary with the names and values from each line. This matches the way the `csv` reader works. It provides a common framework for processing complex data.

We'll use this in a function that iterates through rows of log data. The function will apply the regular expression, and yield the group dictionaries:

```
def extract_row_iter(source_log_file):
    for line in source_log_file:
        match = log_pattern.match(line)
        if match is None:
            # Might want to write a warning
            continue
        yield match.groupdict()
```

This function looks at each line in the given input file. It applies the regular expression to the line. If the line matches, this will capture the relevant fields of data. If there is no match, the line didn't follow the expected format; this may deserve an error message. There's no useful data to yield, so the `continue` statement skips the rest of the body of the `for` statement.

The `yield` statement produces the dictionaries of matches. Each dictionary will have the four named fields and the captured data from the log. The data will be text only, so additional conversions will have to be applied separately.

We can use the `DictWriter` class from the `csv` module to emit a CSV file with these various data elements neatly separated. Once we've created a CSV file, we can process the data simply and much more quickly than the raw log rows.

How to do it...

1. This recipe will need three components:

```
import re
from pathlib import Path
import csv
```

2. Here's the pattern that matches the simple Flask logs. For other kinds of logs, or other formats configured into Flask, a different pattern will be required:

```
log_pattern = re.compile(
    r"\[(?P<timestamp>.*?)\]"
    r"\s(?P<levelname>\w+)"
    r"\sin\s(?P<module>[\w\._]+):"
    r"\s(?P<message>.*)")
```

3. Here's the function that yields dictionaries for the matching rows. This applies the regular expression pattern. Non-matches are silently skipped. The matches will yield a dictionary of item names and their values:

```
def extract_row_iter(source_log_file):
    for line in source_log_file:
        match = log_pattern.match(line)
        if match is None: continue
        yield match.groupdict()
```

4. We'll define the `Path` object for the resulting log summary file:

```
summary_path = Path('summary_log.csv')
```

5. We can then open the results context. Because we're using a `with` statement, we're assured that the file will be properly closed no matter what else happens in this script:

```
with summary_path.open('w') as summary_file:
```

6. Since we're writing a CSV file based on a dictionary, we'll define a `csv.DictWriter`. This is indented four spaces inside the `with` statement. We must provide the expected keys from the input dictionary. This will define the order for the columns in the resulting file:

```
writer = csv.DictWriter(summary_file,
    ['timestamp', 'levelname', 'module', 'message'])
writer.writeheader()
```

7. We'll define a `Path` object for the source directory with log files. In this case, the log files happen to be in the directory with the script. This is rare, and using an environment variable might be a lot more useful:

```
source_log_dir = Path('.')
```

   We can imagine using `os.environ.get('LOG_PATH', '/var/log')` as a more general solution than a hard-coded path.

8. We'll use the `glob()` method of a `Path` object to find all files that match the required name:

```
for source_log_path in source_log_dir.glob('*.log'):
```

   This, too, could benefit from having the pattern string fetched from an environment variable or command-line parameter.

9. We'll define a context for reading each source file. This context manager will guarantee that the input files are properly closed and the resources released. Note that this is indented inside the previous `with` and `for` statements, a total of eight spaces. This is particularly important when processing a large number of files:

```
with source_log_path.open() as source_log_file:
```

10. We'll use the writer's `writerows()` method to write all valid rows from the `extract_row_iter()` function. This is indented inside both `with` statements, as well as the `for` statement. This is the core of the process:

```
writer.writerows(extract_row_iter(source_log_file) )
```

11. We can also write a summary. This is indented inside the outer with and `for` statements. It summarizes the processing of the preceding with statement:

```
print('Converted', source_log_path, 'to', summary_path)
```

How it works...

Python works nicely with multiple context managers. We can easily have deeply-nested `with` statements. Each `with` statement can manage a different context object.

Since open files are context objects, it makes the most sense to wrap every open file in a `with` statement to be sure that the file is properly closed and all OS resources are released from the file.

We've used `Path` objects to represent the filesystem locations. This gives us the ability to easily create output names based on input names, or rename the files after they've been processed. For more information on this, see the *Using pathlib to work with filenames* recipe.

We've used a generator function to combine two operations. Firstly, there's a mapping from source text to individual fields. Secondly, there's a filter that excludes source text that doesn't match the expected pattern. In many cases, we can use the `map()` and `filter()` functions to make this a little more clear.

When using regular expression matching; however, it's not as easy to separate the mapping and filter parts of the operation. The regular expression may not match some input lines, which becomes a kind of filtering that's bundled in to the mapping. Because of this, a generator function works out very nicely.

The `csv` writers have a `writerows()` method. This method accepts an iterator as it's parameter value. This makes it easy to provide a generator function to the writer. The writer will consume objects as they're produced by the generator. Very large files can be handled this way because the entire file isn't read into memory, just enough of the file is read to create a complete line of data.

There's more...

It's often essential to have a summary count of the number of lines of log file read from each source, the number of lines discarded because they didn't match, and the number of lines finally written to the summary file.

This is challenging when using generators. The generator produces lots of rows of data. How can it also produce a summary?

The answer is that we can provide a mutable object as a parameter to the generator. The ideal kind of mutable object is an instance of `collections.Counter`. We can use this to count events including a valid record, an invalid record, or even occurrences of specific data values. The mutable object can be shared by the generator and the overall main program so that the main program can print the count information to a log.

Here's the map-filter function that converts text to useful dictionary objects. We've written a second version called `counting_extract_row_iter()` to emphasize the additional feature:

```
def counting_extract_row_iter(counts, source_log_file):
    for line in source_log_file:
        match = log_pattern.match(line)
        if match is None:
            counts['non-match'] += 1
            continue
        counts['valid'] += 1
        yield match.groupdict()
```

We've provided an additional argument, `counts`. When we find rows that don't match the regular expression, we can increment the `non-match` key in the `Counter`. When we find rows that do match properly, we can increment the `valid` key in the `Counter`. This provides a summary that shows how may rows were processed from the given file.

The overall processing script looks like the following:

```
summary_path = Path('summary_log.csv')
with summary_path.open('w') as summary_file:

    writer = csv.DictWriter(summary_file,
        ['timestamp', 'levelname', 'module', 'message'])
    writer.writeheader()

    source_log_dir = Path('.')
    for source_log_path in source_log_dir.glob('*.log'):
        counts = Counter()
        with source_log_path.open() as source_log_file:
```

```
            writer.writerows(
                counting_extract_row_iter(counts, source_log_file)
                )

    print('Converted', source_log_path, 'to', summary_path)
    print(counts)
```

We've made three small changes:

- Create an empty `Counter` object just before processing a source log file.
- Provide the `Counter` object to the `counting_extract_row_iter()` function. The function will update the counter as it processes rows.
- Print the value of the `counter` after processing the files. The unadorned output isn't very pretty, but it tells an important story.

We might see output like the following:

```
Converted 20160612.log to summary_log.csv
Counter({'valid': 86400})
Converted 20160613.log to summary_log.csv
Counter({'valid': 86399, 'non-match': 1)
```

This kind of output shows us how large the `summary_log.csv` will be, and it also shows that something was wrong in the `20160613.log` file.

We can easily extend this to combine all of the individual source file counters to produce a single large output at the end of the process. We can combine multiple `Counter` objects using the `+` operator to create a grand sum of all of the data. Details are left as an exercise for the reader.

See also

- For the basics of a context, see the *Reading and writing files with context managers* recipe

9
Testing

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

- Using docstrings for testing
- Testing functions that raise exceptions
- Handling common doctest issues
- Creating separate test modules and packages
- Combining unittest and doctest tests
- Testing things that involve dates or times
- Testing things that involve randomness
- Mocking external resources

Introduction

Testing is central to creating working software. Here's the canonical statement in the importance of testing:

> *Any program feature without an automated test simply doesn't exist.*

This is from Kent Beck's book, *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 aggregated suite of software components works properly. This is often used for overall acceptance of software as fit for use.
- **Performance testing**: This assures that a unit meets performance objectives. 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.

Python has two built-in testing frameworks. One examines the docstrings for examples that include the `>>>` prompt. This is the `doctest` tool. While this is widely used for unit testing, it can also be used for simple integration testing.

The other testing framework uses classes built with definitions from the `unittest` module. This module defines a `TestCase` class. This, too, is designed primarily for unit testing, but can also be applied to integration and performance testing.

Of course, we'll want to combine these tools. Both modules have features to allow coexistence. We'll often leverage the test loading protocol from the `unittest` package to merge all of our tests.

Additionally, we might use the tools `nose2` or `py.test` to further automate test discovery and add additional features such as test case coverage. These projects are often helpful for particularly complex applications.

It's sometimes helpful to summarize a test using the GIVEN-WHEN-THEN style of test case naming:

- **GIVEN** some initial state or context
- **WHEN** a behavior is requested
- **THEN** the component under test has some expected result or state change

Using docstrings for testing

Good Python includes docstrings inside every module, class, function, and method. Many tools can create useful, informative documentation from the docstrings.

One important element of a docstring is an example. The example becomes a kind of unit test case. An example often fits the GIVEN-WHEN-THEN model of testing because it shows a unit, a request, and a response.

How can we turn examples into proper test cases?

Getting ready

We'll look at a simple function definition as well as a simple class definition. Each of these will include docstrings that include examples which can be used as formal tests.

Here's a simple function that computes 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:

$$\binom{n}{k} = \frac{n!}{k! \times (n-k!)}$$

This defines a small Python function that we can write like this:

```
from math import factorial
def binom(n: int, k: int) -> int:
    return factorial(n) // (factorial(k) * factorial(n-k))
```

This function does a simple calculation and returns a value. Since it has no internal state, it's 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 simple class which has a lazy calculation of mean and median. It uses an internal `Counter` object which can be interrogated to determine the mode:

```
from statistics import median
from collections import Counter

class Summary:

    def __init__(self):
        self.counts = Counter()

    def __str__(self):
        return "mean = {:.2f}\nmedian = {:d}".format(
        self.mean, self.median)
```

```
    def add(self, value):
        self.counts[value] += 1

    @property
    def mean(self):
        s0 = sum(f for v,f in self.counts.items())
        s1 = sum(v*f for v,f in self.counts.items())
        return s1/s0

    @property
    def median(self):
        return median(self.counts.elements())
```

The `add()` method changes the state of this object. Because of this state change, we'll need to provide more sophisticated examples that show how an instance of the `Summary` class behaves.

How to do it...

We'll show two variations in this recipe. The first is for largely stateless operations, such as the computation of the `binom()` function. The second is for stateful operations, such as the `Summary` class.

1. Put examples into the docstrings.
2. Run the doctest module as a program. This is done in one of two ways:

   - At the command prompt:

     ```
     $ python3.5 -m doctest code/ch11_r01.py
     ```

     If all of the examples pass, there's no output. Using the `-v` option produces verbose output summarizing the tests.

   - By including a `__name__ == '__main__'` section. This can import the doctest module and execute the `testmod()` function:

     ```
     if __name__ == '__main__':
         import doctest
         doctest.testmod()
     ```

     If all of the examples pass, there's no output. To see some output, use the `verbose=1` parameter for the `testmod()` function to create more verbose output.

Writing examples for stateless functions

1. Start the docstring with a summary:

```
'''Computes the binomial coefficient.
This shows how many combinations of
*n* things taken in groups of size *k*.
```

2. Include the parameter definitions:

```
:param n: size of the universe
:param k: size of each subset
```

3. Include the return value definition:

```
:returns: the number of combinations
```

4. Mock up an example of using the function at Python's `>>>` prompt:

```
>>> binom(52, 5)
2598960
```

5. Close the long docstring with the appropriate quotation marks:

```
'''
```

Writing examples for stateful objects

1. Write a class-level docstring with a summary:

```
'''Computes summary statistics.

'''
```

We've left space to fill in examples.

2. Write the method-level docstring with a summary. Here's the `add()` method:

```
def add(self, value):
    '''Adds a value to be summarized.

    :param value: Adds a new value to the collection.
    '''
    self.counts[value] += 1
```

3. Here's the `mean()` method:

```
@property
def mean(self):
    '''Computes the mean of the collection.
    :return: mean value as a float
    '''
    s0 = sum(f for v,f in self.counts.items())
    s1 = sum(v*f for v,f in self.counts.items())
    return s1/s0
```

   A similar string is required for the `median()` method, and any others that are written.

4. Extend the class-level docstring concrete examples. In this case, we'll write two. The first example shows that the `add()` method has no return value, but changes the state of the object. The `mean()` method reveals this state:

```
>>> s = Summary()
>>> s.add(8)
>>> s.add(9)
>>> s.add(9)
>>> round(s.mean, 2)
8.67
>>> s.median
9
```

   We've rounded the result of the mean to avoid displaying a long floating-point value that might not have the exact same text representation on all platforms. When we run doctest, we'll generally get a silent response because the test passed.

The second example shows a multiline result from the `__str__()` method:

```
>>> print(str(s))
mean = 8.67
median = 9
```

What happens when something doesn't work? Imagine that we changed the expected output to have a wrong answer. When we run doctest, we'll see output like this:

```
**********************************************************************
File "__main__", line ?, in __main__.Summary
Failed example:    s.medianExpected:
    10
Got:
    9
**********************************************************************1
```

```
items had failures:
   1 of   6 in __main__.Summary
***Test Failed*** 1 failures.TestResults(failed=1, attempted=9)
```

This shows where the error is. It shows an expected value from the test example, and the actual answer.

How it works...

The `doctest` module includes a main program—as well as several functions—that will scan a Python file for `>>>` examples. We can leverage the module scanning function, `testmod()`, to scan the current module. We can use this to scan any imported module.

The scanning operation looks for blocks of text that have a characteristic pattern of a `>>>` line followed by lines that show the response from the command.

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.
- Single line of response text: This was exemplified by the `binom()` function and the `mean()` method.
- 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.

The doctest module will execute each line of code shown with a `>>>` prompt. It compares the actual results with the expected results. The comparison is a very simple text matching. Unless special annotations are used, the output must precisely match the expectations.

The simplicity of this testing protocol imposes some software design requirements. 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, the design must be kept simple enough that it can be demonstrated interactively. Keeping software simple enough to demonstrate at the `>>>` prompt is often beneficial.

The simplicity of the comparison of the results can impose some complications on the output that's being displayed. Note, for example, that 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.

Python 3.5.1 (on Mac OS X) shows `8.666666666666666` where Python 2.6.9 (also on Mac OS X) shows `8.666666666666661`. The values are equal for 16 of the decimal digits. This is about 48 bits of data, which is the practical limit of floating-point values.

We'll address the exact comparison issue in detail in the *Handling common doctest issues* recipe.

There's more...

One of the important test considerations is edge cases. An **edge case** generally focuses on the limits for which a calculation is designed. There are, for example, two edges to the binomial function:

$$\binom{n}{0} = \binom{n}{n} = 1$$

We can easily add these to the examples to be sure that our implementation is sound; this leads to a function that looks like the following:

```
def binom(n: int, k: int) -> int:
    '''Computes the binomial coefficient.
    This shows how many combinations 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
    '''
    return factorial(n) // (factorial(k) * factorial(n-k))
```

In some cases, we might need to test values that are outside the valid range of values. These cases aren't really ideal for putting into the docstring, because they clutter an explanation of what is supposed to happen with an explanation of other things that should never normally happen.

We can include additional docstring test cases in a global variable named `__test__`. This variable must be a mapping. The keys to the mapping are test case names, and the values of the mapping are doctest examples. These will need to be triple-quoted strings.

Because the examples 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.

We might add something like this:

```
__test__ = {
'GIVEN_binom_WHEN_0_0_THEN_1':
'''
>>> binom(0, 0)
1
''',

}
```

We've written the mapping with the keys with no indent. The values have been indented four spaces so that they stand out from the keys and are slightly easier to spot.

These test cases are found by the doctest program and included in the overall suite of tests. We can use this for tests that are important, but aren't really helpful as documentation.

See also

- In the *Testing functions that raise exceptions* and *Handling common doctest issues* recipes, we'll look at two additional doctest techniques. These are important because exceptions can often include a traceback which may include object IDs that can vary each time the program is run.

Testing functions that raise exceptions

Good Python includes docstrings inside every module, class, function, and method. Many tools can create useful, informative documentation from these docstrings.

One important element of a docstring is an example. The example becomes a kind of unit test case. Doctest does simple, literal matching of the expected output against the actual output.

When an example raises an exception, though, the traceback messages from Python are not always identical. It may include object ID values that change or module line numbers which may vary slightly depending on the context in which the test is executed. The literal matching rules for doctest aren't appropriate when exceptions are involved.

How can we turn exception processing and the resulting traceback messages into proper test cases?

Getting ready

We'll look at a simple function definition as well as a simple class definition. Each of these will include docstrings that include examples which can be used as formal tests.

Here's a simple function that computes the binomial coefficient of two numbers. It shows the number of combinations of *n* things taken in groups of *k*. For example, how many ways a 52-card deck can be dealt into 5-card hands:

$$\binom{n}{k} = \frac{n!}{k! \times (n-k!)}$$

This defines a small Python function that we can write like this:

```
from math import factorial
def binom(n: int, k: int) -> int:
    '''
    Computes the binomial coefficient.
    This shows how many combinations 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
    '''
    return factorial(n) // (factorial(k) * factorial(n-k))
```

This function does a simple calculation and returns a value. 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...

1. Create a global `__test__` variable in the module:

```
__test__ = {

}
```

   We've left space to insert one or more test cases.

2. For each test case, provide a name and a placeholder for the example:

```
__test__ = {
'GIVEN_binom_WHEN_wrong_relationship_THEN_error':
'''
    example goes here.
''',
}
```

3. Include the invocation with a `doctest` directive comment, `IGNORE_EXCEPTION_DETAIL`. This replaces the "example goes here":

```
>>> binom(5, 52)  # doctest: +IGNORE_EXCEPTION_DETAIL
```

   The directive starts with `# doctest:`. Directives are enabled with `+` and disabled with `-`.

4. Include an actual traceback message. This is part of the *example goes here*; it goes after the `>>>` statement to show the expected response:

```
      Traceback (most recent call last):
        File
"/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/doctest.py
", line 1320, in __run
          compileflags, 1), test.globs)
        File "<doctest
```

```
__main__.__test__.GIVEN_binom_WHEN_wrong_relationship_THEN_error[0]>", line
1, in <module>
            binom(5, 52)
          File "/Users/slott/Documents/Writing/Python
Cookbook/code/ch11_r01.py", line 24, in binom
            return factorial(n) // (factorial(k) * factorial(n-k))
        ValueError: factorial() not defined for negative values
```

5. The three lines that start with `File...` will be ignored. The `ValueError:` line will be checked to be sure that the test produces the expected exception.

The overall statement looks like this:

```
    __test__ = {
    'GIVEN_binom_WHEN_wrong_relationship_THEN_error': '''
        >>> binom(5, 52)  # doctest: +IGNORE_EXCEPTION_DETAIL
        Traceback (most recent call last):
          File
"/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/doctest.py
", line 1320, in __run
            compileflags, 1), test.globs)
          File "<doctest
__main__.__test__.GIVEN_binom_WHEN_wrong_relationship_THEN_error[0]>", line
1, in <module>
            binom(5, 52)
          File "/Users/slott/Documents/Writing/Python
Cookbook/code/ch11_r01.py", line 24, in binom
            return factorial(n) // (factorial(k) * factorial(n-k))
        ValueError: factorial() not defined for negative values
    '''
    }
```

We can now use a command like this to test the entire module's features:

```
python3.5 -R -m doctest ch11_r01.py
```

How it works...

The doctest parser has several directives that can be used to modify the testing behavior. The directives are included as special comments with the line of code that performs the test action.

We have two ways to handle tests that include an exception:

- We can use `# doctest: +IGNORE_EXCEPTION_DETAIL` and 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 very easy to copy an actual error and paste it into the documentation.
- We can use `# doctest: +ELLIPSIS` and replace parts of the traceback message with `...`. This, too, allows an expected output to elide details and focus on the last line that has the actual error.

For this second kind of exception example, we might include a test case like this:

```
'GIVEN_binom_WHEN_negative_THEN_exception':
'''
    >>> binom(52, -5)  # doctest: +ELLIPSIS
    Traceback (most recent call last):
    ...
    ValueError: factorial() not defined for negative values
''',
```

The test case uses the `+ELLIPSIS` directive. The details of the error traceback have had irrelevant material replaced with `...`. The relevant material has been left intact so that the actual exception message will match the expected exception message precisely.

Doctest will ignore everything between the first `Traceback...` line and the final `ValueError: ...` line. Generally, the final line is all that matters for proper execution of the test. The intermediate text depends on the context in which the test was run.

There's more...

There are several more comparison directives that can be provided to individual tests.

- `+ELLIPSIS`: This allows an expected result to be generalized by replacing details with `...`.
- `+IGNORE_EXCEPTION_DETAIL`: This allows an expected value to include a complete traceback message. The bulk of the traceback will be ignored, and only the final exception line is checked.
- `+NORMALIZE_WHITESPACE`: In some cases, the expected value might be wrapped onto multiple lines to make it easier to read. Or, it might have spacing that varies slightly from standard Python values. Using this flag allows some flexibility in the whitespace for the expected value.

- `+SKIP`: The test is skipped. This is sometimes done for tests that are designed for a future release. Tests may be included prior to the feature being completed. The test can be left in place for future development work, but skipped in order to release a version on time.
- `+DONT_ACCEPT_TRUE_FOR_1`: This covers a special case that was common in Python 2. Before `True` and `False` were added to the language, values `1` and `0` were used instead. The doctest algorithm for comparing expected results to actual results will honor this older scheme by matching `True` and `1`. This directive can be provided at the command line using `-o DONT_ACCEPT_TRUE_FOR_1`. This change would then be globally true for all tests.
- `+DONT_ACCEPT_BLANKLINE`: Normally, a blank line ends an example. In the case where the example output includes a blank line, the expected results must use the special syntax `<blankline>`. Using this shows where a blank line is expected, and the example doesn't end at this blank line. In very rare cases, the expected output will actually include the string `<blankline>`. This directive assures that `<blankline>` is not used to mean a blank line but stands for itself. This would make sense when writing tests for the doctest module itself.

These can also be provided as the `optionsflags` parameter when evaluating the `testmod()` or `testfile()` functions.

See also

- See the *Using docstrings for testing* recipe for the basics of doctest
- See the *Handling common doctest issues* recipe for other special cases that require doctest directives

Handling common doctest issues

Good Python includes docstrings inside every module, class, function, and method. Many tools can create useful, informative documentation from minimally complete docstrings.

One important element of a docstring is an example. The example becomes a kind of unit test case. Doctest does simple, literal matching of the expected output against the actual output. There are some Python objects, however, that are not consistent every time they're referred to.

For example, all object hash values are randomized. This means that the order of elements in a set or the order of keys in a dictionary can vary. We have several choices for creating test case example output:

- Write tests that can tolerate randomization. Often by converting to a sorted structure.
- Stipulate a value for the `PYTHONHASHSEED` environment variable.
- Require that Python be run with the `-R` option to disable hash randomization entirely.

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 and time 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.

We'll focus on the first two issues with some doctest techniques in this recipe. We'll look at `datetime` and `random` in the *Testing things that involve dates or times* and *Testing things that involve randomness* recipes. We'll look at how to work with external resources in the *Mocking external resources* recipe.

Doctest examples require an exact match with the text. How can we write doctest examples that handle hash randomization or floating-point implementation details appropriately?

Getting ready

In the *Reading delimited files with the CSV module* recipe, we looked at how the `csv` module will read data, creating a mapping for each row of input. In that recipe, we saw a `CSV` file that has some real-time data recorded from the log of a sailboat. This is the `waypoints.csv` file.

The `DictReader` class produces rows that look like this:

```
{'date': '2012-11-27',
 'lat': '32.8321666666667',
 'lon': '-79.9338333333333',
 'time': '09:15:00'}
```

This is a doctest nightmare because the hash randomization assures that the order of the keys in this dictionary is likely to be different.

When we try to write doctest examples that involve a dictionary, we'll often see problems like this:

```
Failed example:
    next(row_iter)
Expected:
    {'date': '2012-11-27', 'lat': '32.8321666666667',
    'lon': '-79.9338333333333', 'time': '09:15:00'}
Got:
    {'lon': '-79.9338333333333', 'time': '09:15:00',
    'date': '2012-11-27', 'lat': '32.8321666666667'}
```

The data in the expected and actual rows clearly matches. However, the string displays of the dictionary values aren't exactly identical. The keys are not shown in a consistent order.

We'll also look at a small real-valued function so that we can work with floating-point values:

$$\phi(n) = \frac{1}{2}\left[1 + erf\left(\frac{n}{\sqrt{2}}\right)\right]$$

This function is the cumulative probability density function for standard z-scores. After normalizing a variable, the mean of the Z-score values for that variable will be zero, and the standard deviation will be one. See the *Creating a partial function* recipe in online `Chapter 12`, *Functional And Reactive Programming Features* (link provided in `Preface`), for more information on the idea of normalized scores.

This function, $\Phi(n)$, tells us exactly what fraction of the population is below a given z-score. For example, $\Phi(0) = 0.5$: half the population has a z-score below zero.

This function involves some rather complex processing. The unit tests have to reflect the floating-point precision issues.

How to do it...

We'll look at mapping (and set) ordering in one recipe. We'll look at the floating point separately.

Writing doctest examples for mapping or set values

1. Import the necessary libraries and define the function:

```
import csv
def raw_reader(data_file):
    """
    Read from a given, open file.

    :param data_file: Open file, ready to be processed.
    :returns: iterator over individual rows as dictionaries.

    Example:

    """
    data_reader = csv.DictReader(data_file)
    for row in data_reader:
        yield row
```

   We've included the example heading in the document string.

2. We can replace actual data files with instances of the `StringIO` class from the `io` package. This can be used inside the example to provide fixed sample data:

```
>>> from io import StringIO
>>> mock_file = StringIO('''lat,lon,date,time
... 32.8321,-79.9338,2012-11-27,09:15:00
... ''')
>>> row_iter = iter(raw_reader(mock_file))
```

3. Conceptually, the test case is this. This code will not work properly because the keys will be scrambled. However, it can be refactored easily:

```
>>> row = next(row_iter)
>>> row
{'time': '09:15:00', 'lat': '32.8321', etc. }
```

We've omitted the rest of the output, since it varies each time the test is run:

The code must be written like this to force the keys into a fixed order:

```
>>> sorted(row.items())  # doctest: +NORMALIZE_WHITESPACE
[('date', '2012-11-27'), ('lat', '32.8321'),
('lon', '-79.9338'), ('time', '09:15:00')]
```

The sorted items are in a consistent order.

Writing doctest examples for floating-point values

1. Import the necessary libraries and define the function:

```
from math import *
def phi(n):
    """
    The cumulative distribution function for the standard normal
    distribution.

    :param n: number of standard deviations
    :returns: cumulative fraction of values below n.

    Examples:
    """
    return (1+erf(n/sqrt(2)))/2
```

   We've left a space for examples in the document string.

2. For each example, include an explicit use of `round()`:

```
>>> round(phi(0), 3)
0.399
>>> round(phi(-1), 3)
0.242
>>> round(phi(+1), 3)
0.242
```

The float values are rounded so that differences in floating-point implementation details don't lead to seemingly incorrect results.

How it works...

Because of hash randomization, the hash keys used for dictionaries are unpredictable. This is an important security feature, and defeats a subtle denial-of-service attack. For details, see `http://www.ocert.org/advisories/ocert-2011-003.html`.

We have two ways to work with dictionary keys that have no defined order:

- We can write test cases that are specific to each key:

```
>>> row['date']
'2012-11-27'
>>> row['lat']
'32.8321'
>>> row['lon']
'-79.9338'
>>> row['time']
'09:15:00'
```

- We can convert to a data structure with a fixed order. The value of `row.items()` is an iterable sequence of pairs with each key and value. The order is not set in advance, but we can use the following to force an order:

```
>>> sorted(row.items())
```

This returns a list with the keys sorted into order. This allows us to create a consistent literal value that will always be the same every time the test is evaluated.

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 trust that all 53 bits have exactly the right value, it's often much easier to round the value to a value that's a good fit with the problem domain.

For most modern processors, floating-point values are often either 32 or 64-bit values. A 32-bit value has about seven decimal digits. Rounding the value so that there are no more than six digits in the value is generally the simplest approach.

Rounding to six digits does not mean using `round(x, 6)`. The `round()` function doesn't preserve a number of digits. This function rounds to a number of digits to the right of the decimal point; it doesn't account for digits to the left of the decimal point. Rounding a number on the order of 10^{12} to six positions to the right of the decimal point leads to 18 digits—too many for a 32-bit value. Rounding a number on the order of 10^{-7} to six positions to the right of the decimal place leads to zero.

There's more...

When working with `set` objects, we must also be careful of the order of the items. We can generally use `sorted()` to convert a `set` to a `list` and impose a specific order.

Python `dict` objects show up in a surprising number of places:

- When we write a function that uses the `**` to collect a dictionary of argument values. There's no guarantee for the order of the arguments.
- When we use a function such as `vars()` to create a dictionary from local variables, or from the attributes of an object, the dictionary has no guaranteed order.
- When we write programs that rely on introspection of a class definition, the methods are defined in a class-level dictionary object. We can't predict their order.

This becomes apparent when there are unreliable test cases. A test case which passes or fails seemingly randomly may have a result which is based on a hash randomization. Extract the keys and sort them to overcome this problem.

We can run the tests using this command-line option too:

```
python3.5 -R -m doctest ch11_r03.py
```

This will turn off hash randomization while running doctest on a specific file, `ch11_r03.py`.

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 will show how to test processing that involves `random`.

Creating separate test modules and packages

We can do any kind of unit testing in docstring examples. There are some things, however, which can become extremely tedious when done this way.

The `unittest` module allows us to step beyond simple examples. These tests rely on test case class definitions. A subclass of `TestCase` can be used to write very complex and sophisticated tests; these can be simpler than the same test done as doctest examples.

The `unittest` module also allows us to package tests outside the docstrings. This can be helpful for particularly complex tests of corner cases that aren't as helpful when placed in the documentation. Ideally, doctest cases illustrate the **happy path –** the most common use cases. It's common to use `unittest` for test cases which are off the happy path.

How can we create more sophisticated tests?

Getting ready

A test can often be summarized by a three-part *Given-When-Then* story:

- **GIVEN**: Some unit in an initial state or context
- **WHEN**: A behavior is requested
- **THEN**: The component under test has some expected result or state change

The `TestCase` class doesn't precisely follow this three-part structure. It has two parts; some design choices must be made regarding where the three parts of a test are allocated:

- A `setUp()` method that implements the *Given* aspect of the test case. It can also handle the *When* aspect.
- A `runTest()` method that must handle the *Then* aspects. This can also handle the *When* aspect. The *Then* conditions are confirmed by a series of assertions. These generally use the sophisticated assertion methods of the `TestCase` class.

The choice of where to implement the *When* aspect is tied to the question of reuse. In most cases, there are many alternative *When* conditions, each with a unique *Then* to confirm correct operation. The *Given* might be common to the `setUp()` method, and shared by a number of `TestCase` subclasses. Each subclass would have a unique `runTest()` method to implement the *When* and *Then* aspects.

In some cases, the *When* aspect is split into some common parts and some test-case-specific parts. In this case, the *When* aspect may be partly defined in the `setUp()` method and partly defined in `runTest()` method.

We'll create some tests for a class that is designed to compute some basic descriptive statistics. We'd like to provide sample data that's far larger than anything we'd ever enter as doctest examples. We'd like to use thousands of data points rather than two or three.

Here's an outline of the class definition that we'd like to test. We'll only provide the methods and some summaries. The bulk of the code was shown in the *Using docstrings for testing* recipe. We've omitted all of the implementation details. This is just an outline of the class, provided as a reminder of what the method names are:

```
from statistics import median
from collections import Counter

class Summary:
    def __init__(self):
       pass

    def __str__(self):
        '''Returns a multi-line text summary.'''

    def add(self, value):
        '''Adds a value to be summarized.'''

    @property
    def count(self):
        '''Number of samples.'''

    @property
    def mean(self):
        '''Mean of the collection.'''

    @property
    def median(self):
        '''Median of the collection.'''
        return median(self.counts.elements())

    @property
    def mode(self):
        '''Returns the items in the collection in decreasing
        order by frequency.
        '''
```

Because we're not looking at the implementation details, this is a kind of black box testing. The code is a black box—the internals are opaque. To emphasize that, we omitted the implementation details from the preceding code.

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...

1. We'll include the test code in the same module as the working code. This will follow the pattern of doctest that bundles tests and code together. We'll use the `unittest` module for creating test classes:

```
import unittest
import random
```

   We'll also be using `random` to scramble the input data.

2. Create a subclass of `unittest.TestCase`. Provide this class with a name that shows the intent of the test:

```
class GIVEN_Summary_WHEN_1k_samples_THEN_mean(unittest.TestCase):
```

   The *GIVEN-WHEN-THEN* names are very long. We'll rely on `unittest` to discover all subclasses of `TestCase` so we don't have to type this class name more than once.

3. Define a `setUp()` method in this class which handles the *Given* aspect of the test. This creates a context for test processing:

```
def setUp(self):
    self.summary = Summary()
    self.data = list(range(1001))
    random.shuffle(self.data)
```

   We've created a collection of `1,001` samples ranging in value from `0` to `1,000`. The mean is 500 exactly, so is the median. We've shuffled the data into a random order.

4. Define a `runTest()` method which handles the *When* aspect of the test. This performs the state change:

```
def runTest(self):
    for sample in self.data:
        self.summary.add(sample)
```

5. Add assertions to implement the *Then* aspect of the test. This confirms that the state changes worked properly:

```
self.assertEqual(500, self.summary.mean)
self.assertEqual(500, self.summary.median)
```

6. To make it very easy to run, add a main program section:

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

With this, the test can be run at Command Prompt. It can also be run from the command line.

How it works...

We're using several parts of the `unittest` module:

- The `TestCase` class is used to define one test case. This can have a `setUp()` method to create the unit and possibly the request . This must have at least a `runTest()` to make the request and check the response.

  We can have as many of these class definitions in a file as we need to build up an appropriate set of tests. For simple classes, there may be only a few test cases. For complex modules, there may be dozens or even hundreds of cases.

- The `unittest.main()` function does several things:
  - It creates an empty `TestSuite` that will contain all the `TestCase` objects.
  - It uses a default loader to examine a module and find all of the `TestCase` instances. These are loaded into the `TestSuite`. This process is something that we might want to modify or extend.
  - It then runs the `TestSuite` and displays a summary of the results.

When we run this module, we'll see output that looks like this:

```
.----------------------------------------------------------------------
Ran 1 test in 0.005s

OK
```

As each test is passed, a `.` is displayed. This shows that the test suite is making progress. After the line of – is a summary of the tests run, and the time. If there are failures, or exceptions, the counts will reflect this.

Finally, there's a summary of `OK` to show whether all tests passed or any tests have failed.

If we change the test slightly to be sure that it fails, we'll see the following output:

```
F
======================================================================
FAIL: runTest (__main__.GIVEN_Summary_WHEN_1k_samples_THEN_mean)
----------------------------------------------------------------------
Traceback (most recent call last):
 File "/Users/slott/Documents/Writing/Python Cookbook/code/ch11_r04.py",
line 24, in runTest
 self.assertEqual(501, self.summary.mean)
AssertionError: 501 != 500.0
----------------------------------------------------------------------
Ran 1 test in 0.004s
FAILED (failures=1)
```

Instead of a `.` for a passing test, a failing test displays an `F`. This is followed by the traceback from the assertion which failed. To force the test to fail, we changed the expected mean to `501`, which is not the computed mean value of `500.0`.

There's a final summary of `FAILED`. This includes the reason why the suite as a whole is a failure: `(failures=1)`.

There's more...

In this example, we have two *Then* conditions inside the `runTest()` method. If one fails, the test stops as a failure, and the other condition is not exercised.

This is a weakness in the design of this test. If the first test fails, we won't get all of the diagnostic information we might want. We should avoid independent collections of assertions in the `runTest()` method. In many cases, a test case may involve multiple dependent assertions; a single failure provides all the diagnostic information that's required. The clustering of assertions is a design trade-off between simplicity and diagnostic detail.

When we want more diagnostic details, we have two general choices:

- Use multiple test methods instead of `runTest()`. Write multiple methods with names that start with `test_`. Remove any method named `runTest()`. The default test loader will execute each `test_` method separately, after rerunning the common `setUp()` method.

- Use multiple subclasses of the `GIVEN_Summary_WHEN_1k_samples_THEN_mean` class, each with a separate condition. Since the `setUp()` is common, this can be inherited.

Following the first alternative, the test class would look like this:

```
    class
GIVEN_Summary_WHEN_1k_samples_THEN_mean_median(unittest.TestCase):

        def setUp(self):
            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):
            self.assertEqual(500, self.summary.mean)

        def test_median(self):
            self.assertEqual(500, self.summary.median)
```

We've refactored the `setUp()` method to include the *Given* and *When* conditions of the test. The two independent *Then* conditions are refactored into their own separate `test_mean()` and `test_median()` methods. There is no `runTest()` method.

Since each test is run separately, we'll see separate error reports for problems with computing mean or computing median.

Some other assertions

The `TestCase` class defines numerous assertions that can be used as part of the *Then* condition; here are a few of the most commonly used:

- `assertEqual()` and `assertNotEqual()` compare actual and expected values using the default == operator.
- `assertTrue()` and `assertFalse()` require a single boolean expression.
- `assertIs()` and `assertIsNot()` use the `is` comparison to determine whether the two arguments are references to the same object.
- `assertIsNone()` and `assertIsNotNone()` use `is` to compare a given value with `None`.
- `assertIsInstance()` and `assertNotIsInstance()` use the `isinstance()` function to determine whether a given value is a member of a given class (or tuple of classes).
- `assertAlmostEquals()` and `assertNotAlmostEquals()` round the given values to seven decimal places to see whether most of the digits are equal.
- `assertRegex()` and `assertNotRegex()` compare a given string using a regular expression. This uses the `search()` method of the regular expression to match the string.
- `assertCountEqual()` compares two sequences to see whether they have the same elements, irrespective of order. This can be handy for comparing dictionary keys and sets too.

There are still more assertion methods. A number of them provide ways to detect exceptions, warnings, and log messages. Another group provides more type-specific comparison capabilities.

For example, the mode feature of the `Summary` class produces a list. We can use a specific `assertListEqual()` assertion to compare the results:

```
class GIVEN_Summary_WHEN_1k_samples_THEN_mode(unittest.TestCase):

    def setUp(self):
        self.summary = Summary()
        self.data = [500]*97
        # Build 993 more elements each item n occurs n times.
        for i in range(1,43):
            self.data += [i]*i
        random.shuffle(self.data)
        for sample in self.data:
            self.summary.add(sample)
```

```
    def test_mode(self):
        top_3 = self.summary.mode[:3]
        self.assertListEqual([(500,97), (42,42), (41,41)], top_3)
```

First, we built a collection of 1,000 values. Of those, 97 are copies of the number 500. The remaining 903 elements are copies of numbers between 1 and 42. These numbers have a simple rule—the frequency is the value. This rule makes it easier to confirm the results.

The `setUp()` method shuffled the data into a random order. Then the `Summary` object is built using the `add()` method.

We've used a `test_mode()` method. This allows for expansion to include other *Then* conditions on this test. In this case, we examined the first three values from the mode to be sure it had the expected distribution of values. The `assertListEqual()` compares two `list` objects; if either argument is not a list, we'll get a more specific error message showing that the argument wasn't of the expected type.

Separate tests directory

We've shown the `TestCase` class definitions in the same module as the code being tested. For small classes, this can be helpful. Everything related to the class can be found in one module file.

In larger projects, it's common practice to sequester the test files into a separate directory. The tests can be (and often are) extremely large. It's not unreasonable to have more test code than application code.

When this is done, we can rely on the discovery application that's part of the `unittest` framework. This application can search all of the files of a given directory for test files. Generally, these will be files with names that match the pattern `test*.py`. If we use a simple, consistent name for all test modules, then they can be located and run with a simple command.

The `unittest` loader will search each module in the 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 `os` command:

```
$ python3 -m unittest discover -s tests
```

This will locate all tests in the `tests` directory of a project.

See also

- We'll combine `unittest` and `doctest` in the *Combining unittest and doctest tests* recipe. We'll look at mocking external objects in the *Mocking external resources* recipe.

Combining unittest and doctest tests

In most cases, we'll have a combination of `unittest` and `doctest` test cases. For examples of doctest, see the *Using docstrings for testing* recipe. For examples of unittest, see the *Creating separate test modules and packages* recipe.

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`.

How can we combine all of these various tests 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.

The class started like this:

```
class Summary:
    '''Computes summary statistics.

    >>> 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
    '''
```

The methods have been omitted here so that we can focus on the example provided in the docstring.

In the *Creating separate test modules and packages* recipe, we wrote some `unittest.TestCase` classes to provide additional tests for this class. We created class definitions like this:

```
    class
GIVEN_Summary_WHEN_1k_samples_THEN_mean_median(unittest.TestCase):

        def setUp(self):
            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):
            self.assertEqual(500, self.summary.mean)

        def test_median(self):
            self.assertEqual(500, self.summary.median)
```

This test creates a `Summary` object; this is the *Given* aspect. It then adds a number of values to that `Summary` object. This is the *When* aspect of the test. The two `test_` methods implement two *Then* aspects of this test.

It's common to see a project folder structure that looks like this:

```
    git-project-name/
        statstools/
            summary.py
        tests/
            test_summary.py
```

We have a top-level folder, `git-project-name`, that matches the project name in the source code repository. We've assumed that Git is being used, but other tools are possible.

Within the top-level directory, we would have some overheads that are common to large Python projects. This would include files such as `README.rst` with a description of the project, `requirements.txt`, which can be used with `pip` to install extra packages, and perhaps `setup.py` to install the package into the standard library.

The directory `statstools` contains a module file, `summary.py`. This has our module that provides interesting and useful features. This module has docstring comments scattered around the code.

The directory `tests` contains another module file, `test_summary.py`. This has the `unittest` test cases in it. We've chosen the names tests and `test_*.py` so that they fit well with automated test discovery.

We need to combine all of the tests into a single, comprehensive test suite.

The example we'll show uses `ch11_r01` instead of some cooler name such as `summary`. A real project often has clever, meaningful names. The book content is quite large, and the names are designed to match the overall chapter and recipe outline.

How to do it...

1. For this example, we'll assume that the unittest test cases are in a file separate from the code being tested. We'll have `ch11_r01` and `test_ch11_r01`.

   To use doctest tests, import the `doctest` module. We'll be combining doctest examples with `TestCase` classes to create a comprehensive test suite:

   ```
   import unittest
   import doctest
   ```

   We'll assume that the `unittestTestCase` classes are already in place and we're adding more tests to the test suite.

2. Import the module which is being tested. This module will have strings with doctests in it:

   ```
   import ch11_r01
   ```

3. To implement the `load_tests` protocol, include the following function in the test module:

   ```
   def load_tests(loader, standard_tests, pattern):
       return standard_tests
   ```

   The function must have this name to be found by the test loader.

4. To incorporate doctest tests, an additional loader is required. We'll use the `doctest.DocTestSuite` class to create a suite. These tests will be added to the suite of tests provided as the `standard_tests` parameter value:

```
def load_tests(loader, standard_tests, pattern):
    dt = doctest.DocTestSuite(ch11_r01)
    standard_tests.addTests(dt)
    return standard_tests
```

   The `loader` argument is the test case loader currently being used. The `standard_tests` value will be all of the tests loaded by default. Generally, this is the suite of all subclasses of `TestCase`. The pattern value was the value provided to the loader.

We can now add `TestCase` classes and the overall `unittest.main()` function to create a comprehensive test module that includes the unittest `TestCase` plus all of the doctest examples.

This can be done by including the following code:

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

This allows us to run the module and execute the tests.

How it works...

When we evaluate `unittest.main()` inside this module, then the test loader process is limited to the current module. The loader will find all classes that extend `TestCase`. These are the standard tests that are provided to the `load_tests()` function.

We will supplement the standard tests with tests created by the `doctest` module. Generally, we'll be able to import the module under test and use the `DocTestSuite` to build a test suite from the imported module.

The `load_tests()` function is used automatically by the `unittest` module. This function can do a variety of things to the test suite that it's given. In this example, we've supplemented the test suite with additional tests.

There's more...

In some cases, a module may be quite complex; this can lead to multiple test modules. There might be several test modules with names such as `tests/test_module_feature.py` or something similar to show that there are multiple tests for separate features of a complex module.

In other cases, we might have a test module which has tests for several different but closely related modules. A package may be decomposed into multiple modules. A single test module, however, may cover all of the modules in the package being tested.

When combining many smaller modules, there may be multiple suites built in the `load_tests()` function. The body might look like this:

```
def load_tests(loader, standard_tests, pattern):
    for module in ch11_r01, ch11_r02, ch11_r03:
        dt = doctest.DocTestSuite(module)
        standard_tests.addTests(dt)
    return standard_tests
```

This will incorporate `doctests` from multiple modules.

See also

- For examples of doctest, see the *Using docstrings for testing* recipe. For examples of unittest, see the *Creating separate test modules and packages* recipe.

Testing things that involve dates or times

Many applications rely on `datetime.datetime.now()` to create a timestamp. When we use this with a unit test, the results are essentially impossible to predict. We have a 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 avoid `now()` and `utcnow()`. Instead of using these directly, we can create a factory function that emits timestamps. For test purposes, this function can be replaced with one that produces known results. It seems awkward to avoid using the `now()` method in a complex application.

Another option is to avoid direct use of the `datetime` class entirely. This requires designing classes and modules that wrap the `datetime` class. A wrapper class that produces known values for `now()` can then be used for testing. This, too, seems needlessly complex.

How can we work with `datetime` stamps?

Getting ready

We'll work with a small function that creates a `CSV` file. This file's name will include the date and time. We'll create files with names that look like this:

```
extract_20160704010203.json
```

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.

We'll use a function like this to create these files:

```
import datetime
import json
from pathlib import Path

def save_data(some_payload):
    now_date = datetime.datetime.utcnow()
    now_text = now_date.strftime('extract_%Y%m%d%H%M%S')
    file_path = Path(now_text).with_suffix('.json')
    with file_path.open('w') as target_file:
        json.dump(some_payload, target_file, indent=2)
```

This function has a use of `utcnow()`. It is technically possible to redesign the function and provide the timestamp as an argument. There are situations where this kind of redesign might be helpful. There's also a handy alternative to a redesign.

We will create a mock version of the `datetime` module, and patch the test context to use mock version instead of the actual version. This test will contain a mock class definition for the `datetime` class. Within that class, we'll provide a mock `utcnow()` method that will provide the expected response.

Since the function being tested creates a file, we need to think about the OS consequences of this. What should happen when a file with the same name already exists? Should an exception be raised? Should a suffix be added to the file name? Depending on our design decision, we may need to have two additional test cases:

- Given a directory empty of conflicts. In this case, a `setUp()` method to remove any previous test output. We may also want to create a `tearDown()` method to remove the file after a test.
- Given a directory with a conflicting name. In this case, a `setUp()` method will create a conflicting file. We may also want to create a `tearDown()` method to remove the file after a test.

For this recipe, we'll assume that duplicate file names don't matter. The new file should simply overwrite any previous file with no warning or notice. This is easy to implement, and often fits the real-world scenario where there's no reason to create multiple files less than 1 second apart in time.

How to do it...

1. For this example, we'll assume that the `unittest` test cases are the same module as the code being tested. Import the `unittest` and `unittest.mock` modules:

```
import unittest
from unittest.mock import *
```

   The `unittest` module is simply imported. To use the features of this module, we'll have to qualify the names with `unittest.`. The various names from `unittest.mock` were all imported so the names can be used without any qualifier. We'll use a number of features of the mock module, and the long qualifying name is awkward.

2. Include the code to be tested. This is shown previously.
3. Create the following skeleton for testing. We've provided one class definition, plus a main script that can be used to execute the tests:

```
class GIVEN_data_WHEN_save_data_THEN_file(unittest.TestCase):
    def setUp(self):
        '''GIVEN conditions for the test.'''

    def runTest(self):
        '''WHEN and THEN conditions for this test.'''
```

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

We didn't define a `load_tests()` function because we don't have any docstring tests to include.

4. The `setUp()` method will have several parts:

   - The sample data to be processed:

     ```
     self.data = {'primes': [2, 3, 5, 7, 11, 13, 17, 19]}
     ```

   - A mock object for the `datetime` module. This object provides precisely the features used by the unit under test. The `Mock` module contains a single `Mock` class definition for the `datetime` class. Within that class, it provides a single mock method, `utcnow()`, which always provides the same response:

     ```
     self.mock_datetime = Mock(
         datetime = Mock(
             utcnow = Mock(
                 return_value = datetime.datetime(2017, 7, 4, 1, 2,
3)
             )
         )
     )
     ```

   - Here's the expected file name given the `datetime` object shown above:

     ```
     self.expected_name = 'extract_20170704010203.json'
     ```

   - Some additional configuration processing is required to establish the *Given* condition. We'll remove any previous edition of the file to be completely sure that the test assertions aren't using a file from a previous test run:

     ```
     self.expected_path = Path(self.expected_name)
     if self.expected_path.exists():
         self.expected_path.unlink()
     ```

5. The `runTest()` method will have two parts:

   - The *When* processing. This will patch the current module, `__main__`, so that a reference to `datetime` will be replaced with the `self.mock_datetime` object. It will then execute the request in that patched context:

     ```
     with patch('__main__.datetime', self.mock_datetime):
         save_data(self.data)
     ```

   - The *Then* processing. In this case, we'll open the expected file, load the content, and confirm that the result matches the source data. This finishes with the necessary assertion. If the file doesn't exist, this will raise an `IOError` exception:

     ```
     with self.expected_path.open() as result_file:
         result_data = json.load(result_file)
     self.assertDictEqual(self.data, result_data)
     ```

How it works...

The `unittest.mock` module has two valuable components that we're using here—the `Mock` object definition and the `patch()` function.

When we create an instance of the `Mock` class, we must provide the methods and attributes of the resulting object. When we provide a named argument value, this will be saved as an attribute of the resulting object. Simple values become attributes of the object. Values which are based on a `Mock` object become method functions.

When we create an instance of `Mock` that provides the `return_value` (or `side_effect`) named argument value, we're creating a callable object. Here's an example of a mock object that behaves like a very dumb function:

```
>>> from unittest.mock import *
>>> dumb_function = Mock(return_value=12)
>>> dumb_function(9)
12
>>> dumb_function(18)
12
```

We created a mock object, `dumb_function`, that will behave like a callable—a function—that only returns the value `12`. For unit testing, this can be very handy, since the results are simple and predictable.

What's more important is this feature of the `Mock` object:

```
>>> dumb_function.mock_calls
[call(9), call(18)]
```

The `dumb_function()` tracked each call. We can then make assertions about these calls. For example, the `assert_called_with()` method checks the last call in the history:

```
>>> dumb_function.assert_called_with(18)
```

If the last call really was `dumb_function(18)`, then this succeeds silently. If the last call doesn't match the assertion, then this raises an `AssertionError` exception that the `unittest` module will catch and register as a test failure.

We can see more detail like this:

```
>>> dumb_function.assert_has_calls( [call(9), call(18)] )
```

This assertion checks the entire call history. It uses the `call()` function from the `Mock` module to describe the arguments provided in a function call.

The `patch()` function can reach into a module's context and change any reference in that context. In this example, we used `patch()` to tweak a definition in the `__main__` module—the one currently running. In many cases, we'll import another module, and will need to patch that imported module. It's crucial to reach out to the context that's in effect for the module under test and patch that reference.

There's more...

In this example, we created a mock for the `datetime` module that had a very narrow feature set.

The module had a single element which is an instance of the `Mock` class, named `datetime`. For the purposes of unit testing, a mocked class generally behaves like a function which returns an object. In this case, the class returned a `Mock` object.

The `Mock` object that stands in for the `datetime` class has a single attribute, `utcnow()`. We used the special `return_value` keyword when defining this attribute so that it would return a fixed `datetime` instance. We can extend this pattern and mock more than one attribute to behave like a function. Here's an example that mocks both `utcnow()` and `now()`:

```
        self.mock_datetime = Mock(
           datetime = Mock(
                utcnow = Mock(
                    return_value = datetime.datetime(2017, 7, 4, 1, 2, 3)
                ),
                now = Mock(
                    return_value = datetime.datetime(2017, 7, 4, 4, 2, 3)
                )
            )
        )
```

The two mocked methods, `utcnow()` and `now()`, each create a different `datetime` object. This allows us to distinguish between the values. We can more easily confirm the correct operation of a unit test.

Note that all of this `Mock` object construction executes during the `setUp()` method. This is long before the patching done by the `patch()` function. During `setUp()`, the `datetime` class is available. In the context of the `with` statement, the `datetime` class is unavailable, and is replaced by the `Mock` object.

We can add the following assertion to confirm that the `utcnow()` function was used properly by the unit under test:

```
        self.mock_datetime.datetime.utcnow.assert_called_once_with()
```

This will examine the `self.mock_datetime` mock object. It looks inside this object at the `datetime` attribute, which we've defined to have a `utcnow` attribute. We expect that this is called exactly once with no argument values.

If the `save_data()` function doesn't make a proper call to `utcnow()`, this assertion will detect that failure. It's essential to test both sides of the interface. This leads to two parts to a test:

- The result of the mocked `datetime` was used properly by the unit being tested
- The unit being tested made appropriate requests to the mocked `datetime` object

In some cases, we might need to confirm that an obsolete or deprecated method is never called. We might have something like this to confirm that another method is not used:

```
self.assertFalse( self.mock_datetime.datetime.called )
```

This kind of testing is used when refactoring software. In this example, the previous version may have used the `now()` method. After the change, the function is required to use the `utcnow()` method. We've included a test to be sure that the `now()` method is no longer being used.

See also

- The *Creating separate test modules and packages* recipe has more information about 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 random order. In many statistical tests, repeated random shuffling or random subset calculations are done. When we want to test one of the algorithms, the results are essentially impossible to predict.

We have two choices for trying to make the `random` module predictable enough to write meaningful unit tests:

- Set a known seed value, this is common, and we've made heavy use of this in many other recipes.
- Use `unittest.mock` to replace the `random` module with something much less random.

How can we 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 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. This overall set of resample metrics shows the likely variance of the measure for the overall population.

In order to be sure that a resampling algorithm works, 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 reason to believe that a truly random version will also work.

Here's our candidate resampling function. We need to validate this to be sure that it properly does sampling with replacement:

```
def resample(population, N):
    for i in range(N):
        sample = random.choice(population)
        yield sample
```

We would normally apply this `resample()` function to populate a `Counter` object that tracks each distinct value for a particular measure such as the mean. The overall resampling procedure looks like this:

```
mean_distribution = Counter()
for n in range(1000):
    subset = list(resample(population, N))
    measure = round(statistics.mean(subset), 1)
    mean_distribution[measure] += 1
```

This evaluates the `resample()` function `1,000` times. This will lead to a number of subsets, each of which may have a distinct value for the mean. These values are used to populate the `mean_distribution` object.

The histogram for `mean_distribution` will provide a meaningful estimate for population variance. This estimate of the variance will help show the population's most likely actual mean value.

How to do it...

1. Define an outline of the overall test class:

```
class GIVEN_resample_WHEN_evaluated_THEN_fair(unittest.TestCase):
    def setUp(self):

    def runTest(self):
```

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

We've included a main program so that we can simply run the module to test it. This is handy when working in tools such as IDLE; we can use the *F5* key to test the module after making a change.

2. Define a mock version of the `random.choice()` function. We'll provide a mock data set, `self.data`, and a mock response to the `choice()` function:

```
self.expected_resample_data.self.data = [2, 3, 5, 7, 11, 13, 17,
19]
self.expected_resample_data = [23, 29, 31, 37, 41, 43, 47, 53]
self.mock_random = Mock(
    choice = Mock(
        side_effect = self.expected_resample_data
    )
)
```

We've defined the `choice()` function using the `side_effect` attribute. This will return values one at a time from the given sequence. We've provided eight mock values that are distinct from the source sequence so that we readily identify the outputs from the `choice()` function.

3. Define the *When* and *Then* aspects of the test. In this case, we'll patch the `__main__` module to replace the reference to the `random` module. The test can then establish that the result has the expected set of values and that the `choice()` function was called multiple times:

```
with patch('__main__.random', self.mock_random):
    resample_data = list(resample(self.data, 8))

self.assertListEqual(self.expected_resample_data, resample_data)
self.mock_random.choice.assert_has_calls( 8*[call(self.data)] )
```

How it works...

When we create an instance of the `Mock` class, we must provide the methods and attributes of the resulting object. When the `Mock` object includes a named argument value, this will be saved as an attribute of the resulting object.

When we create an instance of `Mock` that provides the `side_effect` named argument value, we're creating a callable object. The callable object will return a value from the `side_effect` list each time the `Mock` object is called.

Here's an example of a mock object that behaves like a very dumb function:

```
>>> from unittest.mock import *
>>> dumb_function = Mock(side_effect=[11,13])
>>> dumb_function(23)
11
>>> dumb_function(29)
13
>>> dumb_function(31)
Traceback (most recent call last):
  ... (traceback details omitted)
StopIteration
```

First, we created a `Mock` object and assigned it to the name `dumb_function`. The `side_effect` attribute of this `Mock` object provides a short list of two distinct values that will be returned.

The example then evaluates `dumb_function()` two times with two different argument values. Each time, the next value is returned from the `side_effect` list. The third attempt raises a `StopIteration` exception that becomes a test failure.

This behavior allows us to write a test that detects certain kinds of improper uses of a function or a method. If the function is called too many times, an exception will be raised. Other improper uses must be detected with the various kinds of assertions that can be used for `Mock` objects.

There's more...

We can easily replace other features of the `random` module with mock objects that provide appropriate behavior without actually being random. We could, for example, replace the `shuffle()` function with a function that provides a known order. We might follow the above test design pattern like this:

```
self.mock_random = Mock(
    choice = Mock(
        side_effect = self.expected_resample_data
    ),
    shuffle = Mock(
        return_value = self.expected_resample_data
    )
```

```
)
```

This mock `shuffle()` function returns a distinct set of values that can be used to confirm that some process is making proper use of the `random` module.

See also

- The *Using set methods and operators*, *Creating dictionaries – inserting and updating* recipes in `Chapter 4`, *Built-in Data Structures – list, set, dict*, and the *Using cmd for creating command-line applications* recipe in `Chapter 5`, *User Inputs and Outputs*, show how to seed the random number generator to create a predictable sequence of values.
- In `Chapter 6`, *Basics of Classes and Objects*, there are several other recipes that show the alternative approach, for example, *Using a class to encapsulate data + processing*, *Designing classes with lots of processing*, *Optimizing small objects with __slots__*, and *Using properties for lazy attributes*.
- Also, in `Chapter 7`, *More Advanced Class Design*, see *Choosing between inheritance and extension – the is-a question*, *Separating concerns via multiple inheritance*, *Leveraging Python's duck typing*, *Creating a class which has orderable objects*, and *Defining an ordered collection* recipes.

Mocking external resources

The *Testing things that involve dates or times* and *Testing things that involve randomness* recipes show techniques for mocking relatively simple objects. In the case of the *Testing things that involve dates or times* recipe, the object being mocked is essentially stateless, and a single return value works nicely. In the *Testing things that involve randomness* recipe, the object has a state change, but the state change does not depend on any input arguments.

In these simpler cases, a test provides a series of requests to an object. Mock objects can be built which are based on a known and carefully planned sequence of state changes. The test case follows the object's internal state changes precisely. This is sometimes called white box testing because the implementation details of the object under test are required to define the test sequence and the mock objects.

In some cases, however, a test scenario may not involve a well-defined sequence of state changes. The unit under test may make requests in a difficult-to-predict order. This is sometimes a consequence of black box testing where the implementation details are not known.

How can we create more sophisticated mock objects that have internal state and make their own internal state changes?

Getting ready

We'll look at mocking a stateful RESTful web services request. In this case, we'll be using a database API for the Elastic database. See `https://www.elastic.co/` for more information on this database. The database has the advantage of working with simple RESTful web services. These can easily be mocked to simple, fast unit tests.

For this recipe, we'll test a function that uses the RESTful API to create records. **Representational State Transfer** (**REST**) is a technique for using **Hypertext Transfer Protocol** (**HTTP**) to transfer a representation of an object's state between processes. To create a database record, for example, a client will transfer a representation of an object's state to the database server, using HTTP `POST` requests. In many cases, JSON notation is used to represent object state.

Testing this function will involve mocking one part of the `urllib.request` module. Replacing the `urlopen()` function will allow a test case to simulate database activity. This will allow us to test a function that depends on web services without actually making potentially expensive or slow external requests.

There are two overall approaches to working with the elastic search API in our application software:

- We can install the Elastic database on our laptop or some server to which we have access. The installation is a two-part process that starts by installing a proper **Java Developer Kit** (**JDK**) and then installs the ElasticSearch software. We won't go into details here, because we have an alternative which seems simpler.

  The URLs to create and access objects on a local computer will look like this:

  ```
  http://localhost:9200/eventlog/event/
  ```

  The requests will use a number of data items in the body of the request. These requests don't require any of the HTTP headers for security or authentication purposes.

- We can use a hosting service such as `http://orchestrate.io`. This requires signing up with the service to get an API key instead of installing software. An API key grants access to a defined application. Within the application, a number of collections can be created. Since we won't have to install additional software, this seems like a handy way to proceed.

The URLs to work with objects on a remote server will look like this:

```
https://api.orchestrate.io/v0/eventlog/
```

The requests will also use a number of HTTP headers to provide information to the host. Next, We'll look at details of this service.

The data payload for the document to be created will look like this:

```
{
    "timestamp": "2016-06-15T17:57:54.715",
    "levelname": "INFO",
    "module": "ch09_r10",
    "message": "Sample Message One"
}
```

This JSON document represents a log entry. This came from the `sample.log` file used in earlier examples. This document can be understood as a specific instance of the event type that will be saved in the `eventlog` index of the database. The object has four attributes with string values.

The *Reading complex formats using regular expressions* recipe in `Chapter 8`, *Input/Output, Physical Format, and Logical Layout*, shows how to parse a complex log file. In the *Using multiple contexts for reading and writing files* recipe, the complex log records were written to a `CSV` file. In this example, we'll show how the log records could be placed into cloud-based storage using a database such as Elastic.

Creating an entry document in the entrylog collection

We're going to create entry documents in an `entrylog` collection in the database. An HTTP `POST` request is used to create new items. The response of `201 Created` will indicate that the database created the new event.

To use the `orchestrate.io` database service, each request has a base URL. We can define this with a string like this:

```
service = "https://api.orchestrate.io"
```

The `https` scheme is used so that the **Secure Socket Layer** (**SSL**) is used to assure that the data is private between client and server. The host name is `api.orchestrate.io`. Each request will have a URL based on this base service definition.

The HTTP headers for each request will look like this:

```
headers = {
    'Accept': 'application/json',
    'Content-Type': 'application/json',
    'Authorization': basic_header(api_key, '')
}
```

The `Accept` header shows what kind of response is expected. The `Content-Type` header shows what kind of document representation is being used for the content. These two headers direct the database to use JSON representation for object state.

The `Authorization` header is how the API key is sent. The value for this header is a rather complex string. It's easiest to build the encoded API key string code like the following:

```
import base64
def basic_header(username, password):
    combined_bytes = (username + ':' + password).encode('utf-8')
    encoded_bytes = base64.b64encode(combined_bytes)
    return 'Basic ' + encoded_bytes.decode('ascii')
```

This snippet of code will combine a username and password into a single string of characters, and then encode those characters into a stream of bytes using the `UTF-8` encoding scheme. The `base64` module creates a second stream of bytes. In this output stream, four bytes will contain the bits that comprise three input bytes. The bytes are chosen from a simplified alphabet. This value is then converted back into Unicode characters along with the keyword `'Basic '`. This value can be used with the `Authorization` header.

It's easiest to work with a RESTful API by creating a `Request` object. The class is defined in the `urllib.request` module. The `Request` object combines the data, URL, and headers, and names a specific HTTP method. Here's the code to create a `Request` instance:

```
data_document = {
    "timestamp": "2016-06-15T17:57:54.715",
    "levelname": "INFO",
    "module": "ch09_r10",
    "message": "Sample Message One"
}

headers={
    'Accept': 'application/json',
```

```
        'Content-Type': 'application/json',
        'Authorization': basic_header(api_key, '')
    }

    request = urllib.request.Request(
        url=service + '/v0/eventlog',
        headers=headers,
        method='POST',
        data=json.dumps(data_document).encode('utf-8')
    )
```

The request object includes four elements:

- The value of the `url` parameter is the base service URL plus the collection name, `/v0/eventlog`. The `v0` in the path is the version information which must be provided with each request.
- The `headers` parameter includes the `Authorization` header which has the API Key which authorizes access to the application.
- The method of `POST` will create a new object in the database.
- The `data` parameter is the document to save. We've converted a Python object to a string in JSON notation. Then encoded the Unicode characters into bytes using `UTF-8` encoding.

Seeing a typical response

The processing involves sending the request and receiving a response. The `urlopen()` function accepts the `Request` object as an argument; this builds the request that's transmitted to the database server. The response from the database server will include three elements:

- A status. This includes both a numeric code and a reason string. When creating a document, the expected response code is `201` and the string is `CREATED`. For many other requests, the code is `200` and the string is `OK`.
- The response will also have headers. For a creation request, these will include the following:

```
        [
         ('Content-Type', 'application/json'),
         ('Location',
'/v0/eventlog/12950a87ef024e43/refs/8e50b6bfc50b2dfa'),
         ('ETag', '"8e50b6bfc50b2dfa"'),
         ...
         ]
```

The `Content-Type` header tells us that the content is encoded in JSON. The `Location` header provides a URL that can be used to retrieve the object which is created. It also provides an `ETag` header, which is a hashed summary of the current state of the object; this helps to support caching local copies of an object. Other headers may be present; we've just shown `...` in the example.

- The response may have a body. If present, this will be a JSON-encoded document (or documents) retrieved from the database. The body must be read with the `read()` method of the response. A body can be quite large; a `Content-Length` header provides the exact number of bytes.

Client class for database access

We'll define a simple class for database access. A class can provide context and status information for multiple related operations. When working with the Elastic database, an access class can create the request headers dictionary just once and reuse it in multiple requests.

Here's the essence of a database client class. We'll show this in several sections. First, the overall class definition:

```
class ElasticClient:
    service = "https://api.orchestrate.io"
```

This defines a class-level variable, `service`, with the scheme and hostname. The initialization method, `__init__()`, can build the headers that are used by the various database operations:

```
def __init__(self, api_key, password=''):
    self.headers = {
        'Accept': 'application/json',
        'Content-Type': 'application/json',
        'Authorization': ElasticClient.basic_header(api_key, password),
    }
```

This method takes the API key and creates a set of headers that relies on HTTP basic authorization. The password is not used by the orchestrate service. We've included it, however, because the username and password are used for the example unit test case.

Here's the method:

```
@staticmethod
def basic_header(username, password=''):
    """
    >>> ElasticClient.basic_header('Aladdin', 'OpenSesame')
    'Basic QWxhZGRpbjpPcGVuU2VzYW1l'
    """
    combined_bytes = (username + ':' + password).encode('utf-8')
    encoded_bytes = base64.b64encode(combined_bytes)
    return 'Basic ' + encoded_bytes.decode('ascii')
```

This function can combine a username and a password to create the value for the HTTP `Authorization` header. The `orchestrate.io` API uses an assigned API key as the username; the password is a zero-length string, `''`. The API key is assigned when someone signs up for their service. The free level of service allows a reasonable number of transactions and a comfortably small database.

We've included a unit test case in the form of a docstring. This provides evidence that the results are correct. The test case comes from the Wikipedia page on HTTP basic authentication.

The final part is a method to load one data item into the `eventlog` collection of the database:

```
def load_eventlog(self, data_document):
    request = urllib.request.Request(
        url=self.service + '/v0/eventlog',
        headers=self.headers,
        method='POST',
        data=json.dumps(data_document).encode('utf-8')
    )

    with urllib.request.urlopen(request) as response:
        assert response.status == 201, "Insertion Error"
        response_headers = dict(response.getheaders())
        return response_headers['Location']
```

This function builds a `Request` object with the four required pieces of information—the full URL, the HTTP headers, the method string, and the encoded data. In this case, the data is encoded as a JSON string, and the JSON string encoded into bytes using the `UTF-8` encoding scheme.

Evaluating the `urlopen()` function sends the request and retrieves a response object. This object is used as a context manager. The `with` statement assures that the resources are released properly even if there is an exception raised during response processing.

A `POST` method should respond with a status of `201`. Any other status is a problem. In this code, the status is checked with an `assert` statement. It might be better to provide a message such as `Expected 201 status, got {}.format(response.status)`.

The headers are then examined to get the `Location` header. This provides a URL fragment for locating the object which was created.

How to do it...

1. Create the database access module. This module will have the `ElasticClient` class definition. It will also have any additional definitions that this class needs.
2. This recipe will use `unittest` and `doctest` to create a unified suite of tests. It will use the `Mock` class from `unittest.mock`, as well as `json`. Since this module is separate from the unit under test, it needs to import `ch11_r08_load`, which has the class definitions that will be tested:

```
import unittest
from unittest.mock import *
import doctest
import json
import ch11_r08_load
```

3. Here's the overall framework for a test case. We'll fill in the `setUp()` and `runTest()` methods of this test below. The name shows that we're given an instance of `ElasticClient`, when we invoke `load_eventlog()`, then a proper RESTful API request was made:

```
        class
GIVEN_ElasticClient_WHEN_load_eventlog_THEN_request(unittest.TestCase):

            def setUp(self):

            def runTest(self):
```

4. The first part of the `setUp()` method is a mock context manager that provides responses similar to the `urlopen()` function:

```
    def setUp(self):
        # The context manager object itself.
        self.mock_context = Mock(
            __exit__ = Mock(return_value=None),
            __enter__ = Mock(
                side_effect = self.create_response
            ),
        )

        # The urlopen() function that returns a context.
        self.mock_urlopen = Mock(
            return_value = self.mock_context,
        )
```

When `urlopen()` is called, the return value is a response object which behaves like a context manager. The best way to mock this is to return a mock context manager. The mock context manager's `__enter__()` method does the real work to create a response object. In this case, the `side_effect` attribute identifies a helper function that will be called to prepare the result from calling the `__enter__()` method. The `self.create_response` has not been defined yet. We'll use a function, defined as follows.

5. The second part of the `setUp()` method is some mock data to be loaded:

```
        # The test document.
        self.document = {
            "timestamp": "2016-06-15T17:57:54.715",
            "levelname": "INFO",
            "module": "ch09_r10",
            "message": "Sample Message One"
        }
```

In a more complex test, we might want to simulate a large, iterable collection of documents.

6. Here's the `create_response()` helper method that builds response-like objects. The response objects can be complex, so we've defined a function to create them:

```
        def create_response(self):
            self.database_id =
hex(hash(self.mock_urlopen.call_args[0][0].data))[2:]
            self.location = '/v0/eventlog/{id}'.format(id=self.database_id)
            response_headers = [
```

```
                ('Location', self.location),
                ('ETag', self.database_id),
                ('Content-Type', 'application/json'),
            ]
            return Mock(
                status = 201,
                getheaders = Mock(return_value=response_headers)
            )
```

This method uses `self.mock_urlopen.call_args` to examine the last call to this `Mock` object. This call's arguments are a tuple of positional argument values and keyword arguments. The first `[0]` index picks the positional argument value from the tuple. The second `[0]` index picks the first positional argument value. This will be the object to be loaded to the database. The value of the `hex()` function is a string that includes a `0x` prefix that we discard.

In a more complex test, it may be necessary for this method to keep a cache of objects loaded into the database to make more accurate database-like responses.

7. The `runTest()` method makes a patch to the module under test. It locates the reference from `ch11_r08_load` to `urllib.request` and to the `urlopen()` function. This is replaced with the `mock_urlopen` replacement:

```
        def runTest(self):
            with patch('ch11_r08_load.urllib.request.urlopen',
self.mock_urlopen):
                client = ch11_r08_load.ElasticClient('Aladdin',
'OpenSesame')
                response = client.load_eventlog(self.document)

            self.assertEqual(self.location, response)

            call_request = self.mock_urlopen.call_args[0][0]
            self.assertEqual(
                'https://api.orchestrate.io/v0/eventlog',
call_request.full_url)
            self.assertDictEqual(
                {'Accept': 'application/json',
                 'Authorization': 'Basic QWxhZGRpbjpPcGVuU2VzYW1l',
                 'Content-type': 'application/json'
                },
                 call_request.headers)
            self.assertEqual('POST', call_request.method)
            self.assertEqual(
                json.dumps(self.document).encode('utf-8'),
call_request.data)
```

```
            self.mock_context.__enter__.assert_called_once_with()
            self.mock_context.__exit__.assert_called_once_with(None, None,
None)
```

This test follows the `ElasticClient` requirements of first creating a client object. Instead of using an actual API key, this uses a username and password that will create a known value for the `Authorization` header. The result of the `load_eventlog()` is a response-like object that can be examined to see whether it has the proper values.

All of this interaction will be done through the mock objects. We can use the various assertions to confirm that a proper request object was created. The test examines four attributes of the request object and also checks to be sure that the context was used properly.

8. We'll also define a `load_tests()` function to combine this `unittest` suite with any test examples found docstrings of `ch11_r08_load`:

```
def load_tests(loader, standard_tests, pattern):
    dt = doctest.DocTestSuite(ch11_r08_load)
    standard_tests.addTests(dt)
    return standard_tests
```

9. Finally, we'll provide the overall main program to run the complete suite. This makes it easy to run the test module as a standalone script:

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

How it works...

This recipe combines a number of `unittest` and `doctest` features to create a sophisticated test case. The features include:

- Creating a context manager
- Using the side-effect feature to create a dynamic, stateful test
- Mocking a complex object
- Using the load tests protocol to combine doctest and unittest cases

We'll look at each of these features separately.

Creating a context manager

The context manager protocol wraps an object in an additional layer of indirection. See the *Reading and writing files with context managers* and *Using multiple contexts for reading and writing files* recipes for more information on this. The core features that must be mocked are the `__enter__()` and `__exit__()` methods.

The pattern for mock context managers looks like this:

```
self.mock_context = Mock(
    __exit__ = Mock(return_value=None),
    __enter__ = Mock(
        side_effect = self.create_response
        # or
        # return_value = some_value
    ),
)
```

The context manager object has two attributes. The `__exit__()` will be called once. A return value of `True` will silence any exception. The return value of `None` or `False` will allow exceptions to propagate.

The `__enter__()` method returns the object which is assigned in the `with` statement. In this example, we used the `side_effect` attribute and provided a function so that a dynamic result can be computed.

A common alternative for the `__enter__()` method is to use a fixed `return_value` attribute and provide the same manager object each time. It's also possible to provide a sequence with `side_effect`; in this case, each time the method is called, another object from the sequence is returned.

Creating a dynamic, stateful test

In many cases, the test can use a static, fixed set of objects. The mock responses can be defined in the `setUp()` method. In some cases, however, an object's state may need to change during the operations of a complex test. In this case, the `side_effect` attribute of a `Mock` object can be used to track a state change.

In this example, the `side_effect` attribute used the `create_response()` method to build a dynamic response. A function referenced by `side_effect` can do anything; this can be used to update dynamic state information that is used to compute complex responses.

There's a fine line here. A complex test case can introduce its own bugs. It's generally a good idea to keep the test cases as simple as possible to avoid having to write `meta tests` to test the test cases.

For non-trivial tests, it's important to be sure that the test can actually fail. Some tests involve inadvertent tautologies. It's possible to create a contrived test that is as meaningful as `self.assertEqual(4, 2+2)`. To be sure the test actually uses the unit under test, it should fail when the code is missing or has a bug injected into it.

Mocking a complex object

The response object from `urlopen()` has a large number of attributes and methods. For our unit test, we only needed to set a few of these features.

We used the following:

```
return Mock(
    status = 201,
   getheaders = Mock(return_value=response_headers)
)
```

This created a `Mock` object with two attributes:

- The `status` attribute had a simple numeric value.
- The `getheaders` attribute used a `Mock` object with the `return_value` attribute to create a method function. This method function returned the dynamic `response_headers` value.

The value of `response_headers` is a sequence of two-tuples that has *(key, value)* pairs. This representation of the response headers can be transformed into a dictionary very easily.

The object is built like this:

```
response_headers = [
    ('Location', self.location),
    ('ETag', self.database_id),
    ('Content-Type', 'application/json'),
]
```

This sets three headers: `Location`, `ETag`, and `Content-Type`. Other headers may be needed, depending on the test case. It's important not to clutter the test case with headers that are not used. This kind of clutter can lead to bugs in the test itself.

The database id and location are based on the following calculation:

```
hex(hash(self.mock_urlopen.call_args[0][0].data))[2:]
```

This uses `self.mock_urlopen.call_args` to examine the arguments provided to the test case. The value of the `call_args` attribute is a two-tuple with the positional and keyword argument values. The positional arguments are a tuple too. This means that `call_args[0]` is the positional argument and `call_args[0][0]` is the first positional argument. This will be the document that's loaded to the database.

Many Python objects have hash values. In this case, the object is expected to be a string created by the `json.dumps()` function. The hash value for this string is a large number. The hex value of that number will be a string with a `0x` prefix. We'll use the `[2:]` slice to ignore the prefix. For information on this, see the *Rewriting an immutable string* recipe in `Chapter 1`, *Numbers, Strings, and Tuples*.

Using the load_tests protocol

A complex module will include class and function definitions. The module as a whole needs a descriptive docstring. Each class and function needs a docstring. Each method within a class also needs a docstring. These will provide essential information about the module, class, function, and method.

In addition, each docstring can include an example. The examples can be tested by the `doctest` module. See the *Using docstrings for testing* recipe for examples. We can combine the docstring example tests without more complex unit tests. See the *Combining unittest and doctest tests* recipe for more information on how to do this.

There's more...

The `unittest` module can be used to construct integration tests too. The idea of an integration test is to avoid mocks and actually use the real external service in a test mode. This can be slow or expensive; it's common to avoid integration testing until after all of the unit tests provide confidence that the software is likely to work.

We might, for example, create two applications with `orchestrate.io`—the real application and a test application. This will provide us with two API keys. The test key would be used so that the database can be reset to its initial state without creating problems for actual users of the real data.

We can control this using the `unittest`, `setUpModule()`, and `tearDownModule()` functions. The `setUpModule()` function is executed prior to all of the tests in a given module file. This is a handy way to set the database to a known state.

We can also remove the database with the `tearDownModule()` function. This can be handy for removing unneeded resources created by the test. It's sometimes more helpful to leave resources around for debugging purposes. For this reason, the `tearDownModule()` function may not be as useful as the `setUpModule()` function.

See also

- The *Testing things that involve dates or times* and *Testing things that involve randomness* recipes show techniques.
- The *Reading complex formats using regular expressions* recipe in `Chapter 8`, *Input/Output, Physical Format, and Logical Layout*, shows how to parse a complex log file. In the *Using multiple contexts for reading and writing files* recipe, the complex log records were written to a `CSV` file.
- For information on chopping up strings to replace parts, see the *Rewriting an immutable string* recipe.
- Elements of this can be tested by the `doctest` module. See the *Using docstrings for testing* recipe for examples. It's also important to combine these tests with any doctests. See the *Combining unittest and doctest tests* recipe for more information on how to do this.

10
Web Services

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

- Implementing web services with WSGI
- Using the Flask framework for RESTful APIs
- Parsing the query string in a request
- Making REST requests with urllib
- Parsing the URL path
- Parsing a JSON request
- Implementing authentication for web services

Introduction

Providing web services involves solving several interrelated problems. There are a number of applicable protocols that must be followed, each with its own unique design considerations. The core of web services are the various standards that define the HTTP.

HTTP involves two parties; a client and a server:

- The client makes requests of the server
- The server sends responses back to the client

The relationship is highly asymmetric. We expect a server to process concurrent requests from multiple clients. Because the client requests arrive asynchronously, the server cannot easily distinguish those requests that originate from a single human user. The idea of a human user's session is implemented by designing a server that provides a session token (or cookie) to track the human's sense of current state.

The HTTP protocol is flexible and extensible. One popular use case for HTTP is to serve content in the form of web pages. Web pages are generally encoded as HTML documents, often with links to graphics, style sheets, and JavaScript code. We've looked at parsing HTML in the *Reading HTML documents* recipe from `Chapter 8`, *Input/Output, Physical Format, and Logical Layout*.

Serving web page content further decomposes into two kinds of content:

- Static content is essentially a download of files. A program such as GUnicorn, NGINGX, or Apache HTTPD can reliably serve static files. Each URL defines a path to a file, and the server downloads the file to the browser.
- Dynamic content is built by an application as needed. In this case, we'll use a Python application to build unique HTML (or possibly the graphics) in response to a request.

The other very popular use case for HTTP is to provide web services. In this case, the standard HTTP requests and responses will exchange data in formats other than HTML. One of the most popular formats for encoding information is JSON. We've looked at processing JSON documents in the *Reading JSON documents* recipe from `Chapter 8`, *Input/Output, Physical Format, and Logical Layout*.

Web services can be seen as a variation on using HTTP to serve dynamic content. A client can prepare documents in JSON. The server includes a Python application that creates response documents, also in JSON notation.

In some cases, the services have a very narrow focus. It's possible to bundle a service and database persistence into a single package. This might involve creating a server that has an NGINX-based web interface plus a database using MongoDB or Elastic. The entire package—web service plus persistence—can be called a **microservice**.

The documents exchanged by a web service encode a representation of an object's state. A client application in JavaScript may have an object state that is sent to a server. A server in Python may transfer a representation of object state to a client. This is called **Representational State Transfer** (**REST**). A service using REST processing is often called RESTful.

Handling HTTP for HTML or JSON can be designed as a number of transformation functions. The idea is as follows:

```
response = F(request, persistent state)
```

The response is built from the request by some function, `F(r, s)`, which relies on the request plus some persistent state in a database on the server.

These functions form nested shells or wrappers around a core service. For example, the core processing may be wrapped with additional steps to be sure that the user making the request is authorized to change the database state. We might summarize this as follows:

```
response = auth(F(request, persistent state))
```

The authorization processing may be wrapped in processing to authenticate user's credentials. All of this may be further wrapped in a shell that assures that the client application software expects responses in JSON notation. Using multiple layers like this can provide consistent operation for many different core services. The overall process might start looking like this:

```
response = JSON( user( auth( F(request, persistent state) ) ) )
```

This kind of design fits naturally with a stack of transformational functions. This idea gives us some guidance in ways to design complex web services that include many protocols and many rules for creating a valid response.

A good RESTful implementation should also provide a great deal of information about the service. One way to provide this information is through the OpenAPI specification. For information on the OpenAPI (Swagger) specification, see `http://swagger.io/specification/`.

The core of the OpenAPI specification is a JSON schema specification. For more information on this, see `http://json-schema.org`.

The two foundational ideas are as follows:

1. We write in JSON a specification for the requests that are sent to the service and the responses provided by the service.
2. We provide the specification at a fixed URL, often `/swagger.json`. This can be queried by a client to determine the details of how the service works.

Creating Swagger documents can be challenging. The `swagger-spec-validator` project can help. See `https://pypi.python.org/pypi/swagger-spec-validator`. This is a Python package that we can use to confirm that a Swagger specification meets the OpenAPI requirements.

In this chapter, we'll look at a number of recipes for creating RESTful web services and also serving static or dynamic content.

Implementing web services with WSGI

Many web applications will have several layers. The layers can often be summarized into three common patterns:

- A presentation layer might run on a mobile device or a website. This is the visible, external view.
- An application layer is often implemented as web services. This layer does the processing for the web or mobile presentation.
- A persistence layer handles retention of data and transaction state over a single session as well as across multiple sessions from a single user. This will support the application layer.

A Python-based website or web services application will adhere to the **Web Services Gateway Interface** (**WSGI**) standard. This provides a uniform way for a frontend web server such as Apache HTTPD, NGINX, or GUnicorn to use Python to provide the dynamic content.

Python has a wide variety of RESTful API frameworks. In the *Using the Flask framework for RESTful APIs* recipe, we'll look at Flask. In some cases, however, the core WSGI features are all we need.

How can we create applications that support layered composition following the WSGI standard?

Getting ready

The WSGI standard defines an overall framework for composable web applications. The idea behind this is to define each application so that it stands alone and can be trivially connected to other applications. The overall website is built from a collection of shells or wrappers.

This is a bare-bones approach to web server development. WSGI isn't a sophisticated framework; it's a minimal standard. We'll look at some ways to simplify the design using a better framework in the *Using the Flask framework for RESTful APIs* recipe.

The essence of web services are the HTTP request and response. A server receives requests and creates responses. The HTTP request includes several pieces of data:

- The URL for the resource. A URL can be as complex as `http://www.example.com:8080/?query#fragment`. There are several parts to a URL:
    - The scheme `http`: This ends with `:`.
    - The host `www.example.com`: This is prefixed with `//`. It may include an optional port number. In this case, it's `8080`.
    - The path to the resource: The `/` character in this example. The path, in some form, is required. It is often more complex than a simple `/`.
    - A query string prefaced with `?`: In this example, the query string is just the key `query` with no value.
    - A fragment identifier prefaced with `#`: In this example, the fragment is `fragment`. For HTML documents, this can be the `id` value of a particular tag; the browser will scroll to the named tag.

  Almost all of these URL elements are optional. We can make use of the query string (or the fragment) to provide additional format information about the request.

The WSGI standard requires that the URL is parsed. The various pieces put into the environment. Each piece will be assigned a separate key:

- **Methods**: Common HTTP methods include `HEAD`, `OPTIONS`, `GET`, `POST`, `PUT`, and `DELETE`.
- **Request headers**: The headers are additional information that support the request. Headers are used, for example, to define the kind of content that can be accepted.
- **Attached content**: A request might include input from an HTML form, or a file to be uploaded.

The HTTP response is similar to a request in many ways. It contains response headers and the response body. The headers will include details such as the encoding of the content so that the client can render it correctly. If a server is providing HTML content and is maintaining a server session, then the cookies are sent in headers as part of each request and response.

WSGI is designed to help create application components that can be used to build larger and more sophisticated applications. A WSGI application generally acts like a wrapper, insulating other applications from bad requests, unauthorized users, or unauthenticated users. To do this, each WSGI application must follow a common, standard definition. Each application must be either a function or a callable object, that has the following signature:

```
def application(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/plain')])
    return iterable_strings
```

The `environ` parameter is a dictionary that includes information about the request. This includes all of the HTTP details, plus the OS context, plus the WSGI server context. The `start_response` parameter is a function that must be called prior to returning the response body. This provides the status and the headers for the response.

The return value from the WSGI application function is the HTTP response body. This is generally a sequence of strings or an iterable over string values. The idea here is that a WSGI application might be part of a larger container that will stream the response in pieces from the server to the client as the response is being built.

Since all WSGI applications are callable functions, they can be composed easily. A complex web server might have several WSGI components to handle details of authentication, authorization, standard headers, audit logging, performance monitoring, and so on. These aspects are generally independent of the underlying content; they're universal features of all web applications or RESTful services.

We'll look at a relatively simple web service that emits playing cards from either a deck or a shoe. We'll rely on the `Card` class definition from the *Optimizing small objects with __slots__* recipe from `Chapter 6`, *Basics of Classes and Objects*. Here's the core `Card` class with rank and suit information:

```
class Card:
    __slots__ = ('rank', 'suit')
    def __init__(self, rank, suit):
        self.rank = int(rank)
        self.suit = suit
    def __repr__(self):
        return ("Card(rank={self.rank!r}, "
         "suit={self.suit!r})").format(self=self)
    def to_json(self):
        return {
            "__class__": "Card",
            'rank': self.rank,
            'suit': self.suit}
```

We've defined a small base class for playing cards. Each instance of the class has two attributes, `rank` and `suit`. We've omitted the hash and comparison method definitions. To follow the *Creating a class that has orderable objects* recipe from `Chapter 7`, *More Advanced Class Design*, this class would need a number of additional special methods. This recipe will avoid those complications.

We've defined a `to_json()` method that is handy for serializing this complex object into a consistent JSON format. This method emits a dictionary representation of the state of the `Card`. If we want to deserialize `Card` objects from JSON notation, we'll need to also create an `object_hook` function. We don't need it for this recipe, though, since we won't accept `Card` objects as input.

We'll also need a `Deck` class as a container of `Card` instances. An instance of this class can create the `Card` instances as well as acting as a stateful object that can deal cards. Here's the class definition:

```
import random
 class Deck:
    SUITS = (
        '\N{black spade suit}',
        '\N{white heart suit}',
        '\N{white diamond suit}',
        '\N{black club suit}',
    )

    def __init__(self, n=1):
        self.n = n
        self.create_deck(self.n)

    def create_deck(self, n=1):
        self.cards = [
            Card(r,s)
                for r in range(1,14)
                    for s in self.SUITS
                        for _ in range(n)
        ]
        random.shuffle(self.cards)
        self.offset = 0

    def deal(self, hand_size=5):
        if self.offset + hand_size > len(self.cards):
            self.create_deck(self.n)
        hand = self.cards[self.offset:self.offset+hand_size]
        self.offset += hand_size
        return hand
```

The `create_deck()` method uses a generator to create all 52 combinations of the thirteen ranks and four suits. Each suit is defined by a single character: ♣, ♢, ♡, or ♠. The example spells out the Unicode character names using `\N{}` sequences.

If a value of `n` is provided when creating the `Deck` instance, the container will create multiple copies of the 52-card deck. This multideck shoe is sometimes used to speed up play by reducing the time spent shuffling. Once the sequence of `Card` instances has been created, it is shuffled using the `random` module. For repeatable test cases, a fixed seed can be provided.

The `deal()` method will use the value of `self.offset` to determine where to start dealing. This value starts at `0` and is incremented after each hand of cards is dealt. The `hand_size` argument determines how many cards will be in the next hand. This method updates the state of the object by incrementing the value of `self.offset` so that the cards are dealt just once.

Here's one way to use this class to create `Card` objects:

```
>>> from ch12_r01 import deck_factory
>>> import random
>>> import json

>>> random.seed(2)
>>> deck = Deck()
>>> cards = deck.deal(5)
>>> cards
[Card(rank=4, suit='♠'), Card(rank=8, suit='♡'),
 Card(rank=3, suit='♡'), Card(rank=6, suit='♡'),
 Card(rank=2, suit='♣')]
```

To create a sensible test, we provided a fixed seed value. The script created a single deck using `Deck()`. We can then deal a hand of five `Card` instances from the deck.

In order to use this as part of a web service, we'll also need to produce useful output in JSON notation. Here's an example of how that would look:

```
>>> json_cards = list(card.to_json() for card in deck.deal(5))
>>> print(json.dumps(json_cards, indent=2, sort_keys=True))
    [
      {
        "__class__": "Card",
        "rank": 2,
        "suit": "\u2662"
      },
      {
```

```
    "__class__": "Card",
    "rank": 13,
    "suit": "\u2663"
  },
  {
    "__class__": "Card",
    "rank": 7,
    "suit": "\u2662"
  },
  {
    "__class__": "Card",
    "rank": 6,
    "suit": "\u2662"
  },
  {
    "__class__": "Card",
    "rank": 7,
    "suit": "\u2660"
  }
]
```

We've used `deck.deal(5)` to deal a hand with five more cards from the deck. The expression `list(card.to_json() for card in deck.deal(5))` will use the `to_json()` method of each `Card` object to emit the small dictionary representation of that object. The list of dictionary structure was then serialized into JSON notation. The `sort_keys=True` option is handy for creating a repeatable test case. It's not generally necessary for RESTful web services.

How to do it...

1. Import needed modules and objects. We'll use the `HTTPStatus` class because it defines the commonly-used HTTP status codes. The `json` module is required to produce JSON responses. We'll also use the `os` module to initialize a random number seed:

```
from http import HTTPStatus
import json
import os
import random
```

2. Import or define the underlying classes, `Card` and `Deck`. Generally, it's a good idea to define these as a separate module. The basic features should exist and be tested outside the web services environment. The idea is that web services should wrap existing, working software.
3. Create objects that are shared by all sessions. The value of `deck` is a module global variable:

```
random.seed(os.environ.get('DEAL_APP_SEED'))
deck = Deck()
```

We've relied on the `os` module to examine the environment variables. If the environment variable `DEAL_APP_SEED` is defined, we'll seed the random number generator with the string value. Otherwise, we'll rely on the built-in randomization features of the `random` module.

4. Define the target WSGI application as a function. This function will respond to a request by dealing a hand of cards and then creating a JSON representation of the `Card` information:

```
def deal_cards(environ, start_response):
    global deck
    hand_size = int(environ.get('HAND_SIZE', 5))
    cards = deck.deal(hand_size)
    status = "{status.value} {status.phrase}".format(
     status=HTTPStatus.OK)
    headers = [('Content-Type', 'application/json;charset=utf-8')]
    start_response(status, headers)
    json_cards = list(card.to_json() for card in cards)
    return [json.dumps(json_cards, indent=2).encode('utf-8')]
```

The `deal_cards()` function deals the next group of cards from the `deck`. The OS environment can define a `HAND_SIZE` environment variable to change the size of the deal. The global `deck` object is used to perform the relevant processing.

The status line for a response is a string that has the numeric value and phrase for the HTTP status of `OK`. This can be followed by headers. This example includes the `Content-Type` header to provide information to the client; the content is a JSON document and that the bytes for this document are encoded using `utf-8`. Finally, the document itself is the return value from this function.

5. For demonstration and debugging purposes, it's helpful to build a server that runs the WSGI application. We'll use the `wsgiref` module's server. There are good servers defined in Werkzeug. Servers such as GUnicorn are even better:

```
from wsgiref.simple_server import make_server
httpd = make_server('', 8080, deal_cards)
httpd.serve_forever()
```

   Once the server is running, we can open a browser to see `http://localhost:8080/`. This will return a batch of five cards. Each time we refresh, we get a different batch of cards.

This works because entering a URL in the browser executes a `GET` request with a minimal set of headers. Since our WSGI application didn't require any specific headers, and responded to any HTTP method, it will return a result.

The result is a JSON document that represents five cards dealt from the current deck. Each card is represented with a class name, `rank`, and `suit`:

```
[
  {
    "__class__": "Card",
    "suit": "\u2663",
    "rank": 6
  },
  {
    "__class__": "Card",
    "suit": "\u2662",
    "rank": 8
  },
  {
    "__class__": "Card",
    "suit": "\u2660",
    "rank": 8
  },
  {
    "__class__": "Card",
    "suit": "\u2660",
    "rank": 10
  },
  {
    "__class__": "Card",
    "suit": "\u2663",
    "rank": 11
  }
]
```

We can create web pages with clever JavaScript programs to fetch batches of cards. These web pages and JavaScript programs can animate dealing, and include graphics for the card images.

How it works...

The WSGI standard defines an interface between a web server and an application. This is based on the the Apache HTTPD **Common Gateway Interface** (**CGI**). The CGI was designed to run shell scripts or separate binaries. The WSGI is an enhancement to this legacy concept.

The WSGI standard defines the environment dictionary with a variety of information:

- A number of keys in the dictionary reflect the request after some preliminary parsing and data conversion.
    - `REQUEST_METHOD`: The HTTP request method, such as `GET` or `POST`.
    - `SCRIPT_NAME`: The initial portion of the request URL's path. This is generally taken as an overall application object or function.
    - `PATH_INFO`: The remainder of the request URL's path, designating a location of a resource. In this example, no path parsing is performed.
    - `QUERY_STRING`: The portion of the request URL that follows the `?`, if any:
    - `CONTENT_TYPE`: The contents of any Content-Type header value in the HTTP request.
    - `CONTENT_LENGTH`: The contents of any Content-Length header value in the HTTP request.
    - `SERVER_NAME` and `SERVER_PORT`: The server name and port number from the request.
    - `SERVER_PROTOCOL`: The version of the protocol the client used to send the request. Typically, this will be something like `HTTP/1.0` or `HTTP/1.1`.
- **The HTTP headers**: These will have keys that start with `HTTP_` and contain the header name in all uppercase letters.

Generally the contents of a request are not the only data that's required to create a meaningful response from a server. Often, additional information is required. This information generally includes two other kinds of data:

- **OS environment**: The environment variables that were in place when the service was started provide configuration details for the server. This could provide a path to a directory that contains static content. It could provide information used for authenticating users.
- **WSGI server context**: These keys start with `wsgi.` and are always lowercase. The values include some additional information on the internal state of a server that adheres to the WSGI standard. There are two particularly interesting objects that upload files and logging support:
    - `wsgi.input`: It is a file-like object. From this, the HTTP request body bytes can be read. This will often have to be decoded based on the `Content-Type` header.
    - `wsgi.errors`: It is a file-like object to which error output can be written. This is the server's log.

The return value from a WSGI function can be a sequence object or an iterable. Returning an iterable is the way a very large document can be built in pieces and downloaded via a number of smaller buffers.

This example WSGI application does not check the request path. Any path can be used to retrieve a hand of cards. A more sophisticated application might parse the path to determine information about the size of a hand being requested or the size of the deck from which the hand should be dealt.

There's more...

A web service can be visualized as a number of common pieces that are connected together into nested shells or layers. The uniform interface for WSGI applications encourages this kind of composition of reusable features.

There are a number of common techniques that are used to protect and produce dynamic content. These techniques are cross-cutting concerns for web service applications. We have a few choices for this as follows:

- We can write lots of `if` statements in a single application
- We can extract the common programming and create a common wrapper that separates security concerns from the construction of content

A wrapper is simply another WSGI application that doesn't produce a result directly. Instead, a wrapper hands off the work of producing results to another WSGI application.

We might, for example, need a wrapper that confirms that a JSON response is expected. This wrapper will distinguish requests for human-centric HTML from application-focused JSON.

To make more flexible applications, it's often helpful to use a callable object instead of a simple function. Doing this makes configuration of the various applications and wrappers considerably more flexible. We'll combine the idea of a JSON filter with a callable object.

The outline of this object looks like the following:

```
class JSON_Filter:
    def __init__(self, json_app):
        self.json_app = json_app
    def __call__(self, environ, start_response):
        return json_app(environ, start_response)
```

We'll create a callable object from this class definition by providing another app. The other app, `json_app`, will be wrapped by this callable object.

We'll use it like this:

```
json_wrapper = JSON_Filter(deal_cards)
```

This will wrap the original `deal_cards()` WSGI application. We can now use the composite `json_wrapper` object as a WSGI application. When the server calls `json_wrapper(environ, start_response)`, that will invoke the `__call__()` method of the object, which—in this example, will pass the request to the `deal_cards()` function.

Here's the more complete wrapper application. This wrapper will check the HTTP Accept header for the characters `"json"`. It will also check the query string for `?$format=json` to see if a JSON-formatted request was made. An instance of this class can be configured to reference the `deal_cards()` WSGI application:

```
from urllib.parse import parse_qs
class JSON_Filter:
    def __init__(self, json_app):
        self.json_app = json_app
    def __call__(self, environ, start_response):
        if 'HTTP_ACCEPT' in environ:
            if 'json' in environ['HTTP_ACCEPT']:
                environ['$format'] = 'json'
                return self.json_app(environ, start_response)
        decoded_query = parse_qs(environ['QUERY_STRING'])
```

```
            if '$format' in decoded_query:
                if decoded_query['$format'][0].lower() == 'json':
                    environ['$format'] = 'json'
                    return self.json_app(environ, start_response)
            status = "{status.value}
{status.phrase}".format(status=HTTPStatus.BAD_REQUEST)
            headers = [('Content-Type', 'text/plain;charset=utf-8')]
            start_response(status, headers)
            return ["Request doesn't include ?$format=json or Accept
header".encode('utf-8')]
```

The `__call__()` method checks the Accept header as well as the query string. If the string `json` appears anywhere in the HTTP Accept header, then the given application is invoked. The environment is updated to include header information used by this wrapper.

If the HTTP Accept header is not present or doesn't require a JSON response, then the query string is checked. This fall-back can be helpful because it is difficult to change the headers sent by a browser; using the query string is a browser-friendly alternative to the Accept header. The `parse_qs()` function will decompose the query string into a dictionary of keys and values. If the query string has `$format` as a key, then this is checked to see if the value includes `'json'`. If this is true, then the environment is updated with the format information found in the query string.

In both cases, the environment is modified when calling the wrapped application. The function being wrapped only needs to check the WSGI environment for format information. This wrapper object returns the response without any further modification.

If the request does not request JSON, then a `400 BAD REQUEST` response is sent with a simple text message. This will provide some guidance as to why the query was unacceptable.

We use this `JSON_Filter` wrapper class definition as follows:

```
json_wrapper = JSON_Filter(deal_cards)
httpd = make_server('', 8080, json_wrapper)
```

Instead of making a server from `deal_cards()`, we've created an instance of the `JSON_Filter` class that references the `deal_cards()` function. This will behave almost exactly like the version shown earlier. The important difference is that this requires either an Accept header or a URL like this: `http://localhost:8080/?$format=json`.

This example has a subtle semantic issue. The `GET` method changes the state of the server. This is generally a bad idea.

Because we're looking at a browser, it's difficult to sort out problems. There isn't much debugging support available here. This means that `print()` functions as well as log messages are essential for debugging. Because of the way WSGI works, it's essential to print to `sys.stderr`. It is easier to work with Flask, which we'll show in the *Using the Flask framework for RESTful APIs* recipe.

HTTP supports a number of methods, including `GET`, `POST`, `PUT`, and `DELETE`. Generally, it's sensible to map these methods to database **CRUD** operations; Create is done with `POST`, Retrieve is done with `GET`, Update is done with `PUT`, and Delete maps to `DELETE`. This means that a `GET` operation will not change the state of the database.

This leads to the idea that a web service's `GET` operation should be idempotent. A series of `GET` operations without any other `POST`, `PUT`, or `DELETE` operation should return the same result each time. In this recipe, each `GET` returns a different result. This is a semantic problem with using `GET` to deal cards.

For our purpose of demonstrating the basics, the distinction is minor. In a large and more complex web application, the distinction is an important consideration. Since the deal service is not idempotent, there's a point of view that suggests it should be accessed with the `POST` method.

To make it easy to explore using a browser, we've avoided checking the method in the WSGI application.

See also

- Python has a wide variety of RESTful API frameworks. In the *Using the Flask framework for RESTful APIs* recipe, we'll look at the Flask framework.
- There are three places to look for detailed information on the overall WSGI standard:
    - **PEP 3333**: See `https://www.python.org/dev/peps/pep-3333/`.
    - **The Python standard library**: It includes the `wsgiref` module. This is the reference implementation in the standard library.
    - **The Werkzeug project**: See `http://werkzeug.pocoo.org`. This is an external library with numerous WSGI utilities. This is used widely to implement proper WSGI applications.

- Also, see `http://docs.oasis-open.org/odata/odata-json-format/v4.0/odata-json-format-v4.0.html` for more information on JSON-formatting of data for web services.

Using the Flask framework for RESTful APIs

In the *Implementing web services with WSGI* recipe, we looked at building RESTful APIs and microservices using the WSGI components available in the Python standard library. This leads to a large amount of programming to handle a number of common cases.

How can we simplify all of the common web application programming and eliminate boilerplate code?

Getting ready

First, we'll need to add the Flask framework to our environment. This generally relies on using `pip` to install the latest release of Flask and the other related projects, `itsdangerous`, `Jinja2`, `click`, `MarkupSafe`, and `Werkzeug`.

The installation looks like the following:

```
slott$ sudo pip3.5 install flask
Password:
Collecting flask
  Downloading Flask-0.11.1-py2.py3-none-any.whl (80kB)
    100% |████████████████████████████████| 81kB
3.6MB/s
Collecting itsdangerous>=0.21 (from flask)
  Downloading itsdangerous-0.24.tar.gz (46kB)
    100% |████████████████████████████████| 51kB
8.6MB/s
Requirement already satisfied (use --upgrade to upgrade): Jinja2>=2.4 in
/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-
packages (from flask)
Collecting click>=2.0 (from flask)
  Downloading click-6.6.tar.gz (283kB)
    100% |████████████████████████████████| 286kB
4.0MB/s
Collecting Werkzeug>=0.7 (from flask)
  Downloading Werkzeug-0.11.10-py2.py3-none-any.whl (306kB)
    100% |████████████████████████████████| 307kB
3.8MB/s
Requirement already satisfied (use --upgrade to upgrade): MarkupSafe in
```

```
/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-
packages (from Jinja2>=2.4->flask)
Installing collected packages: itsdangerous, click, Werkzeug, flask
  Running setup.py install for itsdangerous ... done
  Running setup.py install for click ... done
Successfully installed Werkzeug-0.11.10 click-6.6 flask-0.11.1
itsdangerous-0.24
```

We can see that `Jinja2` and `MarkupSafe` were already installed. The missing elements were located by `pip`, downloaded, and installed. Windows users won't use the `sudo` command.

Flask allows us to dramatically simplify our web services application. Instead of creating a large and possibly complex WSGI-compatible function or callable object, we can create a module with separate functions. Each function can handle a specific pattern of URL paths.

We'll look at the same core card-dealing functions we had in the *Implementing web services with WSGI* recipe. The `Card` class defines a simple playing card. The `Deck` class defines a deck of cards.

Because Flask handles the details of URL parsing for us, we can create a much more sophisticated web service quite easily. We'll define a path that looks like this:

`/dealer/hand/?cards=5`.

This route has three important pieces of information:

- The first part of the path, `/dealer/`, is the overall web service.
- The next part of the path, `hand/`, is a specific resource, a hand of cards.
- The query string, `?cards=5`, defines the cards parameter for the query. This is the size of the hand being requested. This is limited to a range of 1 to 52 cards. A value that's out of range will get a `400` status code because the query is invalid.

How to do it...

1. Import some core definitions from the `flask` package. The `Flask` class defines the overall application. The `request` object holds the current web request:

```
from flask import Flask, request, jsonify, abort
from http import HTTPStatus
```

The `jsonify()` function will return a JSON-format object from a Flask view function. The `abort()` function returns an HTTP error status and ends processing of the request.

2. Import the underlying classes, `Card` and `Deck`. Ideally, these are imported from a separate module. It should be possible to test all of the features outside the web services environment:

```
from ch12_r01 import Card, Deck
```

   In order to properly shuffle, we'll also need the `random` module:

```
import random
```

3. Create the `Flask` object. This is the overall web services application. We'll call the Flask application `'dealer'`, and we'll also assign the object to a global variable, `dealer`:

```
dealer = Flask('dealer')
```

4. Create any objects used throughout the application. These can be assigned to the `Flask` object, `dealer`, as attributes. Be sure to create a unique name that doesn't conflict with any of Flask's internal attributes. The alternative is to use module globals.

   Stateful global objects must be able to work in a multi-threaded environment, or threading must be explicitly disabled:

```
import os
random.seed(os.environ.get('DEAL_APP_SEED'))
deck = Deck()
```

   For this recipe, the implementation of the `Deck` class is not thread-safe, so we'll rely on having a single-threaded server. The `deal()` method should use the `Lock` class from the `threading` module to define an exclusive lock to assure proper operation with concurrent threads.

5. Define a route—a URL pattern—to a view function that performs a specific request. This is a decorator, placed immediately in front of the function. It will bind the function to the Flask application:

```
@dealer.route('/dealer/hand/')
```

6. Define the view function, which retrieves data or updates the application state. In this example, the function does both:

```
def deal():
    try:
        hand_size = int(request.args.get('cards', 5))
        assert 1 <= hand_size < 53
    except Exception as ex:
        abort(HTTPStatus.BAD_REQUEST)
    cards = deck.deal(hand_size)
    response = jsonify([card.to_json() for card in cards])
    return response
```

   Flask parses the string after the `?` in the URL—the query string—to create the `request.args` value. A client application or browser can set this value with a query string such as `?cards=13`. This will deal 13-card hands for bridge.

   If the hand size value from the query string is inappropriate, the `abort()` function will end processing and return an HTTP status code of `400`. This indicates that the request was unacceptable. This is a minimal response, with no more detailed content.

   The real work of the application is a single statement, `cards = dealer.deck.deal(hand_size)`. The idea here is to wrap existing functionality in a web framework. The features can be tested without the web application.

   The response is handled by the `jsonify()` function: this creates a response object. The body of the response which will be a Python object represented in JSON notation. If we need to add headers to the response, we can update `response.headers` to include additional information.

7. Define the main program which runs the server:

```
if __name__ == "__main__":
    dealer.run(use_reloader=True, threaded=False, debug=True)
```

   We've included the `debug=True` option to provide rich debugging information in the browser as well as the Flask log file. Once the server is running, we can open a browser to see `http://localhost:5000/`. This will return a batch of five cards. Each time we refresh, we get a different batch of cards.

This works because entering a URL in the browser executes a `GET` request with a minimal set of headers. Since our WSGI application didn't require any specific headers, and responded to all of the HTTP methods, it will return a result.

The result is a JSON document with five cards. Each card is represented by a class name, `rank`, and `suit` information:

```
[
  {
    "__class__": "Card",
    "suit": "\u2663",
    "rank": 6
  },
  {
    "__class__": "Card",
    "suit": "\u2662",
    "rank": 8
  },
  {
    "__class__": "Card",
    "suit": "\u2660",
    "rank": 8
  },
  {
    "__class__": "Card",
    "suit": "\u2660",
    "rank": 10
  },
  {
    "__class__": "Card",
    "suit": "\u2663",
    "rank": 11
  }
]
```

To see more than five cards, the URL can be modified. For example, this will return a bridge hand: `http://127.0.0.1:5000/dealer/hand/?cards=13`.

How it works...

A Flask application consists of an application object with a number of individual view functions. In this recipe, we created a single view function, `deal()`. Applications often have numerous functions. A complex website may have many applications, each of which has many functions.

A route is a mapping between a URL pattern and a view function. This makes it possible to have routes which contain parameters that are used by the view function.

The `@flask.route` decorator is the technique used to add each route and view function into the overall Flask instance. The view function is bound into the overall application based on the route pattern.

The `run()` method of a `Flask` object does the following kinds of processing. This isn't precisely how Flask works, but it provides a broad outline of the various steps:

- It waits for an HTTP request. Flask follow the WSGI standard, the request arrives in the form of a dictionary. For more information on WSGI, see the *Implementing web services with WSGI* recipe.
- It creates a Flask `Request` object from the WSGI environment. The `request` object has all of the information from the request, including all of the URL elements, query string elements, and any attached documents.
- Flask then examines the various routes, looking for a route which matches the request's path.
    - If a route is found, then the view function is executed. The function creates a `Response` object. This is the return value from a view function.
    - If a route is not found, a `404 NOT FOUND` response is sent automatically.
- The WSGI pattern is followed to prepare status and headers to start sending the response. The `Response` object that was returned from the view function is then provided as a stream of bytes.

A Flask application can contain a number of methods that make it very easy to provide a web service. Flask exposes some of these methods as standalone functions that are implicitly bound to the request or the session. This makes it slightly simpler to write view functions.

There's more...

In the *Implementing web services with WSGI* recipe, we wrapped the application in a generic test that confirmed that the request had one of two properties. We used the following two rules:

- An Accept header that required JSON
- A query string with `$format=json` in it

If we're writing a complex RESTful application server, we often want this kind of test applied to all of the view functions. We'd rather not repeat the code for this test.

We can—of course—combine the WSGI solution from the *Implementing web services with WSGI* recipe with the Flask application to build a composite application. We can also accomplish this entirely within Flask. The pure Flask solution is a bit simpler than the WSGI solution, making it desirable.

We've seen the Flask `@flask.route` decorator. Flask has a number of other decorators that can be used to define various stages in request and response processing. In order to apply a test to the incoming request, we can use the `@flask.before_request` decorator. All of the functions with this decoration will be invoked prior to the request being processed:

```
@dealer.before_request
def check_json():
    if 'json' in request.headers.get('Accept'):
    return
    if 'json' == request.args.get('$format'):
        return
    return abort(HTTPStatus.BAD_REQUEST)
```

When a `@flask.before_request` decorator fails to return a value (or returns `None`), then processing will continue. The routes will be checked, and a view function will be evaluated.

In this example, if the Accept header includes `json` or the `$format` query parameter is `json`, then the function returns `None`. This means that the normal view function will then be found to process the request.

When a `@flask.before_request` decorator returns a value, this is the final result, and processing stops. In this example, the `check_json()` function may return an `abort()` response, which will stop processing. The `abort()` response becomes the final response from the Flask application. This makes it very easy to return error messages.

We can now use a browser's address window to enter a URL like the following:

```
http://127.0.0.1:5000/dealer/hand/?cards=13&$format=json
```

This will return a 13-card hand, and the request now explicitly requests the result in JSON format. It is instructive to try other values for `$format` as well as omitting the `$format` key entirely.

This example has a subtle semantic issue. The `GET` method changes the state of the server. This is generally a bad idea.

HTTP supports a number of methods that parallel database CRUD operations. Create is done with `POST`, Retrieve is done with `GET`, Update is done with `PUT`, and Delete maps to `DELETE`.

This idea then leads to the idea that a web services `GET` operation should be idempotent. A series of `GET` operations—without any other `POST`, `PUT`, or `DELETE`—should return the same result each time. In this example, each `GET` returns a different result. Since the deal service is not idempotent, it should be accessed with the `POST` method.

To make it easy to explore using a browser, we've avoided checking the method in the Flask route. Ideally, the route decorator should look like the following:

```
@dealer.route('/dealer/hand/', methods=['POST'])
```

Doing this makes it difficult to use a browser to see that the service is working. In the *Making REST requests with urllib* recipe we'll look at creating a client, and switching to using `POST` for the method.

See also

- For background in web services, see the *Implementing web services with WSGI* recipe.
- See `http://flask.pocoo.org/docs/0.11/` for details of Flask.
- See `https://www.packtpub.com/web-development/learning-flask-framework` to learn more about the Flask framework. Also, `https://www.packtpub.com/web-development/mastering-flask` has more information on mastering Flask.

Parsing the query string in a request

A URL is a complex object. It contains at least six separate pieces of information. More information can be included via optional elements.

A URL such as `http://127.0.0.1:5000/dealer/hand/?cards=13&$format=json` has several fields:

- `http` is the scheme. `https` is for secure connections using encrypted sockets.
- `127.0.0.1` can be called the authority, although network location is more commonly used. This particular IP address means the localhost and is a kind of loopback to the localhost. The name localhost maps to this IP address.
- `5000` is the port number, and is part of the authority.
- `/dealer/hand/` is the path to a resource.
- `cards=13&$format=json` is a query string, and it's separated from the path by the `?` character.

The query string can be quite complex. While not an official standard, it's possible (and common) for a query string to have a repeated key. The following query string is valid, though perhaps confusing:

```
?cards=13&cards=5
```

We've repeated the `cards` key. The web service will provide a thirteen-card hand and a five-card hand.

[*The author is unaware of any card games with hands of varying sizes. The lack of a good user story makes this example somewhat contrived.*]

The ability to repeat a key breaks the possibility of a simple mapping between a URL query string and a built-in Python dictionary. There are several possible solutions to this problem:

- Each key in the dictionary must be associated with a `list` that contains all of the values. This is awkward for the most common case where a key is not repeated; each list has only a single item. This solution is implemented via the `parse_qs()` in `urllib.parse`.
- Each key is only saved once and the first (or last) value is kept, the other values are dropped. This is awful.
- A dictionary not used. Instead the query string can be parsed into a list of *(key, value)* pairs. This also allows keys to be duplicated. For the common case with unique keys, the list can be converted to a dictionary. For the uncommon case, the duplicated keys can be handled some other way. This is implemented by the `parse_qsl()` in `urllib.parse`.

Is there a better way to handle a query string? Can we have a more sophisticated structure that behaves like a dictionary with single values for the common case, and a more complex object for the rare cases where a field key is duplicated and has multiple values?

Getting ready

Flask depends on another project, `Werkzeug`. When we install Flask using `pip`, the requirements will lead `pip` to also install the Werkzeug toolkit. Werkzeug has a data structure that provides an excellent way to handle query strings.

We'll modify the example in the *Using the Flask framework for RESTful APIs* recipe to use a somewhat more complex query string. We'll add a second route that deals multiple hands. The sizes of each hand will be specified in a query string that allows repeated keys.

How to do it...

1. Start with the *Using the Flask framework for RESTful APIs* recipe. We'll be adding a new view function to an existing web application.
2. Define a route—a URL pattern—to a view function that performs a specific request. This is a decorator, placed immediately in front of the function. It will bind the function to the Flask application:

```
@dealer.route('/dealer/hands/')
```

3. Define a view function that responds to requests sent to the particular route:

```
def multi_hand():
```

4. Within the view function, extract the values of a unique key with the `get()` method or use ordinary `[]` syntax that is appropriate for the built-in dict type. This returns individual values without the complication of a list for the common case where the list would only have a single element.
5. For repeated keys, use the `getlist()` method. This returns each of the values as a list. Here's a view function that looks for a query string such as `?card=5&card=5` to deal two five-card hands:

```
try:
    hand_sizes = request.args.getlist('cards', type=int)
    if len(hand_sizes) == 0:
        hand_sizes = [13,13,13,13]
    assert all(1 <= hand_size < 53 for hand_size in hand_sizes)
```

```
    except Exception as ex:
        dealer.logger.exception(ex)
        abort(HTTPStatus.BAD_REQUEST)

    hands = [deck.deal(hand_size) for hand_size in hand_sizes]
    response = jsonify(
        [
            {'hand':i,
             'cards':[card.to_json() for card in hand]
            } for i, hand in enumerate(hands)
        ]
    )
    return response
```

This function will get all of the `cards` keys from the query string. If the values are all integers, and each value is in the range 1 to 52 (inclusive), then the values are valid, and the view function will return a result. If there are no `cards` key values in the query, then four hands of 13 cards will be dealt.

The response will be a JSON representation of each hand as a small dictionary with two keys: a hand ID, and the cards from the hand.

6. Define a main program that runs the server:

```
if __name__ == "__main__":
    dealer.run(use_reloader=True, threaded=False)
```

Once the server is running, we can open a browser to see this URL:

`http://localhost:5000/?cards=5&cards=5&$format=json`

The result is a JSON document with two hands of five cards. We've elided some details to emphasize the structure of the response:

```
[
  {
    "cards": [
      {
        "__class__": "Card",
        "rank": 11,
        "suit": "\u2660"
      },
      {
        "__class__": "Card",
        "rank": 8,
        "suit": "\u2662"
      },
```

```
            ...
          ],
          "hand": 0
        },
        {
          "cards": [
            {
              "__class__": "Card",
              "rank": 3,
              "suit": "\u2663"
            },
            {
              "__class__": "Card",
              "rank": 9,
              "suit": "\u2660"
            },
            ...
          ],
          "hand": 1
        }
      ]
```

Because the web service parses the query string, it's trivial to add more complex hand sizes to the query string. The example includes the `$format=json` based on the *Using the Flask framework for RESTful APIs* recipe.

If the `@dealer.before_request` function, `check_json()`, is implemented to check for JSON, then the `$format` is required. If the `@dealer.before_request` function, `check_json()`, is not implemented, then the additional information in the query string is ignored.

How it works...

The Werkzeug—`Multidict` class is a very handy data structure. This is an extension to the built-in dictionary. It allows multiple, distinct values for a given key.

We can build something like this using the `defaultdict` class from the `collections` module. The definition would be `defaultdict(list)`. The problem with this definition is that the value of every key is a list, even when the list only has a single item as a value.

The advantage provided by the `Multidict` class are the variations on the `get()` method. The `get()` method returns the first value when there are many copies of a key or the only value when the key occurs only once. This has a default parameter, as well. This method parallels the method of the built-in `dict` class.

The `getlist()` method, however, returns a list of all values for a given key. This method is unique to the `Multidict` class. We can use this method to parse more complex query strings.

One common technique that's used to validate query strings is to pop items as they are validated. This is done with the `pop()` and `poplist()` methods. These will remove the key from the `Multidict` class. If any keys remain after checking all the valid keys, these extras can be considered syntax errors, and the web request rejected with `abort(HTTPStatus.BAD_REQUEST)`.

There's more...

The query string uses relatively simple syntax rules. There are one or more key-value pairs using = as the punctuation between key and value. The separator between each pair is the `&` character. Because of the meaning of other characters in parsing a URL, there is one other rule that's important—the keys and values must be encoded.

The URL encoding rules require that certain characters be replaced with HTML entities. The technique is called percent encoding. This means that when we put `&` into the value of a query string, it must be encoded as `%26`, here's an example showing this encoding:

```
>>> from urllib.parse import urlencode
>>> urlencode( {'n':355,'d':113} )
'n=355&d=113'
>>> urlencode( {'n':355,'d':113,'note':'this&that'} )
'n=355&d=113&note=this%26that'
```

The value `this&that` was encoded to `this%26that`.

There's a short list of characters which must have the `%`-encoding rules applied. This comes from the *RFC 3986*, refer to *section 2.2*, *Reserved Characters*. The list includes these characters:

```
! * ' ( ) ; : @ & = + $ , / ? # [ ] %
```

Generally, the JavaScript code associated with a web page will handle encoding query strings. If we're writing an API client in Python, we need to use the `urlencode()` function to properly encode query strings. Flask handles the decoding automatically for us.

There's a practical size limit on the query string. Apache HTTPD, for example, has a `LimitRequestLine` configuration parameter with a default value of `8190`. This limits the overall URL to this size.

In the OData specifications (`http://docs.oasis-open.org/odata/odata/v4.0/`), there are several kinds of value that are suggested for the query options. This specification suggests that our web services should support the following kinds of query option:

- For a URL that identifies an entity or a collection of entities, the `$expand` and `$select` options can be used. Expanding a result means that the query will provide additional details. The select query will impose additional criteria on the collection.
- A URL that identifies a collection should support `$filter`, `$search`, `$orderby`, `$count`, `$skip`, and `$top` options. These don't make sense for a URL that returns a single item. The `$filter` and `$search` options accept complex conditions for finding data. The `$orderby` option defines a particular order to impose on the results.

The `$count` option changes the query fundamentally. It will return the count of items instead of the items themselves.

The `$top` and `$skip` options are used to page through data. If the count is large, it's common to use the `$top` option to limit the results to a specific number that will be shown on a web page. The value of the `$skip` option determines which page of data will be shown. For example, `$top=20$skip=40` would be page 3 of the results—the top twenty after skipping 40.

Generally, all URLs should support the `$format` option to specify the format of the result. We've been focusing on JSON, but a more sophisticated service might offer CSV output or even XML.

See also

- See the *Using the Flask framework for RESTful APIs* recipe for the basics of using Flask for web services.
- In the *Making REST requests with urllib* recipe, we'll look at how to write a client application that can prepare complex query strings.

Making REST requests with urllib

A web application has two essential parts:

- **A client**: This can be a user's browser, but may also be a mobile device app. In some cases, a web server may be a client of other web servers.
- **A server**: This provides the web services and resources we've been looking at, in the *Implementing web services with WSGI*, *Using the Flask framework for RESTful APIs*, and *Parsing the query string in a request* recipes, as well as other recipes, such as *Parsing a JSON request* and *Implementing authentication for web services*.

A browser-based client will generally be written in JavaScript. Mobile apps are written in a variety of languages, with a focus on Java for Android devices and Objective-C with Swift for iOS devices.

There are several user stories that involve RESTful API clients written in Python. How can we create a Python program that is a client of RESTful web services?

Getting ready

We'll assume that we have a web server based on the *Implementing web services with WSGI*, *Using the Flask framework for RESTful APIs*, or *Parsing the query string in a request* recipe. We can write a formal specification for this server's behavior in the following way:

```
{
  "swagger": "2.0",
  "info": {
    "title": "dealer",
    "version": "1.0"
  },
  "schemes": ["http"],
  "host": "127.0.0.1:5000",
  "basePath": "/dealer",
  "consumes": ["application/json"],
```

```
      "produces": ["application/json"],
      "paths": {
        "/hands": {
          "get": {
            "parameters": [
              {
                "name": "cards",
                "in": "query",
                "description": "number of cards in each hand",
                "type": "array",
                "items": {"type": "integer"},
                "collectionFormat": "multi",
                "default": [13, 13, 13, 13]
              }
            ],
            "responses": {
              "200": {
                "description":
                "one hand of cards for each `hand` value in the query
string"
              }
            }
          }
        },
        "/hand": {
          "get": {
            "parameters": [
              {
                "name": "cards",
                "in": "query",
                "type": "integer",
                "default": 5
              }
            ],
            "responses": {
              "200": {
                "description":
                "One hand of cards with a size given by the `hand` value in
the query string"
              }
            }
          }
        }
      }
    }
```

This document provides us some guidance on how to consume these services using Python's `urllib` module. It also describes what the expected responses should be, giving us guidance on how to handle the responses.

Some of the fields in this specification define a base URL. These three fields, in particular, provide this information:

```
"schemes": ["http"],
"host": "127.0.0.1:5000",
"basePath": "/dealer",
```

The `produces` and `consumes` fields provide information that helps to build and verify the HTTP headers. The request `Content-Type` header must be a **Multipurpose Internet Mail Extensions** (**MIME**) type that the server consumes. Similarly, the request Accept header must specify a MIME type that the server produces. In both cases, we'll supply `application/json`.

The detailed service definitions are provided in the `paths` section of the specification. The `/hands` path, for example, shows the details of how to make a request for multiple hands. The path detail is a suffix for the `basePath` value.

When the HTTP method is `get`, then parameters are provided in the query. The `cards` parameter in the query provides an integer number of cards, and it can be repeated multiple times.

The response will include at least the response described. In this case, the HTTP status will be `200`, and the body of the response has a minimal description. It's possible to provide a more formal schema definition for the response, we'll omit that from this example.

How to do it...

1. Import the `urllib` components that are required. We'll be making URL requests, and building more complex objects, such as query strings. We'll need the `urllib.request` and `urllib.parse` modules for these two features. Since the expected response is in JSON, then the `json` module will be useful as well:

```
import urllib.request
import urllib.parse
import json
```

2. Define the query string that will be used. In this case, all of the values happen to be fixed. In a more complex application, some might be fixed and some might be based on user inputs:

```
query = {'hand': 5}
```

3. Use the query to build the pieces of the full URL:

```
full_url = urllib.parse.ParseResult(
    scheme="http",
    netloc="127.0.0.1:5000",
    path="/dealer" + "/hand/",
    params=None,
    query=urllib.parse.urlencode(query),
    fragment=None
)
```

   In this case, we're using a `ParseResult` object to hold the relevant parts of the URL. This class isn't graceful about missing items, so we must provide explicit `None` values for parts of the URL that aren't being used.

   We could use `"http://127.0.0.1:5000/dealer/hand/?cards=5"` in our script. However, this condensed string is awkward to change. It's useful as a compact message when making the request, but it's not ideal for making flexible, maintainable, and testable programs.

   Using this long constructor has the advantage of providing explicit values for each part of a URL. In more complex applications, the individual pieces are built from an analysis of the JSON Swagger specification document shown previously:

4. Build a final `Request` instance. We'll use the URL built from a variety of pieces. We'll explicitly provide an HTTP method (browsers tend to use `GET` as a default). Also, we can provide explicit headers:

```
request = urllib.request.Request(
    url = urllib.parse.urlunparse(full_url),
    method = "GET",
    headers = {
        'Accept': 'application/json',
    }
)
```

We've provided the HTTP Accept header to state MIME type results that will be produced by the server, and accepted by the client. We've provided the HTTP `Content-Type` header to state the request consumed by the server, and provided by our client script.

5. Open a context to process the response. The `urlopen()` function makes the request, handling all of the complexities of the HTTP protocol. The final `result` object is available for processing as a response:

```
with urllib.request.urlopen(request) as response:
```

6. Generally, there are three attributes of the response that are of particular interest:

```
print(response.status)
print(response.headers)
print(json.loads(response.read().decode("utf-8")))
```

The `status` is the final status code. We expect a HTTP status `200` for a normal request. The `headers` include all of the headers that are part of the response. We might, for example, want to check that the `response.headers['Content-Type']` really is `application/json`.

The value of `response.read()` are the bytes downloaded from the server. We'll often need to decode these to get proper Unicode characters. The `utf-8` encoding scheme is very common. We can use `json.loads()` to create a Python object from the JSON document.

When we run this, we'll see the following output:

```
200
Content-Type: application/json
Content-Length: 367
Server: Werkzeug/0.11.10 Python/3.5.1
Date: Sat, 23 Jul 2016 19:46:35 GMT


[{'suit': '♠', 'rank': 4, '__class__': 'Card'},
 {'suit': '♡', 'rank': 4, '__class__': 'Card'},
 {'suit': '♣', 'rank': 9, '__class__': 'Card'},
 {'suit': '♠', 'rank': 1, '__class__': 'Card'},
 {'suit': '♠', 'rank': 2, '__class__': 'Card'}]
```

The initial `200` is the status, showing that everything worked properly. There were four headers provided by the server. Finally, the internal Python object was an array of small dictionaries that provided information about the cards which were dealt.

To reconstruct `Card` objects, we'd need to use a slightly more clever JSON parser. See the *Reading JSON documents* recipe in `Chapter 8`, *Input/Output, Physical Format, and Logical Layout*.

How it works...

We've built up the request through several explicit steps:

1. The query data started as a simple dictionary with keys and values.
2. The `urlencode()` function turned the query data into a query string, properly encoded.
3. The URL as a whole started as individual components in a `ParseResult` object. This makes each piece visible, and changeable. For this particular API, the pieces are largely fixed. In other APIs, the path and the query portion of the URL might both have dynamic values.
4. The request as a whole was built from URL, method, and a dictionary of headers. This example did not provide a separate document as the body of a request. If a complex document is sent, or a file is uploaded, this is also done by providing details to the `Request` object.

The step by step assembly isn't required for a simple application. In the simple cases, a literal string value for the URL might be acceptable. At the other extreme, a more complex application may print out intermediate results as a debugging aid to be sure that the request is being constructed correctly.

The other benefit of spelling out the details like this is to provide a handy avenue for unit testing. See `Chapter 9`, *Testing*, for more information. We can often decompose a web client into request building and request processing. The request building can be tested carefully to be sure that all of the elements are set properly. The request processing can be tested with dummy results that don't involve a live connection to a remote server.

There's more...

User authentication is often an important part of a web service. For HTML-based websites—where user interaction is emphasized—people expect the server to understand a long-running sequence of transactions via a session. The person will authenticate themselves once (often with a username and password) and the server will use this information until the person logs out or the session expires.

For RESTful web services, there is rarely the concept of a session. Each request is processed separately, and the server is not expected to maintain a complex long-running transaction state. This responsibility shifts to the client application. The client is required to make appropriate requests to build up a complex document that can be presented as a single transaction.

For RESTful APIs, each request may include authentication information. We'll look at this in detail in the *Implementing Authentication for web services* recipe. For now, we'll look at providing additional details via headers. This will fit comfortably with our RESTful client script.

There are a number of ways that authentication information is provided to a web server:

- Some services use the HTTP `Authorization` header. When used with the Basic mechanism a client can provide a username and password with each request.
- Some services will invent an entirely new header with a name such as API-Key. The value for this header might be a complex string that has encoded information about the requestor.
- Some services will invent a header with a name such as `X-Auth-Token`. This may be used in a multi-step operation where a username and password credentials are sent as part of an initial request. The result will include a string value (a token) that can be used for subsequent API requests. Often, the token has a short expiration period and must be renewed.

Generally, these methods require the **Secure Socket Layer** (**SSL**) protocol. This is available as the `https` scheme. In order to handle the SSL protocol, the servers (and sometimes the clients) must have proper certificates. These are used as part of the negotiation between client and server to set up the encrypted socket pair.

All of these authentication techniques have a feature in common—they rely on sending additional information in headers. They differ slightly in which header is used, and what information is sent. In the simplest case, we might have something like the following:

```
request = urllib.request.Request(
    url = urllib.parse.urlunparse(full_url),
    method = "GET",
    headers = {
        'Accept': 'application/json',
        'X-Authentication': 'seekrit password',
    }
)
```

This hypothetical request would be for a web service that requires a password provided in an `X-Authentication` header. In the *Implementing Authentication for web services* recipe, we'll add an authentication feature to the web server.

The OpenAPI (Swagger) specification

Many servers will explicitly provide a specification as a file at a fixed, standard URL path of `/swagger.json`. The OpenAPI specification was formerly known as **Swagger**, and the filename that provides the interface reflects that history.

If provided, we can get a website's OpenAPI specification in the following way:

```
swagger_request = urllib.request.Request(
    url = 'http://127.0.0.1:5000/dealer/swagger.json',
    method = "GET",
    headers = {
        'Accept': 'application/json',
    }
)

from pprint import pprint
with urllib.request.urlopen(swagger_request) as response:
    swagger = json.loads(response.read().decode("utf-8"))
    pprint(swagger)
```

Once we have the specification, we can use it to get the details for the service or resource. We can use the technical information in the specification to build URLs, query strings, and headers.

Adding Swagger to the server

For our little demonstration server, one additional view function is required to provide the OpenAPI Swagger specification. We can update the `ch12_r03.py` module to respond to a request for `swagger.json`.

There are several ways to handle this important information:

1. A separate, static file. That's what's shown in this recipe. It's a very simple way to provide the required content.

   Here's a view function we can add that will send a file. Of course, we also need to put the specification into the named file:

   ```
   from flask import send_file
   @dealer.route('/dealer/swagger.json')
   def swagger():
       response = send_file('swagger.json',
mimetype='application/json')
       return response
   ```

   The drawback of this approach is that the specification is separate from the implementation module.

2. Embed the specification as a large blob of text in the module. We could, for example, provide the specification as the docstring for the module itself. This provides a visible place to put important documentation, but it makes it more difficult to include docstring test cases at the module level.

   This view function sends the module docstring, assuming that the string is a valid JSON document:

   ```
   from flask import make_response
   @dealer.route('/dealer/swagger.json')
   def swagger():
       response = make_response(__doc__.encode('utf-8'))
       response.headers['Content-Type'] = 'application/json'
       return response
   ```

   This has the disadvantage of requiring that we check the syntax of the docstring to be sure that it's valid JSON. This is in addition to validating that the module implementation actually conforms to the specification.

3. Create a Python specification object in proper Python syntax. This can then be encoded into JSON and transmitted. This view function sends a `specification` object. This will have to be a valid Python object that can be serialized into JSON notation:

```
from flask import make_response
import json
@dealer.route('/dealer/swagger.json')
def swagger3():
    response = make_response(
        json.dumps(specification, indent=2).encode('utf-8'))
    response.headers['Content-Type'] = 'application/json'
    return response
```

In all cases, there are several benefits to having a formal specification available:

1. Client applications can download the specification to fine-tune their processing.
2. When examples are included, the specification becomes a series of test cases for both client and server.
3. The various details of the specification can also be used by the server application to provide validation rules, defaults, and other details.

See also

- The *Parsing the query string in a request* recipe introduces the core web service
- The *Implementing Authentication for web services* recipe will add authentication to make the service more secure

Parsing the URL path

A URL is a complex object. It contains at least six separate pieces of information. More can be included as optional values.

A URL such as `http://127.0.0.1:5000/dealer/hand/player_1?$format=json` has several fields:

- `http` is the scheme. `https` is for secure connections using encrypted sockets.
- `127.0.0.1` can be called the authority, although network location is more commonly used. This particular IP address means the localhost and is a kind of loopback to the localhost. The name localhost maps to this IP address.

- `5000` is the port number, and is part of the authority.
- `/dealer/hand/player_1` is the path to a resource.
- `$format=json` is a query string.

The path to a resource can be quite complex. It's common in RESTful web services to use the path information to identify groups of resources, individual resources, and even relationships among resources.

How can we handle complex path parsing?

Getting ready

Most web services provide access to some kind of resource. In the *Implementing Web services with WSGI*, *Using the Flask framework for RESTful APIs*, and *Parsing the query string in a request* recipes, the resource was identified on the URL path as a hand or hands. This is—in a way—misleading.

There are actually two resources that are involved in those web services:

- A deck, which can be shuffled to produce one or more random hands
- A hand, which was treated as a transient response to a request

To make matters even more confusing, the hand resource was created via a `GET` request instead of the more common `POST` request. This is confusing because a `GET` request is never expected to change the state of the server.

For simple explorations and technical spikes, `GET` requests are helpful. Because a browser can make `GET` requests, these are a good way to explore some aspects of web services design.

A redesign can provide explicit access to a randomized instance of the `Deck` class. One feature of the deck will be hands of cards. This parallels the idea of `Deck` as a collection and `Hands` as a resource within the collection:

- `/dealer/decks`: A `POST` request will create a new deck object. The response to this request is taken that is used to identify the unique deck.
- `/dealer/deck/{id}/hands`: A `GET` request to this will get a hand object from the given deck identifier. The query string will specify how many cards. The query string can use the `$top` option to limit how many hands are returned. It can also use the `$skip` option to skip over some hands and get cards for later hands.

These queries will require an API client. They can't easily be done from a browser. One possibility is to use Postman as a plug-in to the Chrome browser. We'll leverage the *Making REST requests with urllib* recipe as the starting point for a client to process these more complex APIs.

How to do it...

We'll decompose this into two parts: server and client.

Server

1. Start with the *Parsing the query string in a request* recipe as a template for a Flask application. We'll be changing the view functions in that example:

```
from flask import Flask, jsonify, request, abort, make_response
from http import HTTPStatus
dealer = Flask('dealer')
```

2. Import any additional modules. In this case, we'll use the `uuid` module to create a unique key for a shuffled deck:

```
import uuid
```

   We'll also use the Werkzeug `BadRequest` response. This allows us to provide a detailed error message. This is a little nicer than using `abort(400)` for an erroneous request:

```
from werkzeug.exceptions import BadRequest
```

3. Define the global state. This includes the collection of decks. It also includes the random number generator. For testing purposes, it can help to have a way to force a particular seed value:

```
import os
import random
random.seed(os.environ.get('DEAL_APP_SEED'))
decks = {}
```

4. Define a route—a URL pattern—to a view function that performs a specific request. This is a decorator, placed immediately in front of the function. It will bind the function to the Flask application:

```
@dealer.route('/dealer/decks', methods=['POST'])
```

We've defined the decks resource and limited the route to only handling `HTTP POST` requests. This narrows the semantics of this particular endpoint—a `POST` request generally means that the URL will create something new in the server. In this example, it creates a new instance in the collection of decks.

5. Define the view function that supports this resource:

```
def make_deck():
    id = str(uuid.uuid1())
    decks[id]= Deck()
    response_json = jsonify(
        status='ok',
        id=id
    )
    response = make_response(response_json, HTTPStatus.CREATED)
    return response
```

The `uuid1()` function will create a universally unique ID based on the current host and a randomly-seeded sequence generator. The string version of this is a long hexadecimal string that looks like `93b8fc06-5395-11e6-9e73-38c9861bf556`.

We'll use this string as a key for creating a new instance of `Deck`. The response will be a small JSON document with two fields:

- The `status` field will be `'ok'` because everything worked. This allows us to perhaps provide other state information that includes warnings or errors.
- The `id` field has the ID string for the deck just created. This allows the server to have multiple, concurrent games, each of which is distinguished by a deck ID.

The response is created with the `make_response()` function so that we can provide an HTTP status of `201 CREATED` instead of the default of `200 OK`. This distinction is important because this request changes the state of the server.

6. Define a route that requires a parameter. In this case, the route will include the specific deck ID to deal from:

```
@dealer.route('/dealer/decks/<id>/hands', methods=['GET'])
```

The `<id>` makes this a path template instead of a simple, literal path. Flask will parse the `/` characters and separate the `<id>` field.

7. Define a view function that has parameters which match the template. Since the template included `<id>`, the view function has a parameter named `id` as well:

```
def get_hands(id):
    if id not in decks:
        dealer.logger.debug(id)
        return make_response(
            'ID {} not found'.format(id), HTTPStatus.NOT_FOUND)
    try:
        cards = int(request.args.get('cards',13))
        top = int(request.args.get('$top',1))
        skip = int(request.args.get('$skip',0))
        assert skip*cards+top*cards <= len(decks[id].cards), \
            "$skip, $top, and cards larger than the deck"
    except ValueError as ex:
        return BadRequest(repr(ex))
    subset = decks[id].cards[skip*cards:(skip+top)*cards]
    hands = [subset[h*cards:(h+1)*cards] for h in range(top)]
    response = jsonify(
        [
            {'hand':i, 'cards':[card.to_json() for card in hand]}
             for i, hand in enumerate(hands)
        ]
    )
    return response
```

If the value of the `id` parameter is not one of the keys to the decks collection, the function makes a `404 NOT FOUND` response. Rather than use the `abort()` function, this function uses `BadRequest` to include an explanatory error message. We could also have used the `make_response()` function in Flask.

The values of `$top`, `$skip`, and `cards` from the query string are also extracted by this function. For this example, all of the values happen to be integers, so the `int()` function is used for each value. A rudimentary sanity check is performed on the query parameters. An additional check is actually required, and the reader is encouraged to think through all of the possible bad parameters that might be used.

The `subset` variable is the portion of the deck being dealt. We've sliced the deck to start after `skip` sets of `cards`; we've included just `top` sets of `cards` in this slice. From that slice, the `hands` sequence decomposes the subset into the `top` number of hands, each of which has `cards` in it. This sequence is converted to JSON via the `jsonify()` function, and is returned.

The default status is `200 OK`, which is appropriate here because this query is an idempotent `GET` request. Each time a query is sent, the same set of cards will be returned.

8. Define a main program that runs the server:

```
if __name__ == "__main__":
    dealer.run(use_reloader=True, threaded=False)
```

Client

This will be similar to the client module from the *Making REST requests with urllib* recipe:

1. Import the essential modules for working with RESTful APIs:

```
import urllib.request
import urllib.parse
import json
```

2. There's a sequence of steps to make the `POST` request that will create a new, shuffled deck. This starts by defining the URL in pieces, by creating a `ParseResult` object manually. This will be collapsed into a single string later:

```
full_url = urllib.parse.ParseResult(
    scheme="http",
    netloc="127.0.0.1:5000",
    path="/dealer" + "/decks",
    params=None,
    query=None,
    fragment=None
)
```

3. Build a `Request` object from the URL, method, and headers:

```
request = urllib.request.Request(
    url = urllib.parse.urlunparse(full_url),
    method = "POST",
    headers = {
        'Accept': 'application/json',
    }
)
```

The default method is `GET`, which is unsuitable for this API request.

4. Send the request and process the response object. For debugging purposes, it can be helpful to print status and header information. Generally, we only need to be sure that the status was the expected `201`.

The response document should be a JSON serialization of a Python dictionary with two fields, status and ID. This client confirms the status in the response is `ok` before using the value in the `id` field:

```
with urllib.request.urlopen(request) as response:
    # print(response.status)
    assert response.status == 201
    # print(response.headers)
    document = json.loads(response.read().decode("utf-8"))

print(document)
assert document['status'] == 'ok'
id = document['id']
```

In many RESTful APIs, there will be a location header, which provides a URL that links to the object that was created.

5. Create a URL that includes inserting the ID into a URL path, as well as providing some query string arguments. This is done by creating a dictionary to model the query string, and then building a URL using a `ParseResult` object:

```
query = {'$top': 4, 'cards': 13}

full_url = urllib.parse.ParseResult(
    scheme="http",
    netloc="127.0.0.1:5000",
    path="/dealer" + "/decks/{id}/hands".format(id=id),
    params=None,
    query=urllib.parse.urlencode(query),
    fragment=None
)
```

We've inserted the `id` value into the path using `"/decks/{id}/hands/".format(id=id)`. Another way to do this is `"/".join(["", "decks", id, "hands", ""])`. Note that the empty strings are a way to force the `"/"` to appear at the beginning and end.

6. Make the `Request` object using the full URL, the method, and the standard headers:

```
request = urllib.request.Request(
    url = urllib.parse.urlunparse(full_url),
    method = "GET",
    headers = {
        'Accept': 'application/json',
    }
)
```

7. Send the request and process the response. We'll confirm that the response is `200 OK`. The response can then be parsed to get the details of the cards that are part of the requested hand:

```
with urllib.request.urlopen(request) as response:
    # print(response.status)
    assert response.status == 200
    # print(response.headers)
    cards = json.loads(response.read().decode("utf-8"))

print(cards)
```

When we run this, it will create a fresh, new `Deck` instance. Then it will deal four hands of 13 cards each. The query defines the exact number of hands and the number of cards in each hand.

How it works...

The server defines two routes that follow a common pattern for a collection and an instance of the collection. It's typical to define collection paths with a plural noun, `decks`. Using a plural noun means that the CRUD operations are focused on creating instances within the collection.

In this case, the Create operation is implemented with a `POST` method of the `/dealer/decks` path. Retrieve could be supported by writing an additional view function to handle the `GET` method of the `/dealer/decks` path. This would expose all of the deck instances in the decks collection.

If Delete is supported, this could use the `DELETE` method of `/dealer/decks`. Update (using the `PUT` method) doesn't seem to fit with the idea of a server that creates random decks.

Within the `/dealer/decks` collection, a specific deck is identified by the `/dealer/decks/<id>` path. The design calls for using the `GET` method to fetch several hands of cards from the given deck.

The remaining CRUD operations—Create, Update, and Delete—don't make much sense for this kind of `Deck` object. Once the `Deck` object is created, then a client application can interrogate the deck for various hands.

Deck slicing

The dealing algorithm makes several slices of a deck of cards. The slices are based on the fact that the size of a deck, D, must contain enough cards for the the number of hands, h, and the number of cards in each hand, c. The number of hands and cards per hand must be no larger than the size of the deck:

$$h \times c \le D$$

The social ritual of dealing often involves cutting the deck, which is a very simple shuffle done by the non-dealing player. Traditionally, each h^{th} card is assigned to each hand, H_n:

$$H_n = \{ D_{n+h \times i} : 0 \le i < c \}$$

The idea in the preceding formula is that hand $H_{n=0}$ has cards $H_0 = \{ D_0, D_h, D_{2h}, ..., D_{c \times h} \}$, hand $H_{n=1}$ has cards $H_1 = \{ D_1, D_{1+h}, D_{1+2h}, ..., D_{1+c \times h} \}$, and so on. This distribution of cards looks more fair than simply handing each player the next batch of c cards.

This isn't really necessary, and our Python program deals cards in batches that are slightly easier to compute with Python:

$$H_n = \{ D_{n \times c+1} : 0 \le i < c \}$$

The Python code creates hand $H_{n=0}$ with cards H_0={ D_0, D_1, D_2, ..., D_{c-1} }, hand $H_{n=1}$ has cards H_0 = { D_c, D_{c+1}, D_{c+2}, ..., D_{2c-1} }, and so on. Given a random deck, this is just as fair as any other allocation of cards. It's slightly simpler to enumerate in Python because it involves list slicing. For more information on slicing, see the *Slicing and dicing a list* recipe in `Chapter 4`, *Built-in Data Structures – list, set, dict*.

Client side

The client side of this transaction is a sequence of RESTful requests:

1. Ideally, the operations start with a `GET` to `swagger.json` to get the server's specifications. Depending on the server, this may be as simple as:

```
        with
urllib.request.urlopen('http://127.0.0.1:5000/dealer/swagger.json') as
response
            swagger = json.loads(response.read().decode("utf-8"))
```

2. Then, there's a `POST` to create a new `Deck` instance. This requires creating a `Request` object so that the method can be set to `POST`.
3. Then, there's a `GET` to get some hands from the deck instance. This can be done by tweaking the URL as a string template. It's slightly more general to work the URL as a collection of individual fields instead of a trivial string.

There are two ways to handle errors from RESTful applications:

- Use a simple status response such as `abort(HTTPStatus.NOT_FOUND)` for a resource that's not found.
- Use `make_response(message, HTTPStatus.BAD_REQUEST)` for a request that is in some way invalid. The message can provide needed details.

For some other status codes, such as `403 Forbidden`, we might not want to provide too many details. In the case of an authorization issue, it's often a bad idea to provide too many details. For this, `abort(HTTPStatus.FORBIDDEN)` might be appropriate.

There's more...

We'll look at some features that we should consider adding to the server:

- Check for `JSON` in the Accept header
- Provide a Swagger specification

It's common to use a header to distinguish between RESTful API requests and other requests to a server. The Accept header can provide a MIME type that distinguishes requests for JSON content from requests for user-oriented content.

The `@dealer.before_request` decorator can be used to inject a function that filters each request. This filter can distinguish proper RESTful API requests based on the following requirements:

- The Accept header includes a MIME type that includes `json`. Typically, the full MIME string is `application/json`.
- Additionally, we can make an exception for the `swagger.json` file. This can be treated as a RESTful API request irrespective of any other indicators.

Here's the additional code that implements this:

```
@dealer.before_request
def check_json():
    if request.path == '/dealer/swagger.json':
        return
    if 'json' in request.headers.get('Accept', '*/*'):
        return
    return abort(HTTPStatus.BAD_REQUEST)
```

This filter will simply return an uninformative `400 BAD REQUEST` response. To provide a more explicit error message might divulge too much information about the server's implementation. If it seems helpful, however, we can replace `abort()` with `make_response()` to return a more detailed error.

Providing a Swagger specification

A well-behaved RESTful API provides the OpenAPI specification for the various services available. This is generally packaged in the `/swagger.json` route. This doesn't necessarily mean that a literal file is available. Instead, this path is used as a focus to provide the detailed interface specification in JSON notation following the Swagger 2.0 specification.

We've defined the route, `/swagger.json`, and bound a function, `swagger3()`, to this route. This function will create a JSON representation of a global object, specification:

```
    @dealer.route('/dealer/swagger.json')
    def swagger3():
        response = make_response(json.dumps(specification,
indent=2).encode('utf-8'))
        response.headers['Content-Type'] = 'application/json'
        return response
```

The `specification` object has the following outline. Important details have been replaced with `...` to emphasize the overall structure. The details are as follows:

```
    specification = {
        'swagger': '2.0',
        'info': {
            'title': '''Python Cookbook\nChapter 10, recipe 5.''',
            'version': '1.0'
        },
        'schemes': ['http'],
        'host': '127.0.0.1:5000',
        'basePath': '/dealer',
        'consumes': ['application/json'],
        'produces': ['application/json'],
        'paths': {
            '/decks': {...}
            '/decks/{id}/hands': {...}
        }
    }
```

The two paths correspond to the two `@dealer.route` decorators in the server. This is why it's often helpful to start the design of a server with a Swagger specification, and then build the code to meet specification.

Note the small syntax difference. Flask uses `/decks/<id>/hands` where the OpenAPI Swagger specification uses `/decks/{id}/hands`. This small thing means we can't trivially copy and paste between Python and Swagger documents.

Here's the `/decks` path. This shows the input parameters that come from the query string. It also shows the details of the `201` response that contains the deck ID information:

```
    '/decks': {
     'post': {
        'parameters': [
          {
            'name': 'size',
            'in': 'query',
```

```
                'type': 'integer',
                'default': 1,
                    'description': '''number of decks to build and shuffle'''
              }
            ],
            'responses': {
              '201': {
                'description': '''Create and shuffle a deck. Returns a unique
deck id.''',
                'schema': {
                  'type': 'object',
                    'properties': {
                      'status': {'type': 'string'},
                      'id': {'type': 'string'}
                    }
                  }
                },
              '400': {
                'description': '''Request doesn't accept a JSON response'''
              }
            }
          }
```

The `/decks/{id}/hands` path has a similar structure. It defines all of the parameters that are available in the query string. It also defines the various responses; a `200` response that contains the cards and define the `404` response when the ID value was not found.

We've omitted some of the details of the parameters for each path. We've also omitted details on the structure of the deck. The outline, however, summarizes the RESTful API:

- The `swagger` key must be set to `2.0`.
- The `info` key can provide a great deal of information. This example only has the minimal requirements.
- The `schemes`, `host`, and `basePath` fields define some of the common elements of the URLs used for this service.
- The `consumes` field states what the request `Content-Type` should include.
- The `produces` field states both; that the request Accept header must state, as well as what the response `Content-Type` will be.
- The `paths` field identifies all of the paths that provide a response on this server. This shows the `/decks` and the `/decks/{id}/hands` paths.

The `swagger3()` function transforms this Python object into JSON notation and returns it. This implements what appears to be a download of a `swagger.json` file. The content specifies the resources provided by the RESTful API server.

Using a Swagger specification

In the client programming, we've used simple literal values for building the URL. The example looked like the following:

```
full_url = urllib.parse.ParseResult(
    scheme="http",
    netloc="127.0.0.1:5000",
    path="/dealer" + "/decks",
    params=None,
    query=None,
    fragment=None
)
```

Parts of this can come from the Swagger specification. We could, for example, use `specification['host']` and `specification['basePath']` instead of the `netloc` value and the first part of the `path` value. This use of the Swagger specification can provide a little bit of extra flexibility.

The Swagger specification is meant for consumption by tools that are used by people to make design decisions. The real purpose it to drive automated testing of APIs. Often, Swagger specifications will contain detailed examples that can help to clarify how to write a client application.

See also

- See the *Making REST requests with urllib* and *Parsing the query string in a request* recipes for more examples of RESTful web services

Parsing a JSON request

Many web services involve a request to create a new persistent object or make an update to an existing persistent object. In order to do these kinds of operation, the application will need input from the client.

A RESTful web service will generally accept input (and produce output) in the form of JSON documents. For more information on JSON, see the *Reading JSON documents* recipe in `Chapter 8`, *Input/Output, Physical Format, and Logical Layout*

How can we parse JSON inputs from web clients? What's an easy way to validate the input?

Getting ready

We'll extend the Flask application from the *Parsing the query string in a request* recipe to add a user registration feature; this will add a player who can then request cards. The player is a resource that will involve the essential CRUD operations:

- A client can do a `POST` to the `/players` path to create a new player. This will include a payload of a document that describes the player. The service will validate the document, and if it's valid, create a new, persistent `Player` instance. The response will include the ID assigned to the player. If the document is invalid, a response will be sent back detailing the problems.
- A client can do a `GET` to the `/players` path to get the list of players.
- A client can do a `GET` to the `/players/<id>` path to get the details of a specific player.
- A client can do a `PUT` to the `/players/<id>` path to update the details of a specific player. As with the initial `POST`, this requires a payload document that must be validated.
- A client can do a `DELETE` to the `/players/<id>` path to remove a player.

As with the *Parsing the query string in a request* recipe, we'll implement both the client and the server portion of these services. The server will handle the essential `POST` and `GET` operations. We'll leave the `PUT` and `DELETE` operations as exercises for the reader.

We'll need a JSON validator. See `https://pypi.python.org/pypi/jsonschema/2.5.1`. This is particularly good. It's helpful to have a Swagger specification validator as well. See `https://pypi.python.org/pypi/swagger-spec-validator`.

If we install the `swagger-spec-validator` package, this also installs the latest copy of the `jsonschema` project. Here's how the whole sequence might look:

```
MacBookPro-SLott:pyweb slott$ pip3.5 install swagger-spec-validator
Collecting swagger-spec-validator
  Downloading swagger_spec_validator-2.0.2.tar.gz
Requirement already satisfied (use --upgrade to upgrade):
    jsonschema in /Library/.../python3.5/site-packages
    (from swagger-spec-validator)
Requirement already satisfied (use --upgrade to upgrade):
    setuptools in /Library/.../python3.5/site-packages
    (from swagger-spec-validator)
Requirement already satisfied (use --upgrade to upgrade):
    six in /Library/.../python3.5/site-packages
    (from swagger-spec-validator)
Installing collected packages: swagger-spec-validator
```

```
    Running setup.py install for swagger-spec-validator ... done
Successfully installed swagger-spec-validator-2.0.2
```

We used the `pip` command to install the `swagger-spec-validator` package. This installation also checked that `jsonschema`, `setuptools`, and `six` were already installed.

There's a hint about using `--upgrade`. It can help to use a command such as this to upgrade a package: `pip install jsonschema --upgrade`. This might be necessary if there's a version of `jsonschema` that's below version 2.5.0.

How to do it...

We'll decompose this into three parts: Swagger specification, server, and client.

Swagger specification

1. Here's the outline of the Swagger specification:

```
specification = {
    'swagger': '2.0',
    'info': {
        'title': '''Python Cookbook\nChapter 10, recipe 6.''',
        'version': '1.0'
    },
    'schemes': ['http'],
    'host': '127.0.0.1:5000',
    'basePath': '/dealer',
    'consumes': ['application/json'],
    'produces': ['application/json'],
    'paths': {
        '/players': {...},
        '/players/{id}': {...},
    }
    'definitions': {
        'player: {..}
    }
}
```

   The first fields are essential boilerplate for RESTful web services. The `paths` and `definitions` will be filled in with the URLs and the schema definitions that are part of the service.

2. Here's the schema definition used to validate a new player. This goes inside the definition of the overall specification:

```
'player': {
    'type': 'object',
    'properties': {
        'name': {'type': 'string'},
        'email': {'type': 'string', 'format': 'email'},
        'year': {'type': 'integer'},
        'twitter': {'type': 'string', 'format': 'uri'}
    }
}
```

   The overall input document is formally described as having a type of object. There are four properties of that object:

   - A name, which is a string
   - An e-mail address, which is a string with a specific format
   - A Twitter URL, which is a string with a given format
   - A year, which is a number

   There are a few defined formats that are part of the JSON schema specification language. The `email` and `url` formats are widely used. The complete list of formats includes `date-time`, `hostname`, `ipv4`, `ipv6`, and `uri`. For details on defining a schema, see http://json-schema.org/documentation.html.

3. Here's the overall `players` path that's used to create a new player or get the entire collection of players:

```
'/players': {
    'post': {
        'parameters': [
                {
                    'name': 'player',
                    'in': 'body',
                    'schema': {'$ref': '#/definitions/player'}
                },
            ],
        'responses': {
            '201': {'description': 'Player created', },
            '403': {'description': 'Player is invalid or a
duplicate'}
        }
    },
    'get': {
```

```
                'responses': {
                    '200': {'description': 'All of the players defined so
far'},
                }
            }
        },
```

This path defines two methods—`post` and `get`. The `post` method has one parameter, called `player`. This parameter is the body of the request, and it follows the player schema provided in the definitions section.

The `get` method is shown without any parameters or any formal definition of the structure of the response.

4. Here's the definition of a path to get details about a specific player:

```
'/players/{id}': {
    'get': {
        'parameters': [
            {
                'name': 'id',
                'in': 'path',
                'type': 'string'
            }
        ],
        'responses': {
            '200': {
                'description': 'The details of a specific player',
                'schema': {'$ref': '#/definitions/player'}
            },
            '404': {'description': 'Player ID not found'}
        }
    }
},
```

That path is similar to the one shown in the *Parsing the URL path* recipe. The `player` key is provided in the URL. The response when a player ID is valid is shown in detail. The response has a defined schema that also uses the player schema definition in the definitions section.

This specification will be part of the server. It can be provided by a view function defined in the `@dealer.route('/swagger.json')` route. It's often simplest to create a file with this specification document in it.

Server

1. Start with the *Parsing the query string in a request* recipe as a template for a Flask application. We'll be changing the view functions:

```
from flask import Flask, jsonify, request, abort, make_response
from http import HTTPStatus
```

2. Import the additional libraries required. We'll use the JSON schema for validation. We'll also compute hashes of strings to serve as useful external identifiers in URLs:

```
from jsonschema import validate
from jsonschema.exceptions import ValidationError
import hashlib
```

3. Create the application and the database of players. We'll use a simple global variable. A larger application might use a proper database server to save this information:

```
dealer = Flask('dealer')
players = {}
```

4. Define the route for posting to the overall collection of `players`:

```
@dealer.route('/dealer/players', methods=['POST'])
```

5. Define the function that will parse the input document, validate the content, and then create the persistent `player` object:

```
def make_player():
    document = request.json
    player_schema = specification['definitions']['player']
    try:
        validate(document, player_schema)
    except ValidationError as ex:
        return make_response(ex.message, 403)

    id =
hashlib.md5(document['twitter'].encode('utf-8')).hexdigest()
    if id in players:
        return make_response('Duplicate player', 403)

    players[id] = document

    response = make_response(
        jsonify(
```

```
            status='ok',
            id=id
        ),
        201
    )
    return response
```

This function follows a common four-step design:

- Validate the input document. The schema is defined as part of the overall Swagger specification.
- Create a key and confirm that it's unique. This is a key that's derived from the data. We might also create unique keys using the `uuid` module.
- Persist the new document in the database. In this example, it's only a single statement, `players[id] = document`. This follows the ideal that a RESTful API is built around classes and functions that already provide a complete implementation of the features.
- Build a response document.

6. Define a main program that runs the server:

```
if __name__ == "__main__":
    dealer.run(use_reloader=True, threaded=False)
```

We can add other methods to see multiple players or individual players. These will follow the essential designs of the *Parsing the URL path* recipe. We'll look at these in the next section.

Client

This will be similar to the client module from the *Parsing the URL path* recipe:

1. Import the essential modules for working with RESTful APIs:

```
import urllib.request
import urllib.parse
import json
```

2. Create the URL in pieces by creating a `ParseResult` object manually. This will be collapsed into a single string later:

```
full_url = urllib.parse.ParseResult(
    scheme="http",
    netloc="127.0.0.1:5000",
    path="/dealer" + "/players",
    params=None,
    query=None,
    fragment=None
)
```

3. Create an object that can be serialized to a JSON document and posted to the server. Studying `swagger.json` shows what this document's schema must be. The `document` will include the required four properties:

```
document = {
    'name': 'Xander Bowers',
    'email': 'x@example.com',
    'year': 1985,
    'twitter': 'https://twitter.com/PacktPub'
}
```

4. We'll combine URL, document, method, and headers to create the complete request. This will use `urlunparse()` to collapse the URL parts into a single string. The `Content-Type` header alerts the server that we're going to provide a text document in JSON notation:

```
request = urllib.request.Request(
    url = urllib.parse.urlunparse(full_url),
    method = "POST",
    headers = {
        'Accept': 'application/json',
        'Content-Type': 'application/json;charset=utf-8',
    },
    data = json.dumps(document).encode('utf-8')
)
```

We've included the `charset` option, which specifies the specific encoding used to create bytes from Unicode strings. Since `utf-8` encoding is the default, this isn't required. In the rare case that a different encoding is used, this shows how to provide the alternative.

5. Send the request and process the `response` object. For debugging purposes, it can be helpful to print the `status` and `headers` information. Generally, we only need to be sure that the `status` was the expected `201 CREATED`:

```
with urllib.request.urlopen(request) as response:
    # print(response.status)
    assert response.status == 201
    # print(response.headers)
    document = json.loads(response.read().decode("utf-8"))

print(document)
assert document['status'] == 'ok'
id = document['id']
```

   We've examined the response document to assure that it includes the two expected fields.

We can also include other queries in this client. We might want to retrieve all players or retrieve a specific player. These will follow the design shown in the *Parsing the URL path* recipe.

How it works...

Flask automatically examines inbound documents to parse them. We can simply use `request.json` to leverage the automated JSON parsing that's built-in to Flask.

If the input is not actually JSON, then the Flask framework will return a `400 BAD REQUEST` response. This happens when our server application references the `json` property of the request. We can use a `try` statement to capture the `400 BAD REQUEST` response object and make changes to it, or possibly return a different response.

We've used the `jsonschema` package to validate the input document. This will check a number of features of the JSON document:

- It checks if the overall type of the JSON document matches the overall type of the schema. In this example, the schema required an object, which is a `{}` JSON structure.

- For each property defined in the schema and present in the document, it confirms that the value in the document matches the schema definition. This means that the value fits one of the defined JSON types. If there are other validation rules like a format, or a range specification, or a number of elements for an array, these constraints are checked also. This check proceeds recursively through all levels of the schema.
- If there's a required list of fields, it checks that all of these are actually present in the document.

For this recipe, we've kept the details of the schema to a minimum. A common feature that we've omitted in this example is the list of required properties. We can also provide considerably more detailed attribute descriptions. The year, for example, should probably have a minimum value of `1900`.

We've kept the database update processing to a minimum in this example. In some cases, the database insert might involve a much more complex process where a database client connection is used to execute a command that changes the state of a database server. Ideally, the database processing is kept to a minimum—the application-specific details are often imported from a separate module and presented as RESTful API resources.

In a larger application, there might be a `player_db` module that included all of the player database processing. This module would define all of the classes and functions. This would often provide the detailed schema definitions for a `player` object. The RESTful API service would import these classes, functions, and schema specifications and expose them for external consumers.

There's more...

The Swagger specification allows examples of response documents. This is often helpful in several ways:

- It's common to start designing the sample document that is part of the response. Writing a schema specification that describes a document can be difficult and the schema validation feature helps to ensure that the specification matches the document.
- Once the specification is complete, the next step is to write the server-side programming. It's helpful to have unit tests that leverage schema example documents.
- For users of the Swagger specification, a concrete example of the response can be used to design the client, and write unit tests for the client-side programming.

We can use the following code to confirm that a server has a valid Swagger specification. If this raises an exception, either there's no Swagger document or the document doesn't properly fit the Swagger schema:

```
from swagger_spec_validator import validate_spec_url
validate_spec_url('http://127.0.0.1:5000/dealer/swagger.json')
```

Location header

The `201 CREATED` response included a small document with some status information. The status information included the key that was assigned to the newly-created record.

It's also common for a `201 CREATED` response to have an additional location header in the response. This header will provide a URL that can be used to recover the document which was created. For this application, the location would be a URL, like the following example: `http://127.0.0.1:5000/dealer/players/75f1bfbda3a8492b74a33ee28326649c`.

The location header can be saved by a client. A complete URL is slightly simpler than creating a URL from a URL template and a value.

The server can build this header as follows:

```
response.headers['Location'] = url_for('get_player', id=str(id))
```

This relies on the Flask `url_for()` function. This function takes the name of a view function, and any parameters that come from the URL path. It then uses the route for the view function to construct a complete URL. This will include all the information for the currently running server. After the header is inserted, the `response` object can be returned.

Additional resources

The server should be able to respond with a list of players. Here's a minimal implementation that simply transforms the data into a large JSON document:

```
@dealer.route('/dealer/players', methods=['GET'])
def get_players():
    response = make_response(jsonify(players))
    return response
```

A more sophisticated implementation would support the `$top` and `$skip` query parameters to page through the list of players. Additionally, a `$filter` option might be useful to implement a search for a subset of players.

In addition to the generic query for all players, we need to implement a method that will return an individual player. This kind of view function is often just as simple, as shown in the following code:

```
@dealer.route('/dealer/players/<id>', methods=['GET'])
def get_player(id):
    if id not in players:
        return make_response("{} not found".format(id), 404)

    response = make_response(
        jsonify(
            players[id]
        )
    )
    return response
```

This function confirms that the given ID is a proper key value in the database. If the key is not in the database, the database document is transformed into JSON notation and returned.

Query for a specific player

Here's the client processing required to locate a specific value in the database. This involves multiple steps:

1. First, we'll create the URL for a particular player:

```
id = '75f1bfbda3a8492b74a33ee28326649c'
full_url = urllib.parse.ParseResult(
    scheme="http",
    netloc="127.0.0.1:5000",
    path="/dealer" + "/players/{id}".format(id=id),
    params=None,
    query=None,
    fragment=None
)
```

   We've built the URL from pieces of information. This is created as a `ParseResult` object with separate fields.

2. Given the URL, we can then create a `Request` object:

```
request = urllib.request.Request(
    url = urllib.parse.urlunparse(full_url),
    method = "GET",
    headers = {
        'Accept': 'application/json',
    }
)
```

3. Once we have the `request` object, we can then make the request, and retrieve the response. We need to confirm that the response status is `200`. If so, we can then parse the body of the response to get the JSON document that describes a given player:

```
with urllib.request.urlopen(request) as response:
    assert response.status == 200
    player= json.loads(response.read().decode("utf-8"))
print(player)
```

If the player doesn't exist, the `urlopen()` function will raise an exception. We can enclose this in a `try` statement to capture the `403 NOT FOUND` exceptions that could be raised if the player ID doesn't exist.

Exception handling

Here's the general pattern for all client requests. This includes the explicit `try` statement:

```
try:
    with urllib.request.urlopen(request) as response:
        # print(response.status)
        assert response.status == 201
        # print(response.headers)
        document = json.loads(response.read().decode("utf-8"))

    # process the document here.

except urllib.error.HTTPError as ex:
    print(ex.status)
    print(ex.headers)
    print(ex.read())
```

There are actually two general kinds of exception:

- **Lower-level exceptions**: This exception indicates that the server can't be contacted. The `ConnectionError` exception is a common example of this lower-level exception. This is a subclass of the `OSError` exception.
- **The HTTPError exceptions from the urllib module**: This exception means that the overall HTTP protocol worked, but the response from the server was not a successful status code. Success is generally a value in the range `200` to `299`.
- The `HTTPError` exception has similar attributes to a proper response. It includes a status, headers, and a body.

In some cases, an `HTTPError` exception might be one of several expected responses from a server. It might not indicate an error or problem. It might simply be another meaningful status code.

See also

- See the *Parsing the URL path* recipe for other examples of URL processing.
- The *Making REST requests with urllib* recipe shows other examples of query string processing.

Implementing authentication for web services

Security, in general, is a pervasive issue. Every part of an application will have security considerations. Parts of the implementation of security will involve two closely-related issues:

- **Authentication**: A client must provide some evidence of who they are. This might involve signed certificates or it might involve credentials like a username and password. It might involve multiple factors, such as an SMS message to a phone that the user should have access to. The web server must validate this authentication.
- **Authorization**: A server must define areas of authority and allocate these to groups of users. Furthermore, individual users must be defined as members of the authorization groups.

While it's technically possible to define authorization on an individual basis, this tends to become awkward as a site or application grows and changes. It's easier to define security for groups. In some cases, a group may (initially) have only a single individual.

Application software must implement authorization decisions. For Flask, the authorization can be part of each view function. The connection of individual to group and group to view function defines the resources available to any specific user.

Confusingly, the HTTP standards provide authentication credentials using the HTTP `Authorization` header. This may lead to some confusion because the header's name doesn't precisely reflect its purpose.

There are a variety of ways that authentication details can be provided from a web client to a web server. Here are a few of the alternatives:

- **Certificates**: Certificates which are encrypted and include a digital signature as well as a reference to a **CertificateAuthority** (**CA**): These are exchanged by the **Secure Socket Layer** (**SSL**). In some environments, both client and server must have certificates that are used for mutual authentication. In other environments, the server provides a certificate of authenticity, but the client does not. This is common for the `https` scheme. The server doesn't verify the client's certificate.
- **Static API keys or tokens**: A web service might provide a simple, fixed key. This might be issued with advice to keep it secret, much like a password.
- **Usernames and passwords**: The web server might identify users by a username and password. User identity might be further confirmed using e-mail or SMS messages.
- **Third-party authentication**: This might involve using a service such as OpenID. For details, see `http://openid.net`. This will involve a callback URL so that notification information can be returned by the OpenID provider.

Additionally, there's a question of how the user information gets loaded into a web server. Some websites are self-service, with users providing some minimal contact information and being granted access to the content.

In many cases, websites aren't self-service. A user might be carefully vetted before being allowed access. Access might involve contracts and fees for access to data or services. In some cases, one company will purchase licenses for their employees, providing a finite list of users who have access to a given suite of web services.

This recipe will show a self-service application in which there is no defined set of users. This means that there must be a web service to create new users that doesn't require any authentication. All other services will require a properly authenticated user.

Getting ready

We'll implement a version of HTTP-based authentication using the `Authorization` header. There are two variations on this theme:

- **HTTP basic authentication**: This uses a simple username and password string. It relies on the SSL layer to encrypt the traffic between client and server.
- **HTTP digest authentication**: This uses a much more complex hash of username, password, and a nonce provided by the server. The server computes the expected hash value. If the hash values match, then the same bytes were used to compute the hash, and the password must have been valid. This doesn't require SSL.

SSL is frequently used by web servers to establish their authenticity. Because this technology is so pervasive, it means that HTTP basic authentication can be used. This is a huge simplification in RESTful API processing, since each request will include the `Authorization` header and secure sockets will be used between client and server.

Configuring SSL

The details of getting and configuring certificates is outside of the realm of Python programming. The OpenSSL package provides tools for creating self-signed certificates that can be used for configuring a secure server. CAs such as Comodo Group and Symantec offer trusted certificates that are widely recognized by OS vendors, as well as the Mozilla Foundation.

There are two parts to creating a certificate with OpenSSL:

1. Create a private key file. This is generally done with the following OS-level command:

```
slott$ openssl genrsa 1024 > ssl.key
Generating RSA private key, 1024 bit long modulus
.......++++++
..........................++++++
e is 65537 (0x10001)
```

   The `openssl genrsa 1024` command created a private key file, which was saved under the name `ssl.key`.

2. Create a certificate using the key file. The following command is one way to handle this:

```
slott$ openssl req -new -x509 -nodes -sha1 -days 365 -key ssl.key >
ssl.cert
```

You are about to be asked to enter information that will be incorporated into your certificate request. What you are about to enter is what is called a **Distinguished Name** (**DN**). There are quite a few fields but you can leave some blank. For some fields there will be a default value. If you enter `.`, the field will be left blank.

```
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Virginia
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty
Ltd]:ItMayBeAHack
Organizational Unit Name (eg, section) []:Common Name (e.g. server
FQDN or YOUR name) []:Steven F. Lott
Email Address []:
```

The command `openssl req -new -x509 -nodes -sha1 -days 365 -key ssl.key` created the private certificate file, which was saved in `ssl.cert`. This certificate is privately signed, and doesn't have a CA. It provides only a limited set of features.

These two steps create two files: `ssl.cert` and `ssl.key`. We'll use these files below to secure the server.

Users and credentials

In order for users to be able to supply a username and a password, we'll need to store this information on the server. There's a very important rule about user credentials:

Never store credentials. Never.

It should be clear that storing plain text passwords is an invitation to a security disaster. What's less obvious is that we can't even store encrypted passwords. When the key used to encrypt the passwords is compromised, that will lead to a loss of all of the user identities.

How can a user's password be checked if we do not store the password?

The solution is to store a hash instead of a password. The first time the password is created, the server saves the hashed summary. Each time after that, the user's input is hashed and compared with the saved hash. If the two hashes match, then the password must have been correct. What's central is the extreme difficulty of recovering a password from the hash.

There is a three-step process to create the initial hash value for a password:

1. Create a random `salt` value. Generally, 16 bytes from `os.urandom()` are used.
2. Use the `salt` plus the password to create a `hash` value. Generally, the `hashlib` is used for this. Specifically, `hashlib.pbkdf2_hmac()`. A specific digest algorithm is used for this, for example, `md5` or `sha224`.
3. Save the digest name, the `salt`, and the hashed bytes. Often this is combined into a single string that looks like—`md5$salt$hash`. The `md5` is a literal. The `$` separates the algorithm name, `salt`, and `hash` values.

When a password needs to be checked, a similar process is followed:

1. Given the username, locate the saved hash string. This will have a three-part structure of the digest algorithm name, saved salt, and hashed bytes. The elements may be separated by `$`.
2. Use the saved salt plus the user-supplied candidate password to create a computed `hash` value.
3. If the computed hash bytes match the saved hash bytes, we know the digest algorithm and salt matched; therefore, the password must have matched as well.

We'll define a simple class to retain user information as well as the hashed password. We can use Flask's `g` object to save the user information during request processing.

Flask view function decorator

There are several alternatives for handling the authentication checks:

- If every route has the same security requirements, then the `@dealer.before_request` function can be used to validate all `Authorization` headers. This would require some exception processing for the `/swagger.json` route and the self-service route that allows an unauthorized user to create their new username and password credentials.
- When some routes require authentication and some don't, it works out well to introduce a decorator for the routes that need authentication.

A Python decorator is a function that wraps another function to extend its functionality. The core technique looks like this:

```
from functools import wraps
def decorate(function):
    @wraps(function)
    def decorated_function(*args, **kw):
        # processing before
        result = function(*args, **kw)
        # processing after
        return result
    return decorated_function
```

The idea is to replace a given function, The `function` in this example, with a new function, `decorated_function`. Within the body of the decorated function, it executes the original function. Some processing can be done before and some processing done after the function being decorated.

In a Flask context, we'll put our decorators after the `@route` decorator:

```
@dealer.route('/path/to/resource')
@decorate
def view_function():
    return make_result('hello world', 200)
```

We've wrapped the `view_function()` with the `@decorate` decorator. A decorator can check authentication to be sure that the user is known. We can do a wide variety of processing in these functions.

How to do it...

We'll decompose this into four parts:

- Defining the `User` class
- Defining a view decorator
- Creating the server
- Creating an example client

Defining the User class

This class definition provides an example of a definition of an individual `User` object:

1. Import modules that are required to create and check the password:

```
import hashlib
import os
import base64
```

   Other useful modules include `json` so that a `User` object can be properly serialized.

2. Define the `User` class:

```
class User:
```

3. Since we'll be changing some aspects of password generation and checking, we'll provide two constants as part of the overall class definition:

```
DIGEST = 'sha384'
ROUNDS = 100000
```

   We'll use the **SHA-384** digest algorithm. This provides 64-byte summaries. We'll use 100,000 rounds for the **Password-Based Key Derivation Function 2** (**PBKDF2**) algorithm.

4. Most of the time, we'll create users from a JSON document. This will be a dictionary that can be turned into keyword argument values using `**`:

```
def __init__(self, **document):
    self.name = document['name']
    self.year = document['year']
    self.email = document['email']
    self.twitter = document['twitter']
    self.password = None
```

Note that we don't expect to set the password directly. Instead, we'll set the password separately from creating the user document.

We've omitted additional authorization details, such as a list of groups to which the user belongs. We've also omitted an indicator showing that the password needs to be changed.

5. Define the algorithm for setting the password `hash` value:

```
def set_password(self, password):
    salt = os.urandom(30)
    hash = hashlib.pbkdf2_hmac(
        self.DIGEST, password.encode('utf-8'), salt, self.ROUNDS)
    self.password = '$'.join(
        [self.DIGEST,
         base64.urlsafe_b64encode(salt).decode('ascii'),
         base64.urlsafe_b64encode(hash).decode('ascii')
        ]
    )
```

We've built a random salt using `os.urandom()`. We've then built the complete `hash` value using the given digest algorithm, the password, and `salt`. We've used a configurable number of rounds.

Note that the hash computation works in bytes, not Unicode characters. We've encoded the password into bytes using the `utf-8` encoding.

We assembled a string using the name of the digest algorithm, the salt, and the encoded `hash` value. We've used URL-safe `base64` encoding of the bytes so that the full, hashed password value can be displayed easily. It can be saved in any kind of database because it uses only `A-Z`, `a-z`, `0-9`, `-` and `_`.

Note that `urlsafe_b64encode()` creates a string of byte values. These must be decoded to see what Unicode characters they represent. We use the ASCII encoding scheme here because `base64` only uses sixty-four standard ASCII characters.

6. Define an algorithm for checking a password hash value:

```
def check_password(self, password):
    digest, b64_salt, b64_expected_hash = self.password.split('$')
    salt = base64.urlsafe_b64decode(b64_salt)
    expected_hash = base64.urlsafe_b64decode(b64_expected_hash)
    computed_hash = hashlib.pbkdf2_hmac(
        digest, password.encode('utf-8'), salt, self.ROUNDS)
    return computed_hash == expected_hash
```

We've decomposed the password hash into `digest`, `salt`, and `expected_hash` value. Since the various parts were `base64` encoded, they must be decoded to recover the original bytes.

Note that the hash computation works in bytes, not Unicode characters. We've encoded the password into bytes using the `utf-8` encoding. The computed results of `hashlib.pbkdf2_hmac()` are compared with the expected results. If they match, then the passwords must have been the same.

Here's a demonstration of how this class is used:

```
>>> details = {'name': 'xander', 'email': 'x@example.com',
...     'year': 1985, 'twitter': 'https://twitter.com/PacktPub' }
>>> u = User(**details)
>>> u.set_password('OpenSesame')
>>> u.check_password('opensesame')
False
>>> u.check_password('OpenSesame')
True
```

This test case can be included in the class docstring. See the *Using docstrings for testing* recipe in `Chapter 9`, *Testing*, for more information on this kind of test case.

In more complex applications, there may also be a definition for the collection of users. This often uses a database of some kind to facilitate locating users and inserting new users.

Defining a view decorator

1. Import the `@wraps` decorator from `functools`. This helps define decorators by assuring that the new function has the original name and docstring copied from the function that is being decorated:

```
from functools import wraps
```

2. In order to check passwords, we'll need the `base64` module to help decompose the value of the `Authorization` header. We'll also need to report errors and update the Flask processing context using the global `g` object:

```
import base64
from flask import g
from http import HTTPStatus
```

3. Define the decorator. All decorators have this essential outline. We'll replace the `processing here` part in the next step:

```
def authorization_required(view_function):
    @wraps(view_function)
    def decorated_function(*args, **kwargs):
        processing here
    return decorated_function
```

4. Here are the processing steps to examine the header. Note that every problem encountered simply aborts processing with the `401 UNAUTHORIZED` as the status code. To prevent hackers from exploring the algorithm, all of the results are identical even though the root causes are different:

```
if 'Authorization' not in request.headers:
    abort(HTTPStatus.UNAUTHORIZED)
kind, data = request.headers['Authorization'].split()
if kind.upper() != 'BASIC':
    abort(HTTPStatus.UNAUTHORIZED)
credentials = base64.decode(data)
username, _, password = credentials.partition(':')
if username not in user_database:
    abort(HTTPStatus.UNAUTHORIZED)
if not user_database[username].check_password(password):
    abort(HTTPStatus.UNAUTHORIZED)
g.user = user_database[username]
return view_function(*args, **kwargs)
```

There are a number of conditions that must be successfully passed:

- An `Authorization` header must be present
- The header must specify basic authentication
- The value must include a `username:password` string encoded using `base64`
- The username must be a known username
- The computed hash from the password must match the expected password hash

Any single failure leads to a `401 UNAUTHORIZED` response.

Creating the server

This parallels the server shown in the *Parsing a JSON request* recipe. There are some important modifications:

1. Create the local self-signed certificate or purchase a certificate from a certificate authority. For this recipe, we'll assume the two filenames are `ssl.cert` and `ssl.key`.
2. Import the modules required to build a server. Also import the `User` class definition:

```
from flask import Flask, jsonify, request, abort, url_for
from ch12_r07_user import User
from http import HTTPStatus
```

3. Include the `@authorization_required` decorator definition.
4. Define a route with no authentication. This will be used to create new users. A similar view function was defined in the *Parsing a JSON request* recipe. This version requires a password property in the incoming document. This will be the plain-text password that's used to create the hash. The plain text password is not saved anywhere; only the hash is retained:

```
@dealer.route('/dealer/players', methods=['POST'])
def make_player():
    try:
        document = request.json
    except Exception as ex:
        # Document wasn't even JSON. We can fine-tune
        # the error message here.
        raise
    player_schema = specification['definitions']['player']
    try:
        validate(document, player_schema)
```

```
            except ValidationError as ex:
                return make_response(ex.message, 403)

            id =
hashlib.md5(document['twitter'].encode('utf-8')).hexdigest()
            if id in user_database:
                return make_response('Duplicate player', 403)

            new_user = User(**document)
            new_user.set_password(document['password'])
            user_database[id] = new_user

            response = make_response(
                jsonify(
                    status='ok',
                    id=id
                ),
                201
            )
            response.headers['Location'] = url_for('get_player',
id=str(id))
            return response
```

After creating the user, the password is set separately. This follows the pattern set by some applications where users are loaded in bulk. This processing might provide a temporary password for each user, which must be immediately changed.

Note that each user is assigned a cryptic ID. The assigned ID is computed from a hex digest of their Twitter handle. This is unusual, but it shows that there's a great deal of flexibility available.

If we wanted users to choose their own username, we'd need to add that to the request document. We would use that username instead of the computed ID value.

5. Define a route for which authentication is required. A similar view function was defined in the *Parsing a JSON request* recipe. This version uses the `@authorization_required` decorator:

```
@dealer.route('/dealer/players/<id>', methods=['GET'])
@authorization_required
def get_player(id):
    if id not in user_database:
        return make_response("{} not found".format(id), 404)
```

```
        response = make_response(
            jsonify(
                players[id]
            )
        )
        return response
```

Most of the other routes will have similar `@authorization_required` decorators. Some routes, such as the `/swagger.json` route, will not require authorization.

6. The `ssl` module defines the `ssl.SSLContext` class. The context can be loaded with the self-signed certificate and private key file created previously. The context is then used by the Flask object's `run()` method. This will change scheme in the URL from `http://127.0.01:5000` to `https://127.0.0.1:5000`:

```
import ssl
ctx = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
ctx.load_cert_chain('ssl.cert', 'ssl.key')
dealer.run(use_reloader=True, threaded=False, ssl_context=ctx)
```

Creating an example client

1. Create an SSL context that will work with a self-signed certificate:

```
import ssl
context = ssl.create_default_context(ssl.Purpose.SERVER_AUTH)
context.check_hostname = False
context.verify_mode = ssl.CERT_NONE
```

This context can be used with all `urllib` requests. This will politely ignore the lack of CA signature on the certificate.

Here's how we use this context to fetch the Swagger specification:

```
        with urllib.request.urlopen(swagger_request, context=context) as
response:
            swagger = json.loads(response.read().decode("utf-8"))
            pprint(swagger)
```

2. Create the URL for creating a new player instance. Note that we must use `https` for the scheme. We've built a `ParseResult` object to show the various pieces of the URL separately:

```
full_url = urllib.parse.ParseResult(
    scheme="https",
    netloc="127.0.0.1:5000",
    path="/dealer" + "/players",
    params=None,
    query=None,
    fragment=None
)
```

3. Create a Python object that will be serialized into a JSON document. This schema is similar to the example shown in the *Parsing a JSON request* recipe. This includes one extra property, which is the plain text:

```
password.document = {
    'name': 'Hannah Bowers',
    'email': 'h@example.com',
    'year': 1987,
    'twitter': 'https://twitter.com/PacktPub',
    'password': 'OpenSesame'
}
```

   Because the SSL layer uses an encrypted socket, sending a plain text password like this is feasible.

4. We'll combine URL, document, method, and headers to create the complete `Request` object. This will use `urlunparse()` to collapse the URL parts into a single string. The `Content-Type` header alerts the server that we're going to provide a text document in JSON notation:

```
request = urllib.request.Request(
    url = urllib.parse.urlunparse(full_url),
    method = "POST",
    headers = {
        'Accept': 'application/json',
        'Content-Type': 'application/json;charset=utf-8',
    },
    data = json.dumps(document).encode('utf-8')
)
```

5. We can post this document to create a new player:

```
        try:
            with urllib.request.urlopen(request, context=context) as
response:
                # print(response.status)
                assert response.status == 201
                # print(response.headers)
                document = json.loads(response.read().decode("utf-8"))

            print(document)
            assert document['status'] == 'ok'
            id = document['id']
        except urllib.error.HTTPError as ex:
            print(ex.status)
            print(ex.headers)
            print(ex.read())
```

   The happy path will receive a `201` status response, and the user will be created. The response will include the assigned user ID plus a redundant status code.

   If the user is a duplicate, or the document doesn't match the schema, then there will be an `HTTPError` exception raised. This may have useful error messages that can be displayed.

6. We can use the assigned ID and the known password to create an `Authorization` header:

```
        import base64
        credentials =
base64.b64encode(b'75f1bfbda3a8492b74a33ee28326649c:OpenSesame')
```

   The `Authorization` header has a two-word value: `b"BASIC " + credentials`. The word `BASIC` is required. The credentials must be a `base64` encoding of the `username:password` string. In this example, the username is a specific ID assigned when the user was created.

7. Here's a URL to query all of the players. We've built a `ParseResult` object to show the various pieces of the URL separately:

```
        full_url = urllib.parse.ParseResult(
            scheme="https",
            netloc="127.0.0.1:5000",
            path="/dealer" + "/players",
            params=None,
            query=None,
```

```
        fragment=None
    )
```

8. We can combine the URL, method, and headers into a single `Request` object. This includes the `Authorization` header, which has the `base64` encoding of username and password:

```
request = urllib.request.Request(
    url = urllib.parse.urlunparse(full_url),
    method = "GET",
    headers = {
        'Accept': 'application/json',
        'Authorization': b"BASIC " + credentials
    }
)
```

9. The `Request` object can be used to make the query from the server and process the response with `urllib`:

```
request.urlopen(request, context=context) as response:
    assert response.status == 200
    # print(response.headers)
    players = json.loads(response.read().decode("utf-8"))

pprint(players)
```

The expected status is `200`. The response should be a JSON document with a list of known `players`.

How it works...

There are three parts to this recipe:

- **Using SSL to provide a secure channel**: This makes it possible to exchange usernames and passwords directly. Instead of the more complex HTTP digest authentication, we can use the simpler HTTP basic authentication scheme. There are a variety of other authentication schemes used by web services; most of them require SSL.
- **Using best practices for password hashing**: Saving passwords in any form is a security risk. Rather than save plain passwords, or even encrypted passwords, we only save a computed hash value of a password and a random salt string. This assures us that it's nearly impossible to reverse engineer passwords from the hashed values.

- **Using a decorator**: It is used to distinguish between routes that require authentication and routes that do not require authentication. This allows a great deal of flexibility in creating a web service.

In cases where all routes require authentication, we could add the password check algorithm to the `@dealer.before_request` function. This would centralize all authentication checks. It would also mean that a separate administrative process is required to define users and hashed passwords.

What's essential here is that the security check on the server is a simple `@authorization_required` decorator. It's very easy to be sure that it is in place on all view functions.

There's more...

This server has a relatively simple set of authorization rules:

- Most routes require a valid user. This was implemented by the presence of the `@authorization_required` decorator in the view function.
- A `GET` for `/dealer/swagger.json` and a `POST` to `/dealer/players` do not require a valid user. This was implemented by the absence of an additional decorator.

In many cases, we'll have a considerably more complex configuration of privileges, groups, and users. The principle of least privilege suggests that the users should be segregated into groups, and that each group has the fewest privileges possible to accomplish their goals.

This often means that we'll have an administrative group that creates new users, but has no other access to use the RESTful web services. Users can access the web services, but are unable to create any additional users.

This requires several changes to our data model. We should define user groups and assign users to those groups:

```
class Group:
    '''A collection of users.'''
    pass

administrators = Group()
players = Group()
```

We can then expand the definition of `User` to include group membership:

```
class GroupUser(User):
    def __init__(self, *args, **kw):
        super().__init__(*args, **kw)
        self.groups = set()
```

When we create a new instance of the `GroupUser` class, we can also assign them to a particular group:

```
u = GroupUser(**document)
u.groups = set(players)
```

We can now expand our decorator to check the `groups` attribute of the authenticated user. A decorator with parameters is a bit more complex than a parameterless decorator:

```
def group_member(group_instance):
    def group_member_decorator(view_function):
        @wraps(view_function)
        def decorated_view_function(*args, **kw):
            # Check Password and determine user
            if group_instance not in g.user.groups:
                abort(HTTPStatus.UNAUTHORIZED)
            return view_function(*args, **kw)
        return decorated_view_function
    return group_member_decorator
```

A decorator with a parameter works by creating a concrete decorator that includes the parameter. The concrete decorator, `group_member_decorator`, will wrap a given view function. This will parse the `Authorization` header, locate the `GroupUser` instance and check the group membership.

We've used `# Check Password and determine user` as a placeholder for a refactored function to check the `Authorization` header. The core functionality of the `@authorization_required` decorator needs to be extracted into a stand-alone function so it can be used in several places.

We can then use this decorator as follows:

```
@dealer.route('/dealer/players')
@group_member(administrators)
def make_player():
    etc.
```

This narrows the scope of privilege for each individual view function. It provides assurance that the principle of least privilege is followed by the RESTful web services.

Creating a command-line interface

When working with a site that has special administrator privileges, we often need to provide a way to create an initial administrative user. This user can then create all of the users with non-administrative privileges. This is often done with a CLI application which is run by the administrative user directly on the web server.

Flask supports this with a decorator that defines commands that must be run outside the RESTful web services environment. We can use `@dealer.cli.command()` to define a command that is run from the command line. This command can, for example, load the initial administrative user. A command might be created to load users from a list, also.

The `getpass` module is a way for an administrative user to provide their initial password in a way that won't be echoed on a terminal. This can provide confidence that the site's credentials are being processed securely.

Building the Authentication header

Web services that rely on an HTTP basic `Authorization` header can be supported in one of two common ways:

- Build the `Authorization` header with the credentials and include this in each request. To do this, we need to provide the proper `base64` encoding of the string `username:password`. This alternative has the advantage of being relatively simple.
- Use the `urllib` features to provide the `Authorization` header automatically:

```
        from urllib.request import HTTPBasicAuthHandler,
HTTPPasswordMgrWithDefaultRealm
        auth_handler = urllib.request.HTTPBasicAuthHandler(
            password_mgr=HTTPPasswordMgrWithDefaultRealm)
        auth_handler.add_password(
            realm=None,
            uri='https://127.0.0.1:5000/',
            user='Aladdin',
            passwd='OpenSesame')
        password_opener = urllib.request.build_opener(auth_handler)
```

We've created an instance of `HTTPBasicAuthHandler`. This is populated with all of the usernames and passwords that might be required. For complex applications that gather data from multiple sites, there may be more than one set of credentials added to the handler.

Instead of using `with urllib.request.urlopen(request) as response:`, we would now use `with password_opener(request) as response:`. The `Authorization` header is added to the request by the `password_opener` object.

This alternative has the advantage of being relatively flexible. We can switch to using `HTTPDigestAuthHandler` without any difficulties. We can also add additional usernames and passwords.

The realm information is sometimes confusing. A realm is a container for multiple URLs. When a server requires authentication, it will respond with a `401` status code. This response will include an `Authenticate` header that names a realm to which the credentials must belong. Since the realm contains multiple site URLs, the realm information tends to be extremely static. `HTTPBasicAuthHandler` uses the realm and URL information to choose which of the usernames and passwords to supply in the authorization response.

It's often necessary to write a technical spike that attempts a connection, and prints the headers on the `401` response just to see what the realm string is. Once the realm is known, `HTTPBasicAuthHandler` can be built. An alternative is to use the developer modes available in some browsers to examine the headers and see the details of the `401` response.

See also

- Proper SSL configuration of a server generally involves using certificates signed by a CA. This involves a certificate chain that starts with the server and includes certificates for the various authorities that issued the certificates.
- Many web service implementations use servers such as GUnicorn or NGINX. These servers generally handle the HTTP and HTTPS issues outside our application. They can also handle complex chains and bundles of certificates.
- For details, see `http://docs.gunicorn.org/en/stable/settings.html#ssl` and also `http://nginx.org/en/docs/http/configuring_https_servers.html`.

11
Application Integration

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

- Finding configuration files
- Using YAML for configuration files
- Using Python for configuration files
- Using class-as-namespace for configuration values
- Designing scripts for composition
- Using logging for control and audit output
- 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
- Controlling complex sequences of steps

Introduction

Python's extensible library gives us rich access to numerous computing resources. This makes Python programs particularly strong at integrating components to create sophisticated composite processing.

In the *Using argparse to get command line input*, *Using cmd for creating command-line applications*, and *Using the OS environment settings* recipes in `Chapter 5`, *User Inputs and Outputs*, specific techniques for creating top-level (main) application scripts were shown. In `Chapter 8`, *Input/Output, Physical Format, Logical Layout*, we looked at file-system input and output. In `Chapter 10`, *Web Services*, we looked at creating servers, which are main applications that receive requests from clients.

All of these examples show some aspects of application programming in Python. There are some additional techniques that are helpful:

- Processing configuration from files. In the *Using argparse to get command line input* recipe in `chapter 5`, *User Inputs and Outputs*, we showed techniques for parsing command line arguments. In the *Using the OS environment settings* recipe, we touched on other kinds of configuration details. 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 YAML 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 YAML for configuration files* recipe.
    - The Properties file format is typical of Java programming, and can be handled in Python without writing too much code. The syntax overlaps with Python scripts.
    - For Python scripts, a file with assignment statements looks a lot like a properties file, and is very easy to process using `compile()` and `exec()` methods. We'll look at this in the *Using Python for configuration files* recipe.
    - Python modules 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 class-as-namespace for configuration* recipe.
- In this chapter, we'll look at ways that we can design applications that can be composed to create larger, more sophisticated composite applications.
- We'll look at the complications that can arise from composite applications and the need to centralize some features like command line parsing.
- We'll extend some of the concepts from `Chapter 6`, *Basics of Classes and Objects*, and `Chapter 7`, *More Advanced Class Design*, and apply the idea of the Command design pattern to Python programs.

Finding configuration files

Many applications will have a hierarchy of configuration options. There could be defaults that are built in to a particular release. There might be server-wide (or cluster-wide) values. There might be user-specific values, or perhaps even configuration files that are local to a specific invocation of a program.

In many cases, these configuration parameters will be written in files so that they are 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`.

How can we support a rich hierarchy of locations for configuration files?

Getting ready

The example will be a web service that provides hands of cards to users. The service is shown in several recipes throughout `Chapter 10`, *Web Services*. We'll gloss over some details of the service so that we can focus on fetching configuration parameters from a variety of file-system locations.

We'll follow the design pattern of the **bash** shell, which looks for configuration files in several places:

1. It starts with the `/etc/profile` file.
2. After reading that file, it looks for one of these files, in this order:
    1. `~/.bash_profile`.
    2. `~/.bash_login`.
    3. `~/.profile`.

In a POSIX-compliant operating system, the shell expands the `~` to be the home directory for the logged-in user. This is defined as the value of the `HOME` environment variable. In general, the Python `pathlib` module can handle this automatically.

There are several ways to keep configuration parameters for a program:

- Using a class definition has the advantage of tremendous flexibility and a relatively simply Python syntax. It can use ordinary inheritance to include default values. It doesn't work as well when there are multiple sources of parameters because there's no trivial way to mutate a class definition.

- For a mapping parameter, we can then use the `ChainMap` collection to search multiple dictionaries, each from a different source.
- For the `SimpleNamespace` instance, the `types` module offers this class, which is mutable and can be updated from multiple sources.
- A `Namespace` instance from the `argparse` module can be handy because it mirrors the options that come from the command-line.

The design pattern from the bash shell uses two separate files. When we include application-wide defaults, there are actually three levels of configuration. This can be done elegantly with a mapping and the `ChainMap` class from the `collections` module.

In later recipes, we'll look at ways to parse and process a configuration file. For the purposes of this recipe, we'll assume that a function, `load_config_file()`, has been defined that will load a configuration map from the contents of the file:

```
def load_config_file(config_file):
    '''Loads a configuration mapping object with contents
    of a given file.

    :param config_file: File-like object that can be read.
    :returns: mapping with configuration parameter values
    '''
    # Details omitted.
```

We'll look at ways to implement this function separately. Variations on the implementation are covered in the *Using YAML for configuration files* and *Using Python for configuration files* recipes of this chapter.

The `pathlib` module can help with this processing. This module provides the `Path` class definition that 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 8`, *Input/Output, Physical Format, Logical Layout*.

Why so many choices?

There's a side-bar topic that sometimes arises when discussing this kind of design—Why have so many choices? Why not specify exactly two places?

The answer depends on the context for the design. When creating an entirely new application, the choices can be limited to exactly two. However, when replacing legacy applications, it's common to have a new location that's better in some ways than the legacy location, but the legacy location still needs to be supported. After several such evolutionary changes, it's common to see a number of alternative locations for files.

Also, because of variations among Linux distributions, it's common to see variations that are typical for one distribution, but atypical for another. And, of course, when dealing with Windows, there will be variant file paths that are unique to that platform too.

How to do it...

1. Import the `Path` class and the `ChainMap` class:

```
from pathlib import Path
from collections import ChainMap
```

2. Define an overall function to get the configuration files:

```
def get_config():
```

3. Create paths for the various locations. These are called pure paths because there's no relationship with the file-system. They start as names of *potential* files:

```
system_path = Path('/etc/profile')
home_path = Path('~').expanduser()
local_paths = [home_path/'.bash_profile',
    home_path/'.bash_login',
    home_path/'.profile']
```

4. Define the application's built-in defaults:

```
configuration_items = [
    dict(
        some_setting = 'Default Value',
        another_setting = 'Another Default',
        some_option = 'Built-In Choice',
    )
]
```

5. Each individual configuration file is a mapping from key to value. The various mapping objects will form a list; this becomes the final `ChainMap` configuration mapping. We'll assemble the list of maps by appending items, and then reverse the order after the files are loaded.
6. If a system-wide configuration file exists, load this file:

```
if system_path.exists():
    with system_path.open() as config_file:
        configuration_items.append(config_file)
```

7. Iterate through other locations looking for a file to load. This loads the first of the files that it finds:

```
for config_path in local_paths:
    if config_path.exists():
        with config_path.open() as config_file:
            configuration_items.append(config_file)
        break
```

We've included the **if-break** pattern to stop after the first file is found. This modifies the loop from the default semantics of For All to mean There Exists. See the Avoiding a potential problem with break statements recipe for more information.

7. Reverse the list and create the final `ChainMap`. The list needs to be reversed so that the local file is searched first, then the system settings, and finally the application default settings:

```
configuration = ChainMap(*reversed(configuration_items))
```

8. Return the final configuration mapping:

```
return configuration
```

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...

One of the most elegant features of any object-oriented language is being able to create simple collections of objects. In this case, the objects are filesystem `Path` objects.

As noted in the *Using pathlib to work with file names* recipe in `Chapter 8`, *Input/Output, Physical Format, Logical Layout*, the `Path` object has a `resolve()` method that can return a concrete `Path` built from a pure `Path`. In this recipe, we used the `exists()` method to determine if a concrete path could be built. The `open()` method, when used to read a file, will resolve the pure `Path` and open the associated file.

In the *Creating dictionaries – inserting and updating* recipe in `Chapter 4`, *Built-in Data Structures – list, set, dict*, 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.

Here's an example of creating a `ChainMap` manually:

```
>>> from collections import ChainMap
>>> config = ChainMap(
...     {'another_setting': 2},
...     {'some_setting': 1},
...     {'some_setting': 'Default Value',
...      'another_setting': 'Another Default',
...      'some_option': 'Built-In Choice'})
```

The `config` object is built from three separate mappings. The first might be details from a local file such as `~/.bash_login`. The second might be system-wide settings from the `/etc/profile` file. The third contains application-wide defaults.

Here's what we see when we query this object's values:

```
>>> config['another_setting']
2
>>> config['some_setting']
1
>>> config['some_option']
'Built-In Choice'
```

The value for any given key is taken from the first instance of that key in the chain of maps. This allows a very simple way to have local values that override system-wide values that override the built-in defaults.

There's more...

In the *Mocking External Resources* recipe in `Chapter 9`, *Testing*, we looked at ways to mock external resources so that we could write a unit test that wouldn't accidentally delete files. A test for the code in this recipe needs to mock the filesystem resources by mocking the `Path` class. Here's how the unit test would look, starting with a high-level outline of the test class:

```
import unittest
from unittest.mock import *

class GIVEN_get_config_WHEN_load_THEN_overrides(unittest.TestCase):
    def setUp(self):

    def runTest(self):
```

This provides a boilerplate structure for a unit test. Mocking a `Path` becomes rather complex because of the number of distinct objects involved. Here's a summary of what kinds of object creations occur:

1. A call to the `Path` class creates a `Path` object. The test process will create two `Path` objects, so we can use the `side_effect` feature to return each of these. We need to be sure that the values are in the correct order based on the code in the unit to be tested:

   ```
   self.mock_path = Mock(
       side_effect = [self.mock_system_path, self.mock_home_path]
   )
   ```

2. For the value of `system_path`, there will be a call to a `Path` object `exists()` method; this will determine if the concrete file exists. There will then be calls to open the file and read the content:

   ```
   self.mock_path = Mock(
       side_effect = [self.mock_system_path, self.mock_home_path]
   )
   ```

3. For the value of `home_path`, there will be a call to the `expanduser()` method to change the ~ to a proper home directory:

   ```
   self.mock_home_path = Mock(
       expanduser = Mock(
           return_value = self.mock_expanded_home_path
       )
   )
   ```

4. The expanded `home_path` is then used with the `/` operator to create the three alternative directories:

```
self.mock_expanded_home_path = MagicMock(
    __truediv__ = Mock(
        side_effect = [self.not_exist, self.exist, self.exist]
    )
)
```

5. For the purposes of unit testing, we've decided that the first path to search doesn't exist. The other two do exist, but we expect that only one of these will be read. The second will be ignored:
    - For mock paths that don't exist, we can use this:

```
self.not_exist = Mock(
    exists = Mock(return_value=False) )
```

    - For the mock paths that exist, we'll have something more complex:

```
self.exist = Mock( exists = Mock(return_value=True), open =
mock_open() )
```

    We have to also handle the processing of the file via the `mock_open()` function in the mock module. This can handle all of the various details of files being used as context managers, something that becomes rather complex. The `with` statement requires `__enter__()` and `__exit__()` methods, which is handled by `mock_open()`.

We have to assemble each of these mock objects in reverse order. This assures that each variable is created before it's used. Here's the entire `setUp()` method showing the objects in the proper order:

```
def setUp(self):
    self.mock_system_path = Mock(
        exists = Mock(return_value=True),
        open = mock_open()
    )
    self.exist = Mock(
        exists = Mock(return_value=True),
        open = mock_open()
    )
    self.not_exist = Mock(
        exists = Mock(return_value=False)
    )
    self.mock_expanded_home_path = MagicMock(
```

```
            __truediv__ = Mock(
                side_effect = [self.not_exist, self.exist, self.exist]
            )
        )
        self.mock_home_path = Mock(
            expanduser = Mock(
                return_value = self.mock_expanded_home_path
            )
        )
        self.mock_path = Mock(
            side_effect = [self.mock_system_path, self.mock_home_path]
        )

        self.mock_load = Mock(
            side_effect = [{'some_setting': 1}, {'another_setting': 2}]
        )
```

In addition to the mocks for `Path` manipulation, we've added one more mock module. The `mock_load` object is a stand-in for the undefined `load_config_file()`. We want to separate this test from the path processing, so the mock object uses the `side_effect` attribute to return two separate values, expecting that it will be invoked exactly twice.

Here are some of the tests that will confirm that the path search works as advertised. Each test starts by applying two patches to create a modified context for testing the `get_config()` function:

```
    def runTest(self):
        with patch('__main__.Path', self.mock_path), \
        patch('__main__.load_config_file', self.mock_load):
            config = get_config()
        # print(config)
        self.assertEqual(2, config['another_setting'])
        self.assertEqual(1, config['some_setting'])
        self.assertEqual('Built-In Choice', config['some_option'])
```

The first use of `patch()` replaces the `Path` class with `self.mock_path`. The second use of `patch()` replaces the `load_config_file()` function with the `self.mock_load` function; this function will return two small configuration documents. In both cases, the context being patched is the current module, with the `__name__` value of `"__main__"`. In the cases where the unit test is in a separate module, then the module under test will be imported, and that module's name will be used.

We can check to see that the `load_config_file()` was called properly by examining the calls to `self.mock_load`. In this case, there should be one for each of the configuration files:

```
        self.mock_load.assert_has_calls(
            [
    call(self.mock_system_path.open.return_value.__enter__.return_value),
                call(self.exist.open.return_value.__enter__.return_value)
            ]
        )
```

We've made sure that the `self.mock_system_path` file is checked first. Note the chain of calls—`Path()` returns a `Path` object. That object's `open()` must return a value that will be used as a context. The `__enter__()` method of a context is an object that will be used by the `load_config_file()` function.

We've made sure that the other path is one for which the `exists()` method returns `True`. Here's the check for the filenames that are built:

```
        self.mock_expanded_home_path.assert_has_calls(
            [call.__truediv__('.bash_profile'),
            call.__truediv__('.bash_login'),
            call.__truediv__('.profile')]
        )
```

The `/` operator is implemented by the `__truediv__()` method. Each of the calls builds a separate the `Path` instance. We can confirm that overall, the `Path` object is used just twice. Once for the literal `'/etc/profile'` and once for the literal `'~'`:

```
        self.mock_path.assert_has_calls(
            [call('/etc/profile'), call('~')]
        )
```

Note that two files answer `True` to the `exists()` method. We expect, however, that only one of those two will be checked. Once this is found, the second file will be ignored. Here's a test that confirms that there's only one check for existence:

```
        self.exist.assert_has_calls( [call.exists()] )
```

Just to be complete, we've also checked that the file that exists will go through the entire context management sequence:

```
        self.exist.open.assert_has_calls(
            [call(), call().__enter__(), call().__exit__(None, None, None)]
        )
```

The first call is for the `self.exist` object's `open()` method. The return value from this is a context that will have the `__enter__()` method executed as well as the `__exit__()` method. In the preceding code, we checked that the return value from `__enter__()` is read to get the configuration file content.

See also

- In the *Using YAML for configuration files* and *Using Python for configuration files* recipes we'll look at ways to implement the `load_config_file()` function.
- In the *Mocking external resources* recipe in `Chapter 9`, *Testing*, we looked at ways to test functions such as this, which interact with external resources.

Using YAML for configuration files

Python offers a variety of ways to package application inputs and configuration files. We'll look at writing files in YAML notation because it's elegant and simple.

How can we represent configuration details in YAML notation?

Getting ready

Python doesn't have a YAML parser built in. We'll need to add the `pyyaml` project to our library using the `pip` package management system. Here's how the installation looks:

```
MacBookPro-SLott:pyweb slott$ pip3.5 install pyyaml
Collecting pyyaml
  Downloading PyYAML-3.11.zip (371kB)
    100% |████████████████████████████████| 378kB
2.5MB/s
Installing collected packages: pyyaml
  Running setup.py install for pyyaml ... done
Successfully installed pyyaml-3.11
```

The elegance of the YAML syntax is that simple indentation is used to show the structure of the document. Here's an example of some settings that we might encode in YAML:

```
query:
  mz:
    - ANZ532
    - AMZ117
    - AMZ080
url:
  scheme: http
  netloc: forecast.weather.gov
  path: /shmrn.php
description: >
  Weather forecast for Offshore including the Bahamas
```

This document 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`. The document contains information about building the URL from a scheme, net location, base path, and several query strings. The `yaml.load()` function can load this YAML document; it will create the following Python structure:

```
{'description': 'Weather forecast for Offshore including the Bahamas\n',
 'query': {'mz': ['ANZ532', 'AMZ117', 'AMZ080']},
 'url': {'netloc': 'forecast.weather.gov',
         'path': 'shmrn.php',
         'scheme': 'http'}}
```

This *dict-of-dict* structure can be used by an application to tailor its operations. In this case, it specifies a sequence of URLs to be queried to assemble a larger weather briefing.

We'll often use the *Finding configuration files* recipe to check a variety of locations for finding 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 previous example, the `load_config_file()` function. Here's the template that needs to be filled in:

```
def load_config_file(config_file) -> dict:
    '''Loads a configuration mapping object with contents
    of a given file.

    :param config_file: File-like object that can be read.
    :returns: mapping with configuration parameter values
    '''
    # Details omitted.
```

How to do it...

1. Import the `yaml` module:

```
import yaml
```

2. Use the `yaml.load()` function to load the YAML-syntax document:

```
def load_config_file(config_file) -> dict:
    '''Loads a configuration mapping object with contents
    of a given file.

    :param config_file: File-like object that can be read.
    :returns: mapping with configuration parameter values
```

```
        '''
        document = yaml.load(config_file)
    return document
```

How it works...

The YAML syntax rules are defined at `http://yaml.org`. The idea of YAML is to provide JSON-like data structures with more flexible, human-friendly syntax. JSON is a special case of the more general YAML syntax.

The trade-off here is that some spaces and line breaks in JSON don't matter—there is visible punctuation to show the structure of the document. In some of the YAML variants, line breaks and indentation determine the structure of the document; the use of white-space means that line breaks will matter with YAML documents.

The principle data structures available in JSON syntax are as follows:

- **Sequence**: `[item, item, ...]`
- **Mapping**: `{key: value, key: value, ...}`
- **Scalar**:
    - String: `"value"`
    - Number: `3.1415926`
    - Literal: `true`, `false`, and `null`

JSON syntax is one style of YAML; it's a called flow style. In this style, the document structure is marked by explicit indicators. The syntax requires `{…}` and `[…]` to show the structure.

The alternative that YAML offers is block style. The document structure is defined by line breaks and indentation. Furthermore, long string scalar values can use plain, quoted, and folded styles of syntax. Here is how the alternative YAML syntax works:

- **Block sequence**: We preface each line of a sequence with a -. This looks like a bullet list, and is easy to read. Here's an example:

```
zoneid:
  - ANZ532
  - AMZ117
  - AMZ080
```

When loaded, this will create a dictionary with a list of strings in Python: `{zoneid: ['ANZ532', 'AMZ117', 'AMZ080']}`.

- **Block mapping**: We can use simple `key: value` syntax to associate a key with a simple scalar. We can use `key:` on a line by itself; the value is indented on the following lines. Here's an example:

```
url:
  scheme: http
  netloc: marine.weather.gov
```

This creates a nested dictionary that looks like this in Python: `{'url': {'scheme': 'http', 'netloc': 'marine.weather.gov'}}`.

We can also use explicit key and value markers, `?` and `:`. This can help when the keys are particularly long strings or more complex objects:

```
? scheme
: http
? netloc
: marine.weather.gov
```

Some more advanced features of YAML will make use of this explicit separation between key and value:

- For short string scalar values, we can leave them plain, and the YAML rules will simply use all the characters with leading and trailing white-space stripped away. The examples all use this kind of assumption for string values.
- Quotes can be used for strings, exactly as they are in JSON.
- For longer strings, YAML introduces the `|` prefix; the lines after this are preserved with all of the spacing and newlines intact.

It also introduces the `>` prefix, which preserves the words as a long string of text—any newlines are treated as single white-space characters. This is common for running text.

In both cases, the indentation determines how much of the document is part of the text.

- In some cases, the value may be ambiguous. For example, a US ZIP code is all digits—`22102`. This should be understood as a string, even though the YAML rules will interpret it as a number. Quotes, of course, can be helpful. To be more explicit, a local tag of `!!str` in front of the value will force a specific data type. `!!str 22102`, for example, assures that the digits will be treated as a string object.

There's more...

There are a number of additional features in YAML that are not present in JSON:

- The comments, which begin with `#` and continue to the end of the line. They can go almost anywhere. JSON doesn't tolerate comments.
- The document start, which is indicated by the `---` line at the start of a new document. This allows a YAML file to contain many individual objects. JSON is limited to a single document per file. The alternative to one document-per-file is somewhat a more complex parsing algorithm. YAML provides an explicit document separator and a very simple parsing interface.
- The YAML file with two separate documents:

```
>>> import yaml
>>> yaml_text = '''
... ---
... id: 1
... text: "Some Words."
... ---
... id: 2
... text: "Different Words."
... '''
>>> document_iterator = iter(yaml.load_all(yaml_text))
>>> document_1 = next(document_iterator)
>>> document_1['id']
1
>>> document_2 = next(document_iterator)
>>> document_2['text']
'Different Words.'
```

- The `yaml_text` value contains two YAML documents, each of which starts with `---`. The `load_all()` function is an iterator that loads the documents one at a time. An application must iterate over the results of this to process each of the documents in the stream.
- Document end. A `...` line is the end of a document.
- Complex keys for mappings; JSON mapping keys are limited to the available scalar types—string, number, `true`, `false`, and `null`. YAML allows mapping keys to be considerably more complex.
- What's important is that Python requires a hash able, immutable object for a mapping key. This means that a complex key must be transformed into an immutable Python object, generally a `tuple`. In order to create a Python-specific object, we need to use a more complex local tag. Here's an example:

```
>>> yaml.load('''
... ? !!python/tuple ["a", "b"]
... : "value"
... ''')
{('a', 'b'): 'value'}
```

- This example uses `?` and `:` to mark the key and value of a mapping. We've done this because the key is a complex object. The key `value` uses a local tag, `!!python/tuple`, to create a tuple instead of the default, which would have been a `list`. The text of the key uses a flow-type YAML value, `["a", "b"]`.
- JSON has no provision for a set. YAML allows us to use the `!!set` tag to create a set instead of a simple sequence. The items in the set are identified by a `?` prefix because they are considered keys of a mapping for which there are no values.
- Note that the `!!set` tag is at the same level of indentation as the values within the set collection. It's indented inside the dictionary key of `data_values`:

```
>>> import yaml
>>> yaml_text = '''
... document:
...     id: 3
...     data_values:
...       !!set
...       ? some
...       ? more
...       ? words
... '''
>>> some_document = yaml.load(yaml_text)
>>> some_document['document']['id']
3
>>> some_document['document']['data_values'] == {'some', 'more',
'words'}
True
```

- The `!!set` local tag modifies the following sequence to become a `set` object instead of the default list object. The resulting set is equal to the expected Python set object, `{'some', 'more', 'words'}`.
- Python mutable object rules will have to be applied to the contents of a set. It's impossible to build a set of `list` objects because list instances don't have hash values. The `!!python/tuple` local tag will have to be used to build a set of tuples.

- We can create a Python list-of-two-tuples sequence, as well which implements ordered mapping. The `yaml` module doesn't readily create an `OrderedDict` for us:

```
>>> import yaml
>>> yaml_text = '''
... !!omap
... - key1: string value
... - numerator: 355
... - denominator: 113
... '''
>>> yaml.load(yaml_text)
[('key1', 'string value'), ('numerator', 355), ('denominator', 113)]
```

- Note that it's difficult to take the next step and create an `OrderedDict` from this without specifying a large number of details. Here's the YAML to create an instance of `OrderedDict`.

```
!!python/object/apply:collections.OrderedDict
args:
    -   !!omap
        -   key1: string value
        -   numerator: 355
        -   denominator: 113
```

- The `args` keyword is required to support the `!!python/object/apply` tag. There's only one positional argument, and it's a YAML `!!omap` built from a sequence of keys and values.
- Python objects of almost any class can be built using YAML local tags. Any class with a simple `__init__()` method can be built from a YAML serialization.

Here's a simple class definition:

```
class Card:
    def __init__(self, rank, suit):
        self.rank = rank
        self.suit = suit
    def __repr__(self):
        return "{rank} {suit}".format_map(vars(self))
```

We've defined a class with two positional attributes. Here's the YAML description of this object:

```
!!python/object/apply:__main__.Card
kwds:
    rank: 7
    suit: ♣
```

We've used the `kwds` key to provide two keyword-based argument values to the `Card` constructor function. The Unicode ♣ character works well because YAML files are text written using UTF-8 encoding.

- In addition to local tags, which start with `!!`, YAML also supports tags that are URIs using the `tag:` scheme. This allows URI-based type specifications that are globally unique. This can make YAML documents easier to process in a variety of contexts.

  A tag includes an authority name, a date, and specific details in the form of a `/`-delimited path. For example, a tag might look like this—`!<tag:www.someapp.com,2016:rules/rule1>`.

See also

- See the *Finding configuration files* recipe to see how to search multiple file-system 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

Using Python for configuration files

Python offers a variety of ways that we can package application inputs and configuration files. We'll look at writing files in Python notation because it's elegant and simple.

A number of packages use assignment statements in a separate module for providing configuration parameters. The Flask project in particular, can do this. We looked at Flask in the *Using the Flask framework for RESTful APIs* recipe and a number of related recipes in `Chapter 10`, *Web Services*.

How can we represent configuration details in Python module notation?

Getting ready

Python assignment statement notation is particularly elegant. It's quite simple, easy to read, and extremely flexible. If we use assignment statements, we can import the configuration details from a separate module. This could have a name like `settings.py` to shows that it's focused on configuration parameters.

Because Python treats each imported module as a global **Singleton** object, we can have several parts of an application all use the `import settings` statement to get a consistent view of the current, global application configuration parameters.

In some cases, however, we might want to choose one of several alternative settings files. In this case, we want to load a file using a technique that's more flexible than the `import` statement.

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']}
url = {
  'scheme': 'http',
  'netloc': 'forecast.weather.gov',
  'path': '/shmrn.php'
}
```

This is Python syntax. The parameters include two variables, `query` and `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 finding 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 previous example, the `load_config_file()` function. Here's the template that needs to be filled in:

```
def load_config_file(config_file) -> dict:
    '''Loads a configuration mapping object with contents
    of a given file.

    :param config_file: File-like object that can be read.
    :returns: mapping with configuration parameter values
    '''
```

```
    # Details omitted.
```

How to do it...

This code replaces the `# Details omitted.` line in the `load_config_file()` function:

1. Compile the code in the configuration file using the built-in `compile()` function. 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:

```
code = compile(config_file.read(), config_file.name, 'exec')
```

2. In rare cases where the code doesn't come from a file, the general practice is to provide a name such as `<string>` instead of a 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 where new variables will be created:

```
locals = {}
exec(code, {'__builtins__':__builtins__}, locals)
return locals
```

4. When code is executed at the very top level of a script file—often inside the `if __name__ == "__main__"` condition—it executes in a context where globals and locals are the same. When code is executed inside a function, method, or class definition, the locals for that context are separate from the globals.
5. By creating a separate `locals` object, we've made sure that the imported statements don't make unexpected changes to any other global variables.

How it works...

The details of the Python language; syntax and semantics are embodied in the `compile()` and `exec()` functions. When we launch a script, the process is essentially this:

1. Read the text. Compile it with the `compile()` function to create a code object.
2. Use the `exec()` function to execute that code object.

The `__pycache__` directory holds code objects, and saves recompiling text files that haven't changed. It doesn't have a material impact on the processing.

The `exec()` function reflects the way Python handles global and local variables. There are two namespaces provided to this function. These are visible to a script that's running via the `globals()` and `locals()` functions.

We provided two distinct dictionaries:

- A dictionary of global objects. These variables can be accessed via the `global` statement. The most common use is to provide access to the imported modules, which are always global. The `__builtins__` module, for example, is often provided. In some cases, other modules should be added.
- The dictionary provided for the locals is updated by each assignment statement. This local dictionary allows us to capture the variables created within the `settings` module.

There's more...

This recipe builds a configuration file that can be entirely a sequence of `name = value` assignments. This statement is supported directly by the Python assignment statement syntax.

Additionally, the full spectrum of Python programming is available. There are a number of engineering trade-offs that must be made.

Any statement can be used in the configuration file. However, this can lead to complexity. If the processing becomes too complex, the file ceases to be configuration, and becomes a first-class part of the application. Very complex features should be done by modifying the application programming, not hacking around with the configuration settings. Since Python applications include the source, this is relatively easy to do.

In addition to the simple assignment statement, it can be sensible to use `if` statements to handle alternatives. A file might provide a section for unique features of a specific run-time environment. For example, the `platform` package can be used to isolate features.

Something like this might be included:

```
import platform
if platform.system() == 'Windows':
    tmp = Path(r"C:\TEMP")
else:
    tmp = Path("/tmp")
```

For this to work, the globals should include `platform` and `Path`. This a reasonable extension above and beyond `__builtins__`.

It can also be sensible to do some processing simply to make a number of related settings easier to organize. For example, an application might have a number of related files. It can be helpful to write a configuration file like this:

```
base = Path('/var/app/')
log = base/'log'
out = base/'out'
```

The values of `log` and `out` are used by the application. The value of `base` is only used to assure that the other two locations are placed in the same directory.

This leads to the following variation on the `load_config_file()` function shown earlier. This version includes some additional modules and global classes:

```
from pathlib import Path
import platform
def load_config_file_path(config_file) -> dict:
    code = compile(config_file.read(), config_file.name, 'exec')
    globals = {'__builtins__': __builtins__,
        'Path': Path, 'platform': platform}
    locals = {}
    exec(code, globals, locals)
    return locals
```

Including `Path` and `platform` means that a configuration file can be written without the overhead of `import` statements. This can make the settings simpler to prepare and maintain.

See also

- See the *Finding configuration files* recipe to learn how to search multiple file-system locations for a configuration file.

Using class-as-namespace for configuration

Python offers a variety of ways for packaging application inputs and configuration files. We'll look at writing files in Python notation because it's elegant and simple.

A number of projects allow the use of a class definition for providing configuration parameters. The use of a class hierarchy means that inheritance techniques can be used to simplify organization of parameters. The Flask package in particular, can do this. We looked at Flask in the recipe, *Using the Flask framework for RESTful APIs*, and a number of related recipes.

How can we represent configuration details in Python class notation?

Getting ready

Python notation for defining the attributes of a class is particularly elegant. It's quite simple, easy to read, and reasonably flexible. We can, with 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:
    """
    Weather forecast for Offshore including the Bahamas
    """
    query = {'mz': ['ANZ532', 'AMZ117', 'AMZ080']}
    url = {
      'scheme': 'http',
      'netloc': 'forecast.weather.gov',
      'path': '/shmrn.php'
    }
```

We can create this `Configuration` class in a single `settings` module. To use the configuration, the main application will do this:

```
from settings import Configuration
```

This uses a fixed file with a fixed class name. In spite of the apparent lack of flexibility, this can often be more useful than other alternatives. We have two additional ways to support complex configuration files:

- We can use the `PYTHONPATH` environment variable to list a number of locations for a configuration module

- Use multiple inheritance and mix ins to combine defaults, system-wide settings, and localized settings into a configuration class definition

These techniques can be helpful because the configuration file locations simply follow Python's rules for finding modules. We don't need to implement our own search for the configuration files.

In this recipe, we'll build the missing part of the previous example, the `load_config_file()` function. Here's the template that needs to be filled in:

```
def load_config_file(config_file) -> dict:
    '''Loads a configuration mapping object with contents
    of a given file.

    :param config_file: File-like object that can be read.
    :returns: mapping with configuration parameter values
    '''
    # Details omitted.
```

How to do it...

This code replaces the `# Details omitted.` line in the `load_config_file()` function:

1. Compile the code in the given file using the built-in `compile()` function. 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:

   ```
   code = compile(config_file.read(), config_file.name, 'exec')
   ```

2. 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 = {}
   exec(code, globals, locals)
   return locals['Configuration']
   ```

3. This returns only the defined `Configuration` class from the locals which are set by the executed module. Any other variables will be ignored.

How it works...

The details of the Python language—syntax and semantics—are embodied in the `compile()` and `exec()` functions. The `exec()` function reflects the way Python handles global and local variables. There are two namespaces provided to this function. The global `namespace` instance includes `__builtins__` plus a class and module that might be used in the file.

The local variable namespace will have the new class created in it. This namespace has a `__dict__` attribute that makes it accessible via dictionary methods. Because of this, we can then extract the class by name. The function returns the class object for use throughout the application.

We can put any kind of object into the attributes of a class. Our example showed mapping objects. There's no limitation on what can be done when creating attributes at the class level.

We can have complex calculations within the `class` statement. We can use this to create attributes which are derived from other attributes. We can execute any kind of statement, including `if` statements and `for` statements to create attribute values.

There's more...

Using a class definition means that we can 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
    """
    url = {
      'scheme': 'http',
      'netloc': 'forecast.weather.gov',
      'path': '/shmrn.php'
    }

class Bahamas(Configuration):
    """
```

```
        Weather forecast for Offshore including the Bahamas
        """
        query = {'mz': ['AMZ117', 'AMZ080']}

    class Cheaspeake(Configuration):
        """
        Weather for Cheaspeake Bay
        """
        query = {'mz': ['ANZ532']}
```

This means that 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 is executed like this:

```
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_file()` function.

In order to pick one of the available classes, we'll provide an additional parameter with the class name:

```
import importlib
def load_config_module(name):
    module_name, _, class_name = name.rpartition('.')
    settings_module = importlib.import_module(module_name)
    return vars(settings_module)[class_name]
```

Rather than manually compile and execute the module, we've used the higher-level `importlib` module. This module implements the `import` statement semantics. The requested module is imported; compiled and executed—and the resulting module object assigned to the variable name `settings_module`.

We can then look inside the module's variables and pick out the class that was requested. The `vars()` built-in function will extract the internal dictionary from a module, class, or even the local variables.

Now we can use this function as follows:

```
>>> configuration = load_config_module('settings.Chesapeake')
>>> configuration.__doc__.strip()
'Weather for Cheaspeake Bay'
>>> configuration.query
{'mz': ['ANZ532']}
>>> configuration.url['netloc']
```

```
'forecast.weather.gov'
```

We've located the `Chesapeake` configuration class in the `settings` module.

Configuration representation

One consequence of using a class like this is that the default display for a class isn't too informative. When we try to print the configuration, it looks like this:

```
>>> print(configuration)
<class 'settings.Chesapeake'>
```

This is nearly useless. It provides one nugget of information, but that's not nearly enough for debugging.

We can use the `vars()` function to see more details. However, this shows local variables, not inherited variables:

```
>>> pprint(vars(configuration))
mappingproxy({'__doc__': '\\n    Weather for Cheaspeake Bay\\n    ',
              '__module__': 'settings',
              'query': {'mz': ['ANZ532']}})
```

This is better, but still incomplete.

In order to see all of the settings, we need something a little more sophisticated. Interestingly, we can't simply define `__repr__()` for this class. A method defined in a class will apply to the instances of this class, not the class itself.

Each class object we create is an instance of the built-in `type` class. We can, using a meta-class, tweak the way the `type` class behaves, and implement a slightly nicer `__repr__()` method, which looks through all parent classes for attributes.

We'll extend the built-in type with a `__repr__` that does a somewhat better job at displaying the working configuration:

```
class ConfigMetaclass(type):
    def __repr__(self):
        name = (super().__name__ + '('
            + ', '.join(b.__name__ for b in super().__bases__) + ')')
        base_values = {n:v
            for base in reversed(super().__mro__)
                for n, v in vars(base).items()
                    if not n.startswith('_')}
        values_text = ['    {0} = {1!r}'.format(name, value)
            for name, value in base_values.items()]
```

```
        return '\n'.join(["class {}:".format(name)] + values_text)
```

The class name is available from the superclass, `type`, as the `__name__` attribute. The names of the base classes are included as well, to show the inheritance hierarchy for this configuration class.

The `base_values` are built from the attributes of all of the base classes. Each class is examined in reverse order of the **Method Resolution Order** (**MRO**). Loading all of the attribute values in reverse MRO means that all of the defaults are loaded first, then overridden with subclass values.

The names that lack the _ prefix are included. Names with the _ prefix are quietly ignored.

The resulting values are used to create a text representation that resembles a class definition. It's not the original class source code; it's the net effect of the original class definition.

Here's a `Configuration` class hierarchy that uses this meta-class. The base class, `Configuration`, incorporates the meta-class, and provides default definitions. The subclass extends those definitions with values that are unique to a particular environment or context:

```
class Configuration(metaclass=ConfigMetaclass):
    unchanged = 'default'
    override = 'default'
    feature_override = 'default'
    feature = 'default'

class Customized(Configuration):
    override = 'customized'
    feature_override = 'customized'
```

We can leverage all of the power of Python's multiple inheritance to build `Configuration` class definitions. This can provide the ability to combine details on separate features into a single configuration object.

See also

- We'll look at class definitions in `Chapter 6`, *Basics of Classes and Objects*, and `Chapter 7`, *More Advanced Class Design*

Designing scripts for composition

Many large applications are actually amalgamations of multiple, smaller applications. In enterprise terminology, they are often called application systems comprising individual command-line application programs.

Some large, complex applications include a number of commands. For example, the Git application has numerous individual commands, such as `git pull`, `git commit`, and `git push`. These can also be seen as separate applications that are part of the overall Git system of applications.

An application might start as a collection of separate Python script files. At some point during its evolution, it can become necessary to refactor the scripts to combine features and create new, composite scripts from older disjoint scripts. The other path is also possible, a large application might be decomposed and refactored into a new organization.

How can we design a script so that future combinations and refactoring is made as simple as possible?

Getting ready

We need to distinguish between several design features of a Python script:

- We've seen several aspects of gathering input:
    - Getting highly dynamic input from a command-line interface and environment variables. See the *Using argparse to get command-line input* recipe in `Chapter 5`, *User Inputs and Outputs*.
    - Getting slow for changing configuration options from files. See the recipes, *Finding configuration files*, *Using YAML for configuration files*, and *Using Python for configuration files*.
    - For reading any input files, see the *Reading delimited files with the CSV module*, *Reading complex formats using regular expressions*, *Reading JSON documents*, *Reading XML documents*, and *Reading HTML documents* recipes in `Chapter 8`, *Input/Output, Physical Format, and Logical Layout*.
- There are several aspects to producing output:
    - Creating logs and offering other features that support audit, control, and monitoring. We'll look at some of this in the *Using logging for control and audit output* recipe.

    - Creating the principle output of the application. This might be printed or written to an output file, using the same library modules used to parse inputs.
- The real work of the application. These are the essential features, disentangled from the various input parsing and output formatting considerations. This algorithm works exclusively with Python data structures.

This *separation of concerns* suggests that any application, no matter how simple, should be designed as several separate functions. These should be combined into a complete script. This lets us separate the input and output from the core processing. The processing is the part we'll often want to reuse. The input and output formats should be something that can easily be changed.

As a concrete example, we'll look at a simple application that creates sequences of dice rolls. Each sequence will follow the rules of the game of craps. Here are the rules:

1. The first roll of two dice is the *come out* roll:
    1. A roll of two, three, or twelve is an immediate loss. The sequence has a single value, for example, `[(1, 1)]`.
    2. A roll of seven or eleven is an immediate win. This sequence also has a single value, for example, `[(3, 4)]`.
2. Any other number establishes a point. The sequence starts with the point value and continues until either a seven or the point value is rolled:
    1. A final seven is a loss, for example, `[(3, 1), (3, 2), (1, 1), (5, 6), (4, 3)]`.
    2. A final match of the original point value is a win. There will be a minimum of two rolls. There's no upper bound on the length of a game, for example, `[(3, 1), (3, 2), (1, 1), (5, 6), (1, 3)]`.

The output is a sequence of items with different structures. Some will be short lists. Some will be long lists. This is an ideal place for using YAML format files.

This output can be controlled by two inputs—how many samples to create, and whether or not to seed the random number generator. For testing purposes, it can help to have a fixed seed.

How to do it...

1. Design all of the output display into two broad areas:
    1. Functions (or classes) that do no processing, but display result objects.
    2. Logging may be for debugging, monitoring, audit, or some other control. This is a cross-cutting concern that will be embedded in the rest of the application.

   For this example, there are two outputs—the sequence of sequences, and some additional information to confirm that processing worked properly. A count of each number rolled is a handy way to establish that the simulated dice were fair.

   The sequence of rolls needs to be written to a file. This suggests that the `write_rolls()` function is given an iterator as a parameter. Here's a function that iterates and dumps values to a file in YAML notation:

   ```
   def write_rolls(output_path, roll_iterator):
       face_count = Counter()
       with output_path.open('w') as output_file:
           for roll in roll_iterator:
               output_file.write(
                   yaml.dump(
                       roll,
                       default_flow_style=True,
                       explicit_start=True))
               for dice in roll:
                   face_count[sum(dice)] += 1
       return face_count
   ```

   The monitoring and control output should display the input parameters used to control the processing. It should also provide the counts that show that the dice were fair:

   ```
   def summarize(configuration, counts):
       print(configuration)
       print(counts)
   ```

2. Design (or refactor) the essential processing of the application to look like a single function:
    1. All inputs are parameters.
    2. All outputs are produced by `return` or `yield`. Use `return` to create the single result. Use `yield` to generate an sequence iterate for multiple results.

In this example, we can easily make the core feature a function that emits an sequence iterate of values. The output function can use this iterator:

```
def roll_iter(total_games, seed=None):
    random.seed(seed)
    for i in range(total_games):
        sequence = craps_game()
        yield sequence
```

This function relies on a `craps_game()` function to generate the requested number of samples. Each sample is a full game, showing all of the dice rolls. This function provides the `face_count` counter to this lower-level function to accumulate some totals to confirm that everything worked properly.

The `craps_game()` function implements the craps game rules to emit a single sequence of one or more rolls. This comprises all the rolls in a single game. We'll look at this `craps_game()` function later.

3. Refactor all of the input gathering into a function (or class) that gathers the various input sources. This can include environment variables, command-line arguments, and configuration files. It may also include the names of multiple input files:

```
def get_options(argv):
    parser = argparse.ArgumentParser()
    parser.add_argument('-s', '--samples', type=int)
    parser.add_argument('-o', '--output')
    options = parser.parse_args(argv)

    options.output_path = Path(options.output)

    if "RANDOMSEED" in os.environ:
        seed_text = os.environ["RANDOMSEED"]
        try:
            options.seed = int(seed_text)
        except ValueError:
            sys.exit("RANDOMSEED={0!r} invalid".format(seed_text))
    else:
        options.seed = None
    return options
```

This function gathers command-line arguments. It also checks the `os.environ` collection of environment variables.

The argument parser will handle the details of parsing the `--samples` and `--output` options. We can leverage additional features of `argparse` to better validate the argument values.

The value of `output_path` is created from the the value of the `--output` option. Similarly, the value of the `RANDOMSEED` environment variable is validated and placed into the `options` namespace. This use of the `options` object keeps all of the various arguments in one place.

4. Write the final `main()` function, which incorporates the three previous elements, to create the final, overall script:

```
    def main():
        options = get_options(sys.argv[1:])
        face_count = write_rolls(options.output_path,
roll_iter(options.samples, options.seed))
        summarize(options, face_count)
```

This brings the various aspects of the application together. It parses the command-line and environment options. It creates a control total counter.

The `roll_iter()` function is the core processing. It takes the various options, and it emits a sequence of rolls.

The primary output from `roll_iter()` method is collected by `write_rolls()` and written to the given output path. The control output is written by a separate function, so that we can change the summary without an impact on the primary output.

How it works...

The output looks like this:

```
slott$ python3 ch13_r05.py --samples 10 --output=x.yaml
Namespace(output='x.yaml', output_path=PosixPath('x.yaml'), samples=10,
seed=None)
Counter({5: 7, 6: 7, 7: 7, 8: 5, 4: 4, 9: 4, 11: 3, 10: 1, 12: 1})
slott$ more x.yaml
--- [[5, 4], [3, 4]]
--- [[3, 5], [1, 3], [1, 4], [5, 3]]
--- [[3, 2], [2, 4], [6, 5], [1, 6]]
--- [[2, 4], [3, 6], [5, 2]]
--- [[1, 6]]
--- [[1, 3], [4, 1], [1, 4], [5, 6], [6, 5], [1, 5], [2, 6], [3, 4]]
--- [[3, 3], [3, 4]]
```

```
--- [[3, 5], [4, 1], [4, 2], [3, 1], [1, 4], [2, 3], [2, 6]]
--- [[2, 2], [1, 5], [5, 5], [1, 5], [6, 6], [4, 3]]
--- [[4, 5], [6, 3]]
```

The command line requested ten samples, and specified an output file of `x.yaml`. The control output is a simple dump of the options. It shows the values for the parameters plus the additional values set in the `options` object.

The control output includes the counts from ten samples. This provides some confidence that values such as six, seven, and eight occur more often. It shows that values such as three and twelve occur less frequently.

The central premise here is the separation of concerns. There are three distinct aspects to the processing:

- **Inputs**: The parameters from the command line, and environment variables are gathered by a single function, `get_options()`. This function can grab inputs from a variety of sources, including configuration files.
- **Outputs**: The primary output was handled by the `write_rolls()` function. The other control output was handled by accumulating totals in a `Counter` object and then dumping this output at the end.
- **Process**: The application's essential processing is factored into the `roll_iter()` function. This function can be reused in a variety of contexts.

The goal of this design is to separate the `roll_iter()` function from the surrounding application details. Another application might have different command-line arguments, or different output formats, but reuse the essential algorithm.

For example, there may be a second application that performs some statistical analyses on the sequences of rolls. This might include a count of rolls, and the final outcome of win or lose. We can assume that these two applications are `generator.py` (shown previously) and `overview_stats.py`.

After using these two applications to create rolls, and summarize them, the users may determine that it would be advantageous to combine the roll creation and the statistical overview into a single application. Because the various aspects of each application have been separated, it becomes relatively easy to rearrange the features and create a new application. We can now build a new application that will start with the following two imports:

```
from generator import roll_iter, craps_rules
from stats_overview import summarize
```

This new application can be built without any changes to the other two applications. This leaves the original applications untouched by the introduction of new features.

More importantly, the new application did not involve any copying or pasting of code. The new application imports working software—changes made to fix one application will also fix latent bugs in other applications.

Reuse via copy and paste creates technical debt. Avoid copying and pasting the code.

When we try to copy code from one application to make a new application, we create a confusing situation. Any changes made to one copy won't magically fix latent bugs in the other copy. When changes are made to one copy, and the other copy is not kept up-to-date, this is one kind of *code rot*.

There's more...

In the previous section, we skipped over the details of the `craps_rules()` function. This function creates a sequence of dice rolls that comprise a single game of *Craps*. It can vary from a single roll to a sequence of indefinite length. About 98% of the games will be thirteen or fewer throws of the dice.

The rules depend on the total of two dice. The data captured include the two separate faces of the dice. In order to support these details, it's helpful to have a `namedtuple` instance that has these two related properties:

```
Roll = namedtuple('Roll', ('faces', 'total'))
def roll(n=2):
    faces = list(random.randint(1, 6) for _ in range(n))
    total = sum(faces)
    return Roll(faces, total)
```

This `roll()` function creates a `namedtuple` instance with a sequence that shows the faces of the dice, as well as the total of the dice. The `craps_game()` function will generate enough rules to return one complete game:

```
def craps_game():
    come_out = roll()
    if come_out.total in [2, 3, 12]:
        return [come_out.faces]
    elif come_out.total in [7, 11]:
```

```
            return [come_out.faces]
        elif come_out.total in [4, 5, 6, 8, 9, 10]:
            sequence = [come_out.faces]
            next = roll()
            while next.total not in [7, come_out.total]:
                sequence.append(next.faces)
                next = roll()
            sequence.append(next.faces)
            return sequence
        else:
            raise Exception("Horrifying Logic Bug")
```

The `craps_game()` function implements the rules for craps. If the first roll is two, three, or twelve, the sequence only has a single value, and the game is a loss. If the first roll is seven or eleven, the sequence also has only a single value, and the game is a win. The remaining values establish a point. The sequence of rolls starts with the point value. The sequence continues until it's ended by seven or the point value.

Designing as a class hierarchy

The close relationship between `roll_iter()`, `roll()`, and `craps_game()` methods suggests that it might be better to encapsulate these functions into a single class definition. Here's a class that has all of these features bundled together:

```
class CrapsSimulator:
    def __init__(self, seed=None):
        self.rng = random.Random(seed)
        self.faces = None
        self.total = None

    def roll(self, n=2):
        self.faces = list(self.rng.randint(1, 6) for _ in range(n))
        self.total = sum(self._faces)

    def craps_game(sel):
        self.roll()
        if self.total in [2, 3, 12]:
            return [self.faces]
        elif self.total in [7, 11]:
            return [self.faces]
        elif self.total in [4, 5, 6, 8, 9, 10]:
            point, sequence = self.total, [self.faces]
            self.roll()
            while self.total not in [7, point]:
                sequence.append(self.faces)
                self.roll()
```

```
            sequence.append(self.faces)
            return sequence
        else:
            raise Exception("Horrifying Logic Bug")

    def roll_iter(total_games):
        for i in range(total_games):
            sequence = self.craps_game()
            yield sequence
```

This class includes an initialization of the simulator to include its own random number generator. It will either use the given seed value, or the internal algorithm will pick seed value. The `roll()` method will set the `self.total` and `self.faces` instance variables.

The `craps_game()` generates one sequence of rolls for one game of craps. It uses the `roll()` method and the two instance variables, `self.total` and `self.faces`, to track the state of the dice.

The `roll_iter()` method generates the sequence of games. Note that the signature of this method is not exactly like the preceding `roll_iter()` function. This class separates random number seeding from the game creation algorithm.

Rewriting `main()` to use the `CrapsSimulator` class is left as an exercise for the reader. Since the method names are similar to the original function names, the refactoring should not be terribly complex.

See also

- See the *Using argparse to get command-line input* recipe in `Chapter 5`, *User Inputs and Outputs*, for background on using `argparse` to get inputs from a user
- See the *Finding configuration files* recipe for a way to track down configuration files
- The *Using logging for control and audit output* recipe looks at logging
- In the *Combining two applications into one* recipe, we'll look at ways to combine applications that follow this design pattern

Using logging for control and audit output

In the *Designing scripts for composition* recipe, we examined three aspects of an application:

- Gathering input
- Producing output
- The essential processing that connects input and output

There are several different kinds of output that applications produce:

- The principle output that helps a user make a decision or take action
- Control information that confirms that the program worked completely and correctly
- Audit summaries that are used to track the history of state changes in 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 are conflated into a single stream.

The OS provides 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()` method writes to the `sys.stdout` file. We can change this and write the control, audit, and error messages to `sys.stderr`. This is an important step in the right direction.

Python offers the `logging` package, which can be used to direct the ancillary output to a separate file. It can also be used to format and filter that additional output.

How can we use logging properly?

Getting ready

In the *Designing scripts for composition* recipe, we looked at an application that produced a YAML file with the raw output of a simulation in it. In this recipe, we'll look at an application that consumes that raw data and produces some statistical summaries. We'll call this application `overview_stats.py`.

Following the design pattern of separating the input, output, and processing, we'll have an application `main()` that looks something like this:

```
def main():
    options = get_options(sys.argv[1:])
    if options.output is not None:
        report_path = Path(options.output)
        with report_path.open('w') as result_file:
            process_all_files(result_file, options.file)
    else:
        process_all_files(sys.stdout, options.file)
```

This function will get the options from various sources. If an output file is named, it will create the output file using a `with` statement context. This function will then process all of the command-line argument files as input from which statistics are gathered.

If no output file name is provided, this function will write to the `sys.stdout` file. This will display output that can be redirected using the OS shell's > operator to create a file.

The `main()` function relies on a `process_all_files()` function. The `process_all_files()` function will iterate through each of the argument files, and gather statistics from that file. Here's what that function looks like:

```
def process_all_files(result_file, file_names):
    for source_path in (Path(n) for n in file_names):
        with source_path.open() as source_file:
            game_iter = yaml.load_all(source_file)
            statistics = gather_stats(game_iter)
            result_file.write(
                yaml.dump(dict(statistics), explicit_start=True)
            )
```

The `process_all_files()` function applies `gather_stats()` to each file in the `file_names` iterable. The resulting collection is written to the given `result_file`.

The function shown here conflates processing and output in a design that is not ideal. We'll address this design flaw in the *Combining two applications into one* recipe.

The essential processing is in the `gather_stats()` function. Given a path to a file, this will read and summarize the games in that file. The resulting summary object can then be written as part of the overall display or, in this case, appended to a sequence of YAML-format summaries:

```
def gather_stats(game_iter):
    counts = Counter()
    for game in game_iter:
        if len(game) == 1 and sum(game[0]) in (2, 3, 12):
            outcome = "loss"
        elif len(game) == 1 and sum(game[0]) in (7, 11):
            outcome = "win"
        elif len(game) > 1 and sum(game[-1]) == 7:
            outcome = "loss"
        elif len(game) > 1 and sum(game[0]) == sum(game[-1]):
            outcome = "win"
        else:
            raise Exception("Wait, What?")
        event = (outcome, len(game))
        counts[event] += 1
    return counts
```

This function determines which of the four game termination rules were applied to the sequence of dice rolls. It starts by opening the given source file, and using the `load_all()` function to iterate through all of the YAML documents. Each document is a single game, represented as a sequence of dice pairs.

This function uses the first (and last) rolls to determine the overall outcome of the game. There are four rules, which should enumerate all possible logical combinations of events. In the event, if there is some error in our reasoning, an exception will get raised to alert us to a special case that didn't fit the design in some way.

The game is reduced to a single event with an outcome and a length. These are accumulated into a `Counter` object. The outcome and length of a game are the two values we're computing. These are a stand-in for more complex or sophisticated statistical analyses that are possible.

We've carefully segregated almost all file-related considerations from this function. The `gather_stats()` function will work with any iterable source of game data.

Here's the output from this application. It's not very pretty; it's a YAML document that can be used for further processing:

```
slott$ python3 ch13_r06.py x.yaml
---
? !!python/tuple [loss, 2]
: 2
? !!python/tuple [loss, 3]
: 1
? !!python/tuple [loss, 4]
: 1
```

```
? !!python/tuple [loss, 6]
: 1
? !!python/tuple [loss, 8]
: 1
? !!python/tuple [win, 1]
: 1
? !!python/tuple [win, 2]
: 1
? !!python/tuple [win, 4]
: 1
? !!python/tuple [win, 7]
: 1
```

We'll need to insert logging features into all of these functions to show which file is being read, and any errors or problems with processing the file.

Furthermore, we're going to create two logs. One will have details, and the other will have a minimal summary of files that are created. The first log can go to `sys.stderr`, which will be displayed at the console when the program runs. The other log will be appended to a long-term `log` file to cover all uses of the application.

One approach to having separate needs is to create two loggers, each with a different intent. The two loggers will also have dramatically different configurations. Another approach is to create a single logger, and use a `Filter` object to distinguish content intended for each logger. We'll focus on creating separate loggers because it's easier to develop and easier to unit test.

Each logger has a variety of methods reflecting the severity of the message. 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.
- **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 work, but they should be replaced. These messages are often displayed.
- **ERROR**: Processing is invalid and the output is incorrect or incomplete. In the cases 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.

The method names are similar to the severity levels. We use `logging.info()` to write an INFO-level message.

How to do it...

1. We'll start by implementing basic logging features into the existing functions. This means that we'll need the `logging` module:

   ```
   import logging
   ```

   The rest of the application will use a number of other packages:

   ```
   import argparse
   import sys
   from pathlib import Path
   from collections import Counter
   import yaml
   ```

2. We'll create two logger objects as module globals. The creating functions can go anywhere in the script that creates global variables. One location is to put these early, after the `import` statements. Another common choice is near the end, but outside any `__name__ == "__main__"` script processing. These variables must always be created, even if the module is imported as a library.

   Loggers have hierarchical names. We'll name the loggers using the application name and a suffix with the content. The `overview_stats.detail` logger will have processing details. The `overview_stats.write` logger will identify the files read and the files written; this parallels the idea of an audit log because the file writes track state changes in the collection of output files:

   ```
   detail_log = logging.getLogger("overview_stats.detail")
   write_log = logging.getLogger("overview_stats.write")
   ```

   We don't need to configure these loggers at this time. If we do nothing more, the two logger objects will silently accept individual log entries, but won't do anything further with the data.

3. We'll rewrite the `main()` function to summarize the two aspects of the processing. This will use the `write_log` logger object to show when a new file is created:

   ```
   def main():
       options = get_options(sys.argv[1:])
   ```

```
    if options.output is not None:
        report_path = Path(options.output)
        with report_path.open('w') as result_file:
            process_all_files(result_file, options.file)
        write_log.info("wrote {}".format(report_path))
    else:
        process_all_files(sys.stdout, options.file)
```

We've added the `write_log.info("wrote {}".format(result_path))` line to put an information message into the log for files that have been written.

4. We'll rewrite the `process_all_files()` function to provide a note when a file is read:

```
def process_all_files(result_file, file_names):
    for source_path in (Path(n) for n in file_names):
        detail_log.info("read {}".format(source_path))
        with source_path.open() as source_file:
            game_iter = yaml.load_all(source_file)
            statistics = gather_stats(game_iter)
        result_file.write(
            yaml.dump(dict(statistics), explicit_start=True)
        )
```

We've added the `detail_log.info("read {}".format(source_path))` line to put information messages in the detail log for every file that's read.

5. The `gather_stats()` function can have a log line added to it to track normal operations. Additionally, we've added a log entry for the logic error:

```
def gather_stats(game_iter):
    counts = Counter()
    for game in game_iter:
        if len(game) == 1 and sum(game[0]) in (2, 3, 12):
            outcome = "loss"
        elif len(game) == 1 and sum(game[0]) in (7, 11):
            outcome = "win"
        elif len(game) > 1 and sum(game[-1]) == 7:
            outcome = "loss"
        elif len(game) > 1 and sum(game[0]) == sum(game[-1]):
            outcome = "win"
        else:
            detail_log.error("problem with {}".format(game))
            raise Exception("Wait, What?")
        event = (outcome, len(game))
        detail_log.debug("game {} -> event {}".format(game, event))
        counts[event] += 1
```

```
    return counts
```

The `detail_log` logger is used to collect debugging information. If we set the overall logging level to include debug messages, we'll see this additional output.

6. The `get_options()` function will also have a debugging line written. This can help diagnose problems by displaying the options into the log:

```
def get_options(argv):
    parser = argparse.ArgumentParser()
    parser.add_argument('file', nargs='*')
    parser.add_argument('-o', '--output')
    options = parser.parse_args(argv)
    detail_log.debug("options: {}".format(options))
    return options
```

7. We can add a simple configuration to see the log entries. This works as a first step to simply confirm that there are two loggers and they're being used properly:

```
if __name__ == "__main__":
    logging.basicConfig(stream=sys.stderr, level=logging.INFO)
    main()
```

This logging configuration builds the default handler object. This object simply prints all of the log messages on the given stream. This handler is assigned to the root logger; it will apply to all children of this logger. Therefore, both of the loggers created in the preceding code will go to the same stream.

Here's an example of running this script:

```
slott$ python3 ch13_r06a.py -o sum.yaml x.yaml
INFO:overview_stats.detail:read x.yaml
INFO:overview_stats.write:wrote sum.yaml
```

There are two lines in the log. Both have a severity of INFO. The first line is from the `overview_stats.detail` logger. The second line is from the `overview_stats.write` logger. The default configuration sends all loggers to `sys.stdout`.

8. In order to route the different loggers to different destinations, we'll need a more sophisticated configuration than the `basicConfig()` function. We'll use the `logging.config` module. The `dictConfig()` method can provide a complete set of configuration options. The easiest way to do this is to write the configuration in YAML and then convert this to an internal `dict` object using the `yaml.load()` function:

```
      import logging.config
      config_yaml = '''
  version: 1
  formatters:
      default:
          style: "{"
          format: "{levelname}:{name}:{message}"
          #   Example: INFO:overview_stats.detail:read x.yaml
      timestamp:
          style: "{"
          format: "{asctime}//{levelname}//{name}//{message}"

  handlers:
      console:
          class: logging.StreamHandler
          stream: ext://sys.stderr
          formatter: default
      file:
          class: logging.FileHandler
          filename: write.log
          formatter: timestamp

  loggers:
      overview_stats.detail:
          handlers:
          -   console
      overview_stats.write:
          handlers:
          -   file
          -   console
  root:
      level: INFO
  '''
```

The YAML document is enclosed in a triple-apostrophe string. This allows us to write as much text as necessary. We've defined five things in the big block of text using YAML notation:

- The value of the `version` key must be 1.
- The value of the `formatters` key defines the log format. If this is not specified, the default format shows only the message body, without any level or logger information:
    - The `default` formatter defined here mirrors the format created by the `basicConfig()` function.
    - The `timestamp` formatter defined here 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` key defines the two handlers for the two loggers. The `console` handler writes to the stream, `sys.stderr`. We specified the formatter this handler will use. This definition parallels the configuration created by the `basicConfig()` function.

The `file` handler writes to a file. The default mode for opening the file is `a`, which will append to the file with no upper limit on the file size. There are other handlers that can rotate through multiple files, each of a limited size. We've provided an explicit filename, and the formatter that will put more detail into the file than is shown on the console:

- The `loggers` key provides a configuration for the two loggers that the application will create. 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 `root` key defines the top-level logger. It has a name of `''` (the empty string) in case we need to refer to it in code. Setting the level on the root logger will set the level for all of the children of this logger.

9. Use the configuration to wrap the `main()` function like this:

```
logging.config.dictConfig(yaml.load(config_yaml))
main()
logging.shutdown()
```

10. 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.

How it works...

There are three parts to introducing logging into an application:

- Creating logger objects
- Placing log requests near important state changes
- Configuring the logging system as a whole

Creating loggers can be done in a variety of ways. Additionally, it can also be ignored. As a default, we can use the `logging` module itself as a logger. If we use the `logging.info()` method, for example, this will implicitly use the root logger.

A more 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 will be a variety of loggers serving a variety of purposes. In these cases, simply naming a logger after a module may not provide the needed level of flexibility.

There are two concepts that can be used to assign names to the loggers. It's often best to choose one of these and stick with it throughout a large application:

- Follow the package and module hierarchy. This means that a logger specific to a class might have a name like `package.module.class`. Other classes in the same module would share a common parent logger name. It's then possible to set the logging level for the the whole package, one of the specific modules, or just one of the classes.
- Follow a hierarchy based on the audience or use case. The top-level name will distinguish the audience or purpose for the log. We might have top-level loggers with names such as `event`, `audit`, and perhaps `debug`. This way, all of the audit loggers will have names that start with `"audit."`. This can make it easy to route all loggers under a given parent to a specific handler.

In the recipe, we used the first style of naming. The logger names parallel the software architecture. Placing logging requests near all the important state changes should be relatively straightforward. There are a variety of interesting state changes that belong in a log:

- Any change to a persistent resource might be a good place to include a message of level INFO. Any OS change (usually to the file-system) is a candidate for logging. Similarly, database updates, and requests that should change the state of a web services should be logged.
- Whenever there's a problem making a persistent state change, there should be a message `ERROR`. Any OS-level exceptions can be logged when they are caught and handled.
- In long, complex calculations, it may be helpful to log `DEBUG` messages after particularly import assignment statements. In some cases, this is a hint that the long calculation might need to be decomposed into functions so that they can be tested separately.
- Any change to an internal application resource deserves a `DEBUG` message so that object state changes can be tracked through the log.
- When the application enters an erroneous state. This should generally be due to an exception. In some cases, an `assert` statement will be used to detect the state of the program and raise an exception when there are problems. Some exceptions are logged at the `EXCEPTION` level. Some exceptions, however, only need `DEBUG` level messages because the exception is being silenced or transformed. Some exceptions may be logged at the `ERROR` or `CRITICAL` level.

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 loggers will all quietly create log events, but won't display them.

With a minimal configuration, we can see all of the log events on the console. This can be done with the `basicConfig()` method and covers a large number of simple use cases without any real fuss. Instead of a stream, we can use filename to provide a named file. Perhaps the most important feature is providing a simple way to enable debugging by setting the logging level on the root logger from `basicConfig()` method.

The example configuration in the recipe used two common handlers—the `StreamHandler` and `FileHandler` classes. There are over a dozen more handlers, each with unique features for gathering and publishing log messages.

There's more...

- See the *Designing scripts for composition* recipe for the complementary part of this application

Combining two applications into one

In the *Designing scripts for composition* recipe, we looked at a simple application that creates a collection of statistics by simulating a process. In the *Using logging for control and audit output* recipe, we looked at an application that summarizes a collection of statistics. In this recipe, we'll combine those two separate applications to create a single, composite application that both creates and summarizes the statistical data.

There are several common approaches to combining these two applications:

- A shell script can run the simulator and then run the analyzer
- A Python program can stand in for the shell script and use the `runpy` module to run each program
- We can build a composite application from the essential features of each application

In the *Designing scripts for composition* recipe, we examined three aspects of an application:

- Gathering input
- Producing output
- The essential processing that connects input and output

In the recipe, we looked at a design pattern that would allow several Python language components to be combined into a larger application.

How can we combine applications to create a composite?

Getting ready

In the *Designing scripts for composition* and *Using logging for control and audit output* recipes, we followed a design pattern that separated the input gathering, the essential processing, and the production of output. The objective behind that design pattern was gathering 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. In electrical engineering, it's a problem with circuit design, and it's often solved by using a device called a transformer. This can be used to match impedance between circuit components.

In database engineering, this kind of problem surfaces when the database has normalized, flat data, but the programming language uses richly structured complex objects. For SQL databases, this is a common problem and packages such as **SQLAlchemy** are used as an **Object-Relational Management** (**ORM**) layer. This layer is a transformer between flat database rows (often from multiple tables) and complex Python structures.

When building a composite application, the impedance mismatch that surfaces in this example is a cardinal problem. The simulator is designed to run more frequently than the statistical summarizer. We have several choices for addressing cardinal issues such as this one:

- **Total Redesign**: This may not be a sensible alternative because the two component applications have an established base of users. In other cases, the new use cases are an opportunity to make sweeping fixes and retire some technical debt.
- **Include the Iterator**: This means that when we build the composite application, we'll add a `for` statement to perform many simulation runs and then process this into a single summary. This parallels the original design intent.
- **List of One**: This means that the composite application will run one simulation and provide this single simulation output to the summarizer. This modifies the structure to do more summarization; the summaries may need to be combined into the expected single result.

The choice between these depends on the user story that leads to creating the composite application in the first place. It may also depend on the established base of users. For our purposes, we'll assume that the users have come to realize that 1,000 simulation runs of 1,000 samples is standard, and they would like to follow the *Include the Iterator* design to create a composite process.

As an exercise, the reader should pursue the alternative design. Assume instead that the users would rather run 1,000,000 samples in a single simulation. For this, the users would prefer the summarizer work with a *List of One* design.

We'll also look at another option. In this case, we'll perform 100 simulation runs spread over a number of concurrent worker processes. This will reduce the time to create a million samples. This is a variation of the *Include the Iterator* composite design.

How to do it...

1. Follow a design pattern that decomposes a complex process into functions that are independent of input or output details. See the *Designing scripts for composition* recipe for details on this.
2. Import the essential functions from the working modules. In this case, the two modules have the relatively uninteresting names, `ch13_r05` and `ch13_r06`:

```
from ch13_r05 import roll_iter
from ch13_r06 import gather_stats
```

3. Import any other modules required. We'll use a `Counter` function to prepare the summaries in this example:

```
from collections import Counter
```

4. Create a new function that combines the existing functions from the other applications. The output from one function is input to another:

```
    def summarize_games(total_games, *, seed=None):
        game_statistics = gather_stats(roll_iter(total_games,
seed=seed))
        return game_statistics
```

   In many cases, it makes more sense to explicitly stack the functions, creating intermediate results. This is particularly important when there are multiple functions creating a kind of map-reduce pipeline:

```
def summarize_games_2(total_games, *, seed=None):
    game_roll_history = roll_iter(total_games, counts, seed=seed)
    game_statistics = gather_stats(game_roll_history)
    return game_statistics
```

   We've broken the processing into steps with intermediate variables. The `game_roll_history` variable is the output from the `roll_iter()` function. The output from this generator is the iterable input to the `gather_states()` function, which is saved in the `game_statistics` variable.

5. Write the output-formatting functions that use this composite process. Here, for example, is a composite process that exercises the `summarize_games()` function. This also writes the output report:

```
def simple_composite(games=100000):
    start = time.perf_counter()
    stats = summarize_games(games)
```

```
end = time.perf_counter()
games = sum(stats.values())
print('games', games)
print(win_loss(stats))
print("{:.2f} seconds".format(end-start))
```

6. Gathering command-line options can be done using the `argparse` module. There are examples of this in recipes including the *Designing scripts for composition* recipe.

How it works...

The central feature of this design is a separation of the various concerns of the application into isolated functions or classes. The two component applications started with a design divided up among input, process, and output concerns. Starting from this base made it easy to import and reuse the processing. This also left the two original applications in place, unchanged.

The objective is to import functions from working modules and avoid copy and paste programming. Copying a function from one file and pasting it into another means that any change made to one is unlikely to be made to the other. The two copies will slowly diverge, leading to a phenomenon sometimes called *code rot*.

When a class or function does several things, the reuse potential is reduced. This leads to the observation of **Inverse Power Law of Reuse**—the re-usability of a class or function, *R(c)*, is related to the inverse of the number of features in that class or function, *F(c)*:

$$R(c) \propto 1 / F(c)$$

A single feature aids reuse. Multiple features reduce the opportunities for reuse of a component.

When we look at the two original applications from the *Designing scripts for composition* and *Using logging for control and audit output* recipes, we can see that the essential functions had few features. The `roll_iter()` function simulated a game, and yielded results. The `gather_stats()` function gathered statistics from any source of data.

The idea of counting features depends, of course, on the level of abstraction. From a small-scale view, the functions do many small things. From a very large scale view, the functions require several helpers to form a complete application; from this viewpoint, an individual function is only a part of a feature.

Our focus here is on technical features of the software. This has nothing to do with the agile concept of feature as a unifying concept behind multiple user stories. In this context, we're talking about software architecture technical features—input, output, processing, OS resources used, dependencies, and so on.

Pragmatically, the relevant technical features are tied to user stories. This puts the scale question into the realm of software properties as perceived by users. If the user sees more than one feature, it means that reuse may be a struggle.

In this case, one application created files. The second application summarized files. Feedback from users may have revealed that the distinction was not important or perhaps confusing. This lead to a redesign to create a one-step operation from the two original steps.

There's more...

We'll look at three other architectural features that can be part of the composite application:

- **Refactoring**: The *Combining two applications into one* recipe did not properly distinguish between processing and output. When trying to create a composite application, we may need to refactor the component modules.
- **Concurrency**: Running several `roll_iter()` instances in parallel to use multiple cores.
- **Logging**: When multiple applications are combined, the combined logging can become complex.

Refactoring

In some cases, it becomes necessary to rearrange software to extract the useful features. In one of the components, the `ch13_r06` module, the following function was available:

```
def process_all_files(result_file, file_names):
    for source_path in (Path(n) for n in file_names):
        detail_log.info("read {}".format(source_path))
        with source_path.open() as source_file:
            game_iter = yaml.load_all(source_file)
            statistics = gather_stats(game_iter)
            result_file.write(
                yaml.dump(dict(statistics), explicit_start=True)
            )
```

This combines source file iteration, detailed processing, and output creation in one place. The `result_file.write()` output processing is a single, complex statement that's difficult to extract from this function.

In order to reuse this feature properly between two applications, we'll need to refactor the `ch13_r06` application so that the file output is not buried in the `process_all_files()` function. In this case, the refactoring isn't too difficult. In some cases, the wrong abstractions are chosen, and the refactoring is extremely difficult.

One line of code, `result_file.write(...)`, needs to be replaced with a separate function. This is a small change. Details are left as an exercise for the reader. When defined as a separate function, it is easier to replace.

This refactoring makes the new function available for other composite applications. When multiple applications share a common function, then it's much more likely that outputs between the applications are actually compatible.

Concurrency

The underlying reason for running many simulations followed by a single summary is a kind of map-reduce design. The detailed simulations can be run concurrently, using multiple cores and multiple processors. The final summary, however, needs to be created from all of the simulations via a statistical reduction.

We often use OS features to run multiple concurrent processes. The POSIX shells include the `&` operator which can be used to fork concurrent subprocesses. Windows has a **`start`** command, which is similar. We can leverage Python directly to spawn a number of concurrent sumulation processes.

One module for doing this is the `futures` module from the `concurrent` package. We can build a parallel simulation processor by creating an instance of `ProcessPoolExecutor`. We can submit requests to this executor and then collect the results from those concurrent requests:

```
import concurrent.futures

def parallel():
    start = time.perf_counter()
    total_stats = Counter()
    worker_list = []
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for i in range(100):
            worker_list.append(executor.submit(summarize_games, 1000))
```

```
        for worker in worker_list:
            stats = worker.result()
            total_stats.update(stats)
    end = time.perf_counter()

    games = sum(total_stats.values())
    print('games', games)
    print(win_loss(total_stats))
    print("{:.2f} seconds".format(end-start))
```

We've initialized three objects: `start`, `total_stats`, and `worker_list`. The `start` object has the time at which processing started; `time.perf_counter()` is often the most accurate timer available. `total_stats` is a `Counter` object that will collect the final statistical summary. `worker_list` will be a list of individual `Future` objects, one for each request that's made.

The `ProcessPoolExecutor` method defines a processing context in which a pool of workers are available to handle requests. By default, the pool has as many workers as the number of processors. Each worker process is running an executor which imports the given module. All functions and classes defined in the module are available to the workers.

The `submit()` method of an executor is given a function to execute along with arguments to that function. In this example, there will be 100 requests made, each of which will simulate 1,000 games and return the sequence of dice rolls for those games. `submit()` returns a `Future` object, which is a model for the working request.

After submitting all 100 requests, the results are collected. The `result()` method of a `Future` object waits for the processing to finish and gathers the resulting object. In this example, the result is a statistical summary of 1,000 games. These are then combined to create the overall `total_stats` summary.

Here's a comparison between serial and parallel execution:

```
games 100000
Counter({'loss': 50997, 'win': 49003})
2.83 seconds
games 100000
Counter({'loss': 50523, 'win': 49477})
1.49 seconds
```

The processing time is cut in half. Since there are 100 concurrent requests, why isn't the time cut by 1/100th of the original time? The observation is that there is considerable overhead in spawning the subprocesses, communicating the request data, and communication the result data.

Logging

In the *Using logging for control and audit output* recipe, 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.

Logging involves a three-part recipe:

1. Creating logger objects. This is generally a line such as `logger = logging.get_logger('some_name')`. It's generally done once at the class or module level.
2. Using the logger objects to collect events. This involves lines such as `logger.info('some message')`. These lines are scattered throughout an application.
3. Configuring the logging system as a whole. There are two possibilities for log configuration in an application:

    - As external as possible. In this case, the logging configuration is done only at the outermost, global scope of the application:

    ```
    if __name__ == "__main__":
        logging configuration goes only here.
        main()
        logging.shutdown()
    ```

    This guarantees that there will only be a single configuration of the logging system.

    - Somewhere inside a class, function, or module. In this case, we may have several modules that are all attempting to do logging configuration. This is tolerated by the logging system. It can, however, be confusing to debug.

These recipes all follow the first approach. If all applications configure logging in the most global scope, then it's easy to understand how to configure a composite application.

In cases where there are multiple logging configurations, there are two approaches that a composite application can follow:

- The composite application contains a final configuration, which intentionally overwrites all previously-defined loggers. This is the default, and can be stated explicitly via `incremental: false` in a YAML configuration document.

- The composite application preserves other application loggers, and merely modifies the logger configurations, perhaps by setting the overall level. This is done by including `incremental: true` in the YAML configuration document.

The use of incremental configuration is helpful when combining Python applications that don't isolate the logging configuration. It can take some time to read and understand the code from each application in order to properly configure logging for composite applications.

See also

- In the *Designing scripts for composition* recipe, we looked at the core design pattern for a composable application

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. For example, `git pull`, `git commit`, and `git push`.

What's central to this design is the idea of a collection of individual commands. Each of the various features of git can be thought of as a separate class definition that performs a given function.

When we enter a command such as `git pull`, it's as if the program, `git`, is locating a class to implement the command.

How can we create families of closely related commands?

Getting ready

We'll imagine an application built from three commands. This is based on the applications shown in the *Designing scripts for composition*, *Using logging for control and audit output*, and *Combining two applications into one* recipes. We'll have three applications—*simulate*, *summarize*, and a combined application called *simsum*.

These features are based on modules with names such as `ch13_r05`, `ch13_r06`, and `ch13_r07`. The idea is that we can restructure these separate modules into a single class hierarchy following the Command design pattern.

There are two key ingredients to this design:

1. The client depends only on 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.

When we've done this, then an overall application script can create and execute any one of the `Command` subclasses.

How to do it...

1. The overall application will have a structure that attempts to separate the features into two categories—argument parsing and command execution. Each subcommand will include both processing and the output bundled together.

   Here's the `Command` superclass:

   ```
   from argparse import Namespace

   class Command:
       def execute(self, options: Namespace):
           pass
   ```

   We're going to rely on the `argparse.Namespace` to provide a very flexible collection of options to each subclass. This is not required, but will be helpful in the *Managing arguments and configuration in composite applications* recipe. Since that recipe will include option parsing, it seems best to focus each class on using `argparse.Namespace`.

2. Create a subclass of the `Command` superclass for the `Simulate` command:

   ```
   import ch13_r05

   class Simulate(Command):
       def __init__(self, seed=None):
           self.seed = seed
       def execute(self, options):
           self.game_path = Path(options.game_file)
           data = ch13_r05.roll_iter(options.games, self.seed)
   ```

```
                ch13_r05.write_rolls(self.game_path, data)
```

We've wrapped the processing and output from the `ch13_r05` module into the `execute()` method of this class.

3. Create a subclass of the `Command` superclass for the `Summarize` command:

```
        import ch13_r06

        class Summarize(Command):
            def execute(self, options):
                self.summary_path = Path(options.summary_file)
                with self.summary_path.open('w') as result_file:
                    ch13_r06.process_all_files(result_file,
options.game_files)
```

For this class, we've wrapped the file creation and the file processing into the `execute()` method of the class.

4. All of the overall processes can be performed by the following `main()` function:

```
        from argparse import Namespace

        def main():
            options_1 = Namespace(games=100, game_file='x.yaml')
            command1 = Simulate()
            command1.execute(options_1)

            options_2 = Namespace(summary_file='y.yaml',
game_files=['x.yaml'])
            command2 = Summarize()
            command2.execute(options_2)
```

We've created two commands, an instance of `Simulate` class, and an instance of the `Summarize` class. These can be executed to provide a combined feature that both simulates 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 so that any command can be created and executed. Also, new commands can be added that fit the framework.

One of the SOLID design principles is the **Liskov Substitution Principle** (**LSP**). Any of the subclasses of the `Command` abstract class can be used in place of the parent class.

Each `Command` instance has a simple interface. There are two features:

- The `__init__()` method expects a namespace object that's created by the argument parser. Each class will pick only the needed values from this namespace, ignoring any others. This allows global arguments to be ignored by a subcommand that doesn't require it.
- The `execute()` method does the processing and writes any output. This is based entirely on the values provided during initialization.

The use of the Command design pattern makes it easy to be sure that they can be interchanged with each other. The overall `main()` script can create instances of the `Simulate` or the `Summarize` class. 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 that implements a combination of existing commands:

```
class CommandSequence(Command):
    def __init__(self, *commands):
        self.commands = [command() for command in commands]
    def execute(self, options):
        for command in self.commands:
            command.execute(options)
```

This class will accept other `Command` classes via the `*commands` parameter. This sequence will combine all of the positional argument values. From the classes, it will build the individual class instances.

We might use this `CommandSequence` class like this:

```
options = Namespace(games=100, game_file='x.yaml',
    summary_file='y.yaml', game_files=['x.yaml']
)
sim_sum_command = CommandSequence(Simulate, Summarize)
sim_sum_command.execute(options)
```

We created an instance of `CommandSequence` using two other classes—`Simulate` and `Summarize`. The `__init__()` method will build an internal sequence of the two objects. The `execute()` method of the `sim_sum_command` object will then perform the two processing steps in sequence.

This design, while simple, exposes many implementation details. In particular, the two class names, and the intermediate `x.yaml` file are details that can be encapsulated into a better class design.

We can create a slightly nicer subclass of `CommandSequence` 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 SimSum(CommandSequence):
    def __init__(self):
        super().__init__(Simulate, Summarize)
```

This class definition incorporates two other classes into the already defined `CommandSequence` structure. We can continue this idea by also modifying the options slightly to eliminate the explicit values for `game_file` output from the `Simulate` step, which must also be part of the `game_files` input to the `Summarize` step.

We want to build and use a simpler `Namespace` with options like this:

```
options = Namespace(games=100, summary_file='y.yaml')
sim_sum_command = SimSum()
sim_sum_command.execute(options)
```

This means that some missing options must be injected by the `execute()` method. We'll add this method to the `SimSum` class:

```
def execute(self, options):
    new_namespace = Namespace(
        game_file='x.yaml',
        game_files=['x.yaml'],
        **vars(options)
    )
    super().execute(new_namespace)
```

This `execute()` method clones the options. It adds two additional values that are part of the integration of the commands, but not something that a user should provide.

This design avoids updating the stateful set of options. In order to leave the original options object intact, a copy was made. The `vars()` function exposes the `Namespace` as a simple dict. We can then use the `**` keyword argument technique to make the dictionary into the keyword arguments for a new `Namespace` object. This will create a shallow copy. If any stateful objects within the namespace are updated, it will be clear that both the original `options` and `new_namespace` arguments have access to the same underlying value objects.

Since `new_namespace` is a distinct collection, we can add new keys and values to this `Namespace` instance. These will only appear in `new_namespace`, leaving the original options object alone.

See also

- In the *Designing scripts for composition*, *Using logging for control and audit output*, and *Combining two applications into one* recipes, 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.

Managing arguments and configuration in composite applications

When we have a complex suite (or system) of individual applications, it's common for several applications to share common features. We can, of course, use ordinary inheritance to define a library module that provides the common classes and functions to each of the individual applications in a complex suite.

The downside of creating a number of separate applications is that the external CLI is tied directly to the software architecture. It becomes awkward to rearrange the software components because changes will also alter the visible CLI.

The coordination of common features among many application files can become awkward. For example, defining the various, one-letter abbreviated options for command-line arguments is difficult. It requires keeping some kind of master list of options, outside all of the individual application files. It seems like this should be kept in one place in the code somewhere.

Is there an alternative to inheritance? How can we assure that a suite of applications can be refactored without creating unexpected changes to the CLI or requiring complex additional design notes?

Getting ready

Many complex suites of applications follow a design pattern similar to the one used by Git. There's a base command, `git`, with a number of subcommands. For example, `git pull`, `git commit`, and `git push`. The core of the command-line interface can be centralized by the `git` command. The subcommands can then be organized and reorganized as needed with fewer changes to the visible CLI.

We'll imagine an application built from three commands. This is based on the applications shown in the *Designing scripts for composition*, *Using logging for control and audit output*, and *Combining two applications into one* recipes. We'll have three applications with three commands: `craps simulate`, `craps summarize`, and the combined application `craps simsum`.

We'll rely on the subcommand design from the *Combining many applications using the Command design pattern* recipe. This will provide a handy hierarchy of `Command` subclasses:

- The `Command` class is an abstract superclass
- The `Simulate` subclass performs the simulation functions from the *Designing scripts for composition* recipe
- The `Summarize` subclass performs summarization functions from the *Using logging for control and audit output* recipe
- A `SimSum` subclass can perform combined simulation and summarization, following the ideas of the *Combining two applications into one* recipe

In order to create a simple command-line application, we'll need appropriate argument parsing.

This argument parsing will rely on the subcommand parsing capability of the `argparse` module. We can create a common set of command options that apply to all subcommands. We can also create unique options for each subcommand.

How to do it...

1. Define the command interface. This is an exercise in **User Experience** (**UX**) design. While most UX is focused on web and mobile device applications, the core principles are appropriate for CLI applications and servers, as well.

   Earlier, we noted that the root application will be `craps`. It will have the following three subcommands:

   ```
   craps simulate -o game_file -g games
   craps summarize -o summary_file game_file ...
   craps simsum -g games
   ```

2. Define the root Python application. Consistent with other files in this book, we'll call it `ch13_r08.py`. At the OS level, we can provide an alias or a link to make the visible interface match the user expectation of `craps`.
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 `Simulate`, `Summarize`, and `SimSum` subclasses.
4. Create the overall argument parser then create a subparser builder. The `subparsers` object will be used to create each subcommand's argument definition:

   ```
   import argparse
   def get_options(argv):
       parser = argparse.ArgumentParser(prog='craps')
       subparsers = parser.add_subparsers()
   ```

   For each command, create a parser and add arguments that are unique to that command.

5. Define the `simulate` command with the two options that are unique to simulation. We'll also provide a special default value that will initialize the resulting `Namespace` object:

   ```
           simulate_parser = subparsers.add_parser('simulate')
           simulate_parser.add_argument('-g', '--games', type=int,
   default=100000)
           simulate_parser.add_argument('-o', '--output',
   dest='game_file')
           simulate_parser.set_defaults(command=Simulate)
   ```

6. Define the `summarize` command, with the arguments unique to this command. Provide the default value that will populate the `Namespace` object:

```
            summarize_parser = subparsers.add_parser('summarize')
            summarize_parser.add_argument('-o', '--output',
dest='summary_file')
            summarize_parser.add_argument('game_files', nargs='*')
            summarize_parser.set_defaults(command=Summarize)
```

7. Define the `simsum` command, and similarly, provide a unique default value that makes processing the `Namespace` easier:

```
            simsum_parser = subparsers.add_parser('simsum')
            simsum_parser.add_argument('-g', '--games', type=int,
default=100000)
            simsum_parser.add_argument('-o', '--output',
dest='summary_file')
            simsum_parser.set_defaults(command=SimSum)
```

8. Parse the command-line values. In this case the overall argument to the `get_options()` function is expected to be the value of `sys.argv[1:]`, which includes the arguments to the Python command. We can override the argument value for testing purposes:

```
        options = parser.parse_args(argv)
        if 'command' not in options:
            parser.print_help()
            sys.exit(2)
        return options
```

   The overall parser includes three subcommand parsers. One will handle the `craps simulate` command, another handles `craps summarize`, and the third handles `craps simsum`. Each subcommand has slightly different combinations of options.

   The `command` option is set only via the `set_defaults()` method. This sends useful, additional information about the command to be executed. In this case, we've provided the class that must be instantiated.

9. The overall application is defined by the following `main()` function:

```
    def main():
        options = get_options(sys.argv[1:])
        command = options.command(options)
        command.execute()
```

The options will be parsed. Each distinct subcommand sets a unique class value for the `options.command` argument. This class is used to build an instance of a `Command` subclass. This object will have an `execute()` method that does the real work of this command.

10. Implement the OS wrapper for the root command. We might have a file named `craps`. The file would have rx permissions so that it was readable by other users. The content of the file could be this line:

```
python3.5 ch13_r08.py $*
```

This small shell script provides a handy way to enter a command of **`craps`** and have it properly execute a Python script with a different name.

We can create a bash shell alias like this:

```
alias craps='python3.5 ch13_r08.py'
```

This can be placed in a `.bashrc` file to define a **`craps`** 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 that is used to build each distinct subcommand parser. We assigned this object to the variable `subparsers`.

We also defined a simple `command` argument in the top-level parser. This argument can only be filled by the defaults defined for each of the sub-parsers. This provides a value that shows which of the subcommands was actually invoked.

Each sub-parser is built using the `add_parser()` method of the sub-parsers object. The `parser` object that is returned can then have arguments and defaults defined.

When the overall parser is executed, it will parse any arguments defined outside the subcommands. If there's a subcommand, this is used to determine how to parse the remaining arguments.

Look at the following command:

```
craps simulate -g 100 -o x.yaml
```

This command will be parsed to create a `Namespace` object that looks like this:

```
Namespace(command=<class '__main__.Simulate'>, game_file='x.yaml',
games=100)
```

The `command` attribute in the `Namespace` object is the default value provided as part of the subcommand definition. The values for `game_file` and `games` come from the `-o` and `-g` options.

The Command design pattern

Creating interchangeable, polymorphic classes for the various subcommands creates a design that's easily refactored or expanded. The `Command` design pattern strongly encourages each individual subclass to have an identical signature so that any one of the available command classes can be created and executed.

One of the SOLID design principles is the Liskov Substitution Principle. Any of the subclasses of the Command abstract class can be used in place of the parent class.

Each `Command` has a consistent interface:

- The `__init__()` method expects a namespace object that's created by the argument parser. Each class will pick only the needed values from this namespace, ignoring any others. This allows global arguments to be ignored by a subcommand that doesn't require it.
- The `execute()` method does the processing and writes any output. This is based entirely on the values provided during initialization.

The use of the Command design pattern makes it easy to ensure that they can be interchanged with each other. The substitution principle means that the `main()` function can simply create an instance and then execute the `execute()` method of the object.

There's more...

We can consider pushing the subcommand parser details down into each class definition. For example, the `Simulate` class defines two arguments:

```
simulate_parser.add_argument('-g', '--games', type=int, default=100000)
simulate_parser.add_argument('-o', '--output', dest='game_file')
```

It doesn't seem appropriate for the `get_option()` function to define these details about the implementation class. It seems like a properly encapsulated design would allocate this detail to each `Command` subclass.

We would need to add a static method that configures a given parser. The new class definitions would look like this:

```
import ch13_r05
class Simulate(Command):
    def __init__(self, options, *, seed=None):
        self.games = options.games
        self.game_file = options.game_file
        self.seed = seed
    def execute(self):
        data = ch13_r05.roll_iter(self.games, self.seed)
        ch13_r05.write_rolls(self.game_file, data)
    @staticmethod
    def configure(simulate_parser):
        simulate_parser.add_argument('-g', '--games', type=int,
default=100000)
        simulate_parser.add_argument('-o', '--output',
dest='game_file')
```

We've added a `configure()` method to configure a parser. This change makes it very easy to see how the `__init__()` arguments will be created by parsing the command-line values. This allows us to rewrite the `get_option()` function, as well:

```
import argparse
def get_options(argv):
    parser = argparse.ArgumentParser(prog='craps')
    subparsers = parser.add_subparsers()

    simulate_parser = subparsers.add_parser('simulate')
    Simulate.configure(simulate_parser)
    simulate_parser.set_defaults(command=Simulate)

    # etc. for each class
```

This will leverage the static `configure()` method to provide the parameter details. The default value for the command argument can be handled by the overall `get_options()` because it doesn't involve internal details.

See also

- See the *Designing scripts for composition*, *Using logging for control and audit output*, and *Combining two applications into one* recipes for background on the components
- See the *Using argparse to get command-line input* recipe in `Chapter 5`, *User Inputs and Outputs*, for more background in argument parsing

Wrapping and combining CLI applications

One common kind of automation involves running several programs, none of which are actually Python applications. Since the programs aren't written in Python, it's impossible to rewrite each program to create a composite Python application. We can't follow the *Combining two applications into one* recipe.

Instead of aggregating the functionality, the alternative is to wrap the other programs in Python to provide a higher level construct. The use case is very similar to the use case for writing a shell script. The difference is that Python is used instead of the shell language. Using Python has some advantages:

- Python has a rich collection of data structures. The shell only has strings and arrays of strings.
- Python has an outstanding unit test framework. This provides confidence that the Python version of a shell script works without the risk of crashing a widely-used service.

How do we run other applications from within Python?

Getting ready

In the *Designing scripts for composition* recipe, we identified an application that did some processing leading 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 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 assume that the `ch13_r05` application is a native binary application; it might have been written in C++ or Fortran. This means that we can't simply import the Python module that comprises the application. Instead, we'll have to process this application by running a separate OS process.

We will use the `subprocess` module to run an application program at the OS level. There are two common use cases for running another binary program from within Python:

- There isn't any output, or we don't want to gather it 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 where many child programs are all writing to the standard error logs; the parent Python program is merely starting a child processes.
- We need to capture and possibly analyze the output to retrieve information or ascertain the level of success.

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 is scrutinized by the Python wrapper program.

How to do it...

1. Import the `subprocess` module:

```
import subprocess
```

2. Design the command line. Generally, this should be tested at the OS prompt to be sure that it does the right things:

```
slott$ python3 ch13_r05.py --samples 10 --output x.yaml
```

   The output filename needs to be flexible, so that we can run the program hundreds of times. This means creating files with names such as `game_{n}.yaml`.

3. Write a statement that iterates through the appropriate commands. Each command can be built as a sequence of individual words. Start with the working shell command and split that line on the spaces to create a proper sequence of words:

```
files = 100
for n in range(files):
    filename = 'game_{n}.yaml'.format_map(vars())
    command = ['python3', 'ch13_r05.py',
        '--samples', '10', '--output', filename]
```

   This will create the various commands. We can use a `print()` function to show each command and confirm that the filenames are defined properly.

4. Evaluate the `run()` function from the `subprocess` module. This will execute the given command. Provide `check=True` so that if there's any problem, it will raise a `subprocess.CalledProcessError` exception:

```
subprocess.run(command, check=True)
```

5. In order to test this properly, the entire sequence should be transformed into a proper function. If there will be more, related commands in the future, it should be a method of a subclass in a `Command` class hierarchy. See the *Managing arguments and configuration in composite applications* recipe.

How it works...

The `subprocess` module is how Python programs run other programs available on a given computer. The `run()` function, does a number of things for us.

In a POSIX (such as Linux or Mac OS X) context, the steps are similar to the following sequence:

- Prepare the `stdin`, `stdout`, and `stderr` file descriptors for the child process. In this case, we've accepted the defaults, which means that the child inherits the files being used by the parent. If the child process prints to `stdout`, it will appear on the same console being used by the parent.
- Invoke the `os.fork()` function to split the current process into a parent and a child. The parent will be given the process ID of the child; it can then wait for the child to finish.
- In the child, execute the `os.execl()` function (or a similar function) to provide the command path and arguments that will be executed by the child.

- The child process then runs, using the given `stdin`, `stdout`, and `stderr` files.
- The parent, meanwhile, uses a function such as `os.wait()` to wait for the child to finish and return the final status.
- Since we used the `check=True` option, a non-zero status is transformed into an exception by the `run()` function.

An OS shell, such as bash, conceals these details from application developers. The `subprocess.run()` function, similarly, hides the details of creating and waiting for a child process.

Python, with the `subprocess` module, offers many features similar to the shell. Most importantly, Python offers several additional sets of features:

- A much richer collection of data structures.
- Exceptions to identify problems that arise. This is much simpler and more reliable than inserting `if` statements throughout a shell script to check status codes.
- A way to unit test the script without using OS resources.

There's more...

We'll add a simple cleanup feature to this script. The idea is that all of the output files should be created as an atomic operation. We want all of the files, or none of the files. We don't want an incomplete collection of data files.

This fits with the ACID properties:

- **Atomicity**: The entire set of data is available or it is not available. The collection is a single, indivisible unit of work.
- **Consistency**: The file-system should move from one internally consistent state to another consistent state. Any summaries or indices will properly reflect the actual files.
- **Isolation**: If we want to process data concurrently, then having multiple, parallel processes should work. Concurrent operations should not interfere with each other.
- **Durability**: Once the files are written, they should remain on the file-system. This property almost goes without saying for files. For more complex databases, it becomes necessary to consider transaction data that might be acknowledged by a database client, but not actually written yet to a server.

Most of these features are relatively simple to achieve using OS processes with separate working directories. The atomicity property, however, leads to a need for a cleanup operation.

In order to clean up, we'll need to wrap the core processing with a `try:` block. The overall function would look like this:

```
import subprocess
from pathlib import Path

def make_files(files=100):
    try:
        for n in range(files):
            filename = 'game_{n}.yaml'.format_map(vars())
            command = ['python3', 'ch13_r05.py',
                '--samples', '10', '--output', filename]
            subprocess.run(command, check=True)
    except subprocess.CalledProcessError as ex:
        for partial in Path('.').glob("game_*.yaml"):
            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.

Since the processing has failed, it's important to raise an exception. In some cases, an application may define a new exception, unique to this application. That new exception can be raised instead, re-raising the original `CalledProcessError` exception.

Unit test

In order to unit test this, we'll need to mock two external objects. We need a mock for the `run()` function in the `subprocess` module. We don't want to actually run the other process, but we want to be sure that the `run()` function is called appropriately from the `make_files()` function.

We also need to mock the `Path` class and the resulting `Path` object. These provide the filenames, and will have the `unlink()` method called. We need to create mocks for this so that we can be sure only the appropriate files will be unlinked by the real application.

Testing with mock objects means that we never run the risk of accidentally deleting useful files when testing. This is a significant benefit of using Python for this kind of automation.

Here's the setup where we define the various mock objects:

```
import unittest
from unittest.mock import *

class GIVEN_make_files_exception_WHEN_call_THEN_run(unittest.TestCase):
    def setUp(self):
        self.mock_subprocess_run = Mock(
            side_effect = [
                None,
                subprocess.CalledProcessError(2, 'ch13_r05')]
        )
        self.mock_path_glob_instance = Mock()
        self.mock_path_instance = Mock(
            glob = Mock(
                return_value = [self.mock_path_glob_instance]
            )
        )
        self.mock_path_class = Mock(
            return_value = self.mock_path_instance
        )
```

We've defined `self.mock_subprocess_run`, which will behave somewhat like the `run()` function. We've used the `side_effect` attribute to provide multiple return values for this function. The first response will be the `None` object. The second response, however, will be a `CalledProcessError` exception. This exception requires two arguments, a process return code, and the original command.

The `self.mock_path_class`, shown last, responds to calls to the `Path` class requests. This will return a mocked instance of the class. The `self.mock_path_instance` object is the mock instance of `Path`.

The first path instance that's created will have the `glob()` method evaluated. For this, we've used the `return_value` attribute to return a list of `Path` instances to be deleted. In this case, the return value will be a single `Path` object that we expect to be unlinked.

The `self.mock_path_glob_instance` object is the return from `glob()`. This should be unlinked if the algorithm operates correctly.

Here's the `runTest()` method for this unit test:

```
def runTest(self):
    with patch('__main__.subprocess.run', self.mock_subprocess_run), \
        patch('__main__.Path', self.mock_path_class):
        self.assertRaises(
            subprocess.CalledProcessError, make_files, files=3)
```

```
            self.mock_subprocess_run.assert_has_calls(
                [call(
                    ['python3', 'ch13_r05.py', '--samples', '10',
                     '--output', 'game_0.yaml'],
                    check=True),
                 call(
                    ['python3', 'ch13_r05.py', '--samples', '10',
                    '--output', 'game_1.yaml'],
                    check=True),
                 ]
             )
             self.assertEqual(2, self.mock_subprocess_run.call_count)
             self.mock_path_class.assert_called_once_with('.')
    self.mock_path_instance.glob.assert_called_once_with('game_*.yaml')
             self.mock_path_glob_instance.unlink.assert_called_once_with()
```

We've applied two patches:

- In the `__main__` module, a reference to `subprocess` will have the `run()` function replaced with the `self.mock_subprocess_run` object. This will allow us to track how many times `run()` is called. It will allow us to confirm that `run()` is called with the correct arguments.
- In the `__main__` module, the reference to `Path` will be replaced with the `self.mock_path_class` object. This will both return known values, and allow us to confirm that only the expected calls were made.

The `self.assertRaises` method is used to confirm that a `CalledProcessError` exception is properly raised when `make_files()` method is called in this particular patched context. The mocked version of `run()` method will raise an exception—we expect that exact exception to be the one that stops processing.

The mocked `run()` function be called just two times. The first call will succeed. The second call will raise an exception. We can confirm that there are exactly two calls to `run()` using the `call_count` attribute of a `Mock` object.

The `self.mock_path_instance` method is a mock for the `Path('.')` object that's created as part of exception handling. This object must have the `glob()` method evaluated. The test assertion checks the argument value to be sure that `'game_*.yaml'` is used.

Finally, the `self.mock_path_glob_instance` is a mock for the `Path` object created by `Path('.').glob('game_*.yaml')`. This object will have the `unlink()` method evaluated. This will result in deleting the file.

This unit test provides confidence that the algorithm will work as advertised. The testing is done without tying up a lot of compute resources. Most importantly, the testing is done without accidentally deleting the wrong files.

See also

- This kind of automation is often combined with other Python processing. See the *Designing scripts for composition* recipe.
- The goal is often to create a composite application; see the *Managing arguments and configuration in composite applications* recipe.
- For a variation on this recipe, see the *Wrapping a program and checking the output* recipe.

Wrapping a program and checking the output

One common kind of automation involves running several programs, none of which are actually Python applications. In this case, it's impossible to rewrite each program to create a composite Python application. In order to properly aggregate the functionality, the other programs must be wrapped as a Python class or module to provide a higher level construct.

The use case for this is very similar to the use case for writing a shell script. The difference is that Python can be a better programming language than the OS's built-in shell languages.

In some cases, the advantage Python offers is the ability to analyze the output files. A Python program might transform, filter, or summarize the output from a subprocess.

How do we run other applications from within Python and process their output?

Getting ready

In the *Designing scripts for composition* recipe, 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 assume that the `ch13_r05` application is a native binary application written in Fortran or C++. This means that we can't simply import the Python module that comprises the application. Instead, we'll have to process this application by running a separate OS process.

We will use the `subprocess` module to run an application program at the OS level. There are two common use cases for running another binary program from within Python:

- There isn't any output, or we don't want to gather it 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, we'll look at the first case, where the output is simply ignored.

Here's an example of running the `ch13_r05` application:

```
slott$ python3 ch13_r05.py --samples 10 --output=x.yaml
Namespace(output='x.yaml', output_path=PosixPath('x.yaml'), samples=10,
seed=None)
Counter({5: 7, 6: 7, 7: 7, 8: 5, 4: 4, 9: 4, 11: 3, 10: 1, 12: 1})
```

There are two lines of output that are written to the OS standard output file. The first has a summary of the options. The second line of output is a `Counter` object with a summary of the file. We want to capture the details of these `'Counter'` lines.

How to do it...

1. Import the `subprocess` module:

```
import subprocess
```

2. Design the command line. Generally, this should be tested at the OS prompt to be sure that it does the right things. We've shown an example of the command.
3. Define a generator for the various commands to be executed. Each command can be built as a sequence of individual words. The original shell command is split on spaces to create the sequence of words:

```
def command_iter(files):
    for n in range(files):
        filename = 'game_{n}.yaml'.format_map(vars())
```

```
                command = ['python3', 'ch13_r05.py',
                    '--samples', '10', '--output', filename]
                yield command
```

This generator will yield a sequence of command strings. A client can use a `for` statement to consume each of the generated commands.

4. Define a function which executes the various commands, collecting the output from each:

```
        def command_output_iter(iterable):
            for command in iterable:
                process = subprocess.run(command, stdout=subprocess.PIPE,
check=True)
                output_bytes = process.stdout
                output_lines = list(l.strip() for l in
output_bytes.splitlines())
                yield output_lines
```

Using the argument value of `stdout=subprocess.PIPE` means that the parent process will collect the output from the child. An OS-level pipe is created so that the child output can be read by the parent.

This generator will yield a sequence of lists of lines. Each list of lines will be the output lines from the `ch13_r05.py` application. There will, generally, be two lines in each list. The first line is the argument summary, and the second line is the `Counter` object.

5. Define an overall process to combine the two generators so that each command that is generated is then executed:

```
        command_sequence = command_iter(100)
        output_lines_sequence = command_output_iter(command_sequence)
        for batch in output_lines_sequence:
            for line in batch:
                if line.startswith('Counter'):
                    batch_counter = eval(line)
                    print(batch_counter)
```

The `command_sequence` variable is a generator that will produce a number of commands. This sequence is built by the `command_iter()` function.

The `output_lines_sequence` is a generator that will produce a number of lists of output lines. This is is built by the `command_output_iter()` function, which will use the given `command_sequence object`, runs a number of commands, collecting the output.

Each batch in `output_lines_sequence` will be a list of, ideally, two lines. The line that begins with `Counter` has the representation of a `Counter` object.

We've used the `eval()` function to recreate the original `Counter` object from this text representation. We can use these `Counter` objects for analysis or summarization.

Most practical applications will have to use a function that's more complex than the built-in `eval()` to interpret output. For information on processing complex line formats, see the *String parsing with regular expressions* in `Chapter 1`, *Numbers, Strings, and Tuples*, and *Reading complex formats using regular expressions* recipe in `Chapter 8`, *Input/Output, Physical Format, and Logical Layout*.

How it works...

The `subprocess` module is how Python programs run other programs available on a given computer. The `run()` function, does a number of things for us.

In a POSIX (such as Linux or Mac OS X) context, the steps are similar to the following:

- Prepare the `stdin`, `stdout`, and `stderr` files descriptors for the child process. In this case, we've arranged for the parent to collect output from the child. The child will produce `stdout` file to a shared buffer (a pipe in Linux parlance) that is consumed by the parent. The `stderr` output, on the other hand, is left alone—the child inherits the same connection the parent has, and error messages will be displayed on the same console being used by the parent.
- Invoke the `os.fork()` and `os.execl()` functions to split the current process into parent and child, and then start the child process.
- The child process then runs, using the given `stdin`, `stdout`, and `stderr`.
- The parent, meanwhile, is reading from the child's pipe while waiting for the child process to finish.
- Since we used the `check=True` option, a non-zero status is transformed into an exception.

There's more...

We'll add a simple summarization feature to this script. Each individual batch of samples produces two lines of output. The output text is split into a sequence of two lines by the expression `list(l.strip() for l in output_bytes.splitlines())`. This splits text into lines and also strips leading and trailing spaces from each line, leaving text that's slightly easier to process.

The overall script filtered these lines, looking for the line that started with `'Counter'`. Each of these lines is a text representation of a `Counter` object. Using the `eval()` function on the line will rebuild a copy of that original `Counter`. Many Python class definitions work like this—the `repr()` and `eval()` functions are inverses of each other. The `repr()` function transforms an object to text, and the `eval()` function can convert the text back to an object. This isn't true for all classes, but it is true for many.

We can create a summary of the various `Counter` objects. In order to do this, it helps to have a generator that will process the batches and yield the final summaries.

The function should look like this:

```
def process_batches():
    command_sequence = command_iter(2)
    output_lines_sequence = command_output_iter(command_sequence)
    for batch in output_lines_sequence:
        for line in batch:
            if line.startswith('Counter'):
                batch_counter = eval(line)
                yield batch_counter
```

This will create the processing commands with the `command_iter()` function. The `command_output_iter()` will process each individual command, collecting the entire set of output lines.

The nested `for` statements will examine each batch's list of lines. Within each list, it will examine each line. The line that starts with `Counter` will be evaluated with the `eval()` function. The resulting sequence of `Counter` objects is the output from this generator.

We can use a process like this to summarize the sequence of `Counter` instances:

```
total_counter = Counter()
for batch_counter in process_batches():
    print(batch_counter)
    total_counter.update(batch_counter)
print("Total")
print(total_counter)
```

We'll create `Counter` to hold the grand total, `total_counter`. The `process_batches()` will yield individual `Counter` instances from each file that's processed. These batch-level objects are used to update the `total_counter`. We can then print the grand total to show the aggregate distribution of data in all of the files created.

See also

- See the *Wrapping and combining CLI applications* recipe for another approach to this recipe.
- This kind of automation is often combined with other Python processing. See the *Designing scripts for composition* recipe.
- The goal is often to create a composite application; see the *Managing arguments and configuration in composite applications* recipe.

Controlling complex sequences of steps

In the *Combining two applications into one* recipe, we looked at ways to combine multiple Python scripts into a single, longer, more complex operation. In the *Wrapping and combining CLI applications* and *Wrapping a program and checking the output* recipes, we looked at ways to use Python to wrap non-Python programs.

How can we combine these techniques effectively? Can we create longer, more complex sequences of operations using Python?

Getting ready

In the *Designing scripts for composition* recipe, we created application that did some processing that lead to the creation of a rather complex result. In the *Using logging for control and audit output* recipe, we looked at a second application that built on those results to create a sophisticated statistical summary.

The overall process looks like this:

1. Run the `ch13_r05` program 100 times to create 100 intermediate files.
2. Run the `ch13_r06` program to summarize those intermediate files.

We've kept this simple so that it's easy to focus on the Python programming involved.

For the purposes of this recipe, we'll assume that neither of these applications is written in Python. We'll pretend that they're written in Fortran or Ada or some other language that's not directly compatible with Python.

In the *Combining two applications into one* recipe, we looked at how we can combine Python applications. When the applications are written in Python, this is the preferred approach. When applications are not written in Python, some additional work is required.

This recipe uses the Command design pattern; this supports the expansion and modification of the sequences of commands.

How to do it...

1. We'll define an abstract `Command` class. The other commands will be defined as subclasses. We'll push the subprocess processing into this class definition to simplify the subclasses:

```
import subprocess
class Command:
    def execute(self, options):
        self.command = self.create_command(options)
        results = subprocess.run(self.command,
            check=True, stdout=subprocess.PIPE)
        self.output = results.stdout
        return self.output
    def create_command(self, options):
        return ['echo', self.__class__.__name__,
repr(self.options)]
```

   The `execute()` method works by first creating the OS-level command to execute. Each subclass will provide distinct rules for the commands which are wrapped. Once the command has been built, then the `run()` function of the `subprocess` module will process this command.

   The `create_command()` method builds the sequence of words that comprise the command to be executed by the OS. The options, generally, will be used to customize the command arguments that are created. The superclass implementation of this method provides some debugging information. Each subclass must override this method to produce useful output.

2. We can use the `Command` superclass to define a command to simulate the game and create samples:

```
import ch13_r05

class Simulate(Command):
    def __init__(self, seed=None):
        self.seed = seed
    def execute(self, options):
        if self.seed:
            os.environ['RANDOMSEED'] = str(self.seed)
        super().execute(options)
    def create_command(self, options):
        return ['python3', 'ch13_r05.py`,
            '--samples', str(options.samples),
            '-o', options.game_file]
```

In this case, we provided an override for the `execute()` method so that this class could change the environment variables. This allows an integration test to set a specific random seed and confirm that the results match a fixed set of expected values.

The `create_command()` method emits the words for a command-line execution of the `ch13_r05` command. This converts the numeric value of `options.samples` to a string.

3. We can also use the `Command` superclass to define a command to summarize the various simulation processes:

```
import ch13_r06

class Summarize(Command):
    def create_command(self, options):
        return ['python3', 'ch13_r06.py',
            '-o', options.summary_file,
            ] + options.game_files
```

In this case, we only implemented `create_command()`. This implementation provides the arguments for the `ch13_r06` command.

4. Given these two commands, the overall main program can follow the design pattern from the *Designing scripts for composition* recipe. We need to gather the options, and then use these options to execute the two commands:

```
from argparse import Namespace

def demo():
    options = Namespace(samples=100,
        game_file='x12.yaml', game_files=['x12.yaml'],
        summary_file='y12.yaml')
    step1 = Simulate()
    step2 = Summarize()
    step1.execute(options)
    step2.execute(options)
```

This demonstration function, `demo()`, creates a `Namespace` instance with the parameters that could have come from the command line. It builds the two processing steps. Finally, it executes each step.

This kind of function provides a high-level script for executing a sequence of applications. It's considerably more flexible than the shell, because we can make use of Python's rich collection of data structures. Because we're using Python, we can include unit tests as well.

How it works...

There are two interlocking design patterns in this recipe:

- The `Command` class hierarchy
- Wrapping external commands by using the `subprocess.run()` function

The idea behind a `Command` class hierarchy is to make each separate step or operation into a subclass of a common, abstract superclass. In this case, we've called that superclass `Command`. The two operations are subclasses of the `Command` class. This assures that we can provide common features to all of the classes.

Wrapping external commands has several considerations. One primary question is how to build the command-line options that are required. In this case, the `run()` function will use a list of individual words, making it very easy to combine literal strings, filenames, and numeric values into a valid set of options for a program. The other primary question is how to handle the OS-defined standard input, standard output, and standard error files. In some cases, these files can be displayed on the console. In other cases, the application might capture those files for further analysis and processing.

The essential idea here is to separate two considerations:

1. The overview of the commands to be executed. This includes questions about sequence, iteration, conditional processing, and potential changes to the sequence. These are higher-level considerations related to the user stories.
2. The details of how to execute each command. This includes command-line options, output files used, and other OS-level considerations. These are more technical considerations of the implementation details.

Separating the two makes it easier to implement or modify the user stories. Changes to the OS-level considerations should not alter the user stories; the process might be faster or use less memory, but is otherwise identical. Similarly, changes to the user stories should not break the OS-level considerations.

There's more...

A complex sequence of steps can involve iteration of one or more steps. Since the high-level script is written in Python, adding iteration is done with the `for` statement:

```
def process_i(options):
    step1 = Simulate()
    options.game_files = []
    for i in range(options.simulations):
        options.game_file = 'game_{i}.yaml'.format_map(vars())
        options.game_files.append(options.game_file)
        step1.execute(options)
    step2 = Summarize()
    step2.execute(options)
```

This `process_i()` function will process the `Simulate` step many times. It uses the `simulations` option to specify how many simulations to run. Each simulation will produce the expected number of samples.

This function will set a distinct value for the `game_file` option for each iteration of the processing. Each of the resulting filenames will be unique, leading to a number of sample files. The list of files is also collected into the `game_files` option.

When the next step, the `Summarize` class, is executed, it will have the proper list of files to process. The `Namespace` object, assigned to the `options` variable, can be used to track global state changes and provide this information to subsequent processing steps.

Building conditional processing

Since the high-level programming is written in Python, it's quite easy to add additional processing that isn't based on the two applications that are wrapped. One feature might be an optional summarization step.

For example, if the options do not have a `summary_file` option, then the processing can be skipped. This might lead to a version of the `process()` function that looks like this:

```
def process_c(options):
    step1 = Simulate()
    step1.execute(options)
    if 'summary_file' in options:
        step2 = Summarize()
        step2.execute(options)
```

This `procees_c()` function will process the `Summarize` step conditionally. If there is a `summary_file` option, it will execute the second step. Otherwise, it will skip the summary step.

In this case, and the previous example, we've used Python programming features to augment the two application programs.

See also

- Generally, these kinds of processing steps are done for larger or more complex applications. See the *Combining two applications into one* and *Managing arguments and configuration in composite applications for more* recipes that work with larger and more complex composite applications.

Index

A

abductive reasoning 316
Abstract Base Classes (ABCs)
  about 157
  mapping 157
  sequence 157
  set 157
abstract class 285
abstract data type 255
ACID properties
  atomicity 636
  consistency 636
  durability 636
  isolation 636
ActiveState 61
advanced graph manipulation
  reference 158
aggregation 302
Algorithm Design Manual
  reference 92
append() method
  used, for building list 160
application processing
  decomposing 272
applications
  multiple applications, combining with command design pattern 621, 623
  testing, that involve date or time 453, 454, 455, 457, 458
  testing, that involve randomness 460, 461, 463
  two applications, combining 613, 614, 615, 616
approaches, of function designing with optional parameters
  General to Particular 112, 115, 116
  Particular to General 112, 114
architectural features, composite applications
  concurrency 617, 618
  logging 617, 620
  refactoring 617
argparse
  used, for obtaining command-line input 232, 233, 234, 235, 236, 237, 238
argument styles 238
arguments
  managing, in composite applications 627, 628, 630
arithmetic operators 16
arrays 158
ASCII characters
  creating 49, 51
aspects, processing
  inputs 598
  outputs 598
  process 598
assertions, TestCase class
  assertAlmostEquals() 447
  assertCountEqual() 447
  assertEqual() 447
  assertFalse() 447
  assertIs() 447
  assertIsInstance() 447
  assertIsNone() 447
  assertIsNot() 447
  assertIsNotNone() 447
  assertNotAlmostEquals() 447
  assertNotEqual() 447
  assertNotIsInstance() 447
  assertNotRegex() 447
  assertRegex() 447
  assertTrue() 447
assignment 205
associative store 194
audit output

logging, using for 602, 604, 606, 610, 611
authentication
about 515
implementing, for web services 544, 545
authorization 545

B

backslash
used, for breaking long statement into logical lines 67
bash 64
bash shell 566
BBEdit
reference 61
Beautiful Soup
reference 54
Tag objects 403
binary operators 16
bisect module 278
block files 350
body element, RST parser
block quotes 79
hyperlink targets 79
lists 79
literal blocks 79
paragraphs of text 79
RST comments 79
substitution definitions 79
tables 79
bootstrapping 460
break statements
potential problems, avoiding with 93, 95
built-in collections, in collections module
ChainMap 157, 278
Counter 157, 277
defaultdict 157, 277
deque 157, 277
namedtuple 157
OrderedDict 157, 278
bytes
decoding 52, 53, 54, 55
versus string 51

C

C3
reference 312
calculations 297
Cascading Style Sheet (CSS) 398
Certificate Authority (CA) 545
certificates 545
character codes
reference 48
character files 350
chunking 117
ckdate command 229
ckgid command 229
ckint command 229
ckitem command 229
ckkeywd command 229
ckpath command 229
ckrange command 229
ckstr command 230
cktime command 230
ckuid command 230
ckyorn command 230
class definition
using 589
class-as-namespace
using, for configuration 586, 588
class-level static variable 323
class
about 8
creating, that has orderable objects 329, 332, 333
designing, techniques 307
designing, with lot of processing 260, 261, 263, 264, 265
designing, with unique processing 268, 269, 270, 272
used, for data processing 254, 255, 257, 258
used, for encapsulating data 254, 255, 257, 258
clear documentation strings
writing, with RST markup 137, 138, 139, 140
CLI applications
combining 633, 635
wrapping 633, 635
client class
creating, for database access 469, 470, 471,

472
Cmd class 247
cmd module
used, for building interactive interface 229
used, for creating command-line applications 240, 241, 242, 243, 244, 246
collaborators 256
collection of objects
extending 305, 306
collection
extending 282, 283, 284
collections modules
built-in collections 277
Command design pattern 631
command-line applications
creating, cmd module used 240, 241, 242, 243, 244, 246
command-line input
obtaining, argparse used 232, 233, 234, 235, 236, 237, 238
common doctest issues
handling 434, 435, 436
Common Gateway Interface (CGI) 490
comparison directives, for individual tests
+DONT_ACCEPT_BLANKLINE 434
+DONT_ACCEPT_TRUE_FOR_1 434
+ELLIPSIS 433
+IGNORE_EXCEPTION_DETAIL 433
+NORMALIZE_WHITESPACE 433
+SKIP 433
complex data structure
deserializing 389, 390
serializing 387, 388
complex files
operations 381, 382
complex formats
reading, regular expressions used 376, 377
complex if...elif chains
designing 82, 83, 84, 85
complex numbers 24
complex object
mocking 476, 477
complex sequences
controlling, of steps 645, 647, 649
complex strings
building 38, 39, 40, 41
building, from lists of characters 43, 44
composite applications
arguments, managing in 626, 628, 630
configuration, managing in 626, 628, 630
composition
about 302
scripts, designing for 593, 598
comprehension 162
concerns
separating, via multiple inheritance 309, 310, 313
types 313
conditional processing
building 650
configuration files
finding 566, 569
Python, using for 582, 585
YAML, using for 575, 577
configuration representation 591
configuration
class-as-namespace, using for 587, 589
managing, in composite applications 627, 628, 630
context managers
about 109
creating 475
files, reading with 361, 362, 363
files, writing with 361, 362, 363
working 364, 365
context
managing, with statement used 107, 108, 109
control
logging, using for 602, 604, 606, 610, 612
copy of list
reversing 176, 177, 178, 179
Counter class
extending 261
Counter object
updating 266, 267
wrapping 261
Coupon Collector's Test 180, 260
CRUD operations 194, 494
CSV (comma-separated value) format 326, 350
csv module

used, for reading delimited files 370, 371, 372, 373, 374
CSV
upgrading, from DictReader to namedtuple reader 404, 406, 407
upgrading, from DictReader to namespace reader 410, 411
currency 17
currency calculations
performing 18, 19

D

dangling else
reference 87
data processing, from files
logical layout of data 350
physical format of data 350
data structure
reference 158
selecting 154, 156, 157
data
encapsulating, class used 254, 255, 257, 258
processing, class used 254, 255, 257, 258
database access
client class, creating for 469, 470, 471, 472
debugging tools 230, 231
deck slicing 526, 527
deep copy of objects
making 207, 208, 209, 210
delimited files
reading, with csv module 370, 371, 372, 373, 374
DeMorgan's Law 93
Dependency Inversion Principle 259
descriptions
including 71, 72
descriptive names 12
dice() function 118
dict objects 440
dictionary keys
working with 439
dictionary
creating 189, 190, 191, 192
handling, in doctest examples 201, 203
items, removing from 195, 196, 197
setdefault() method, using of 192
DictReader, to namedtuple reader
CSV, upgrading from 404, 406, 407
DictReader, to namespace reader
CSV, upgrading from 410, 411, 412
directives
about 80
using 81
directory
creating 355, 356
distinct names
number of sibling output files, making with 354, 355
div-mod pair 25
division operators 25, 27
docstring, for script
example 73
docstring
about 62
RST markup, writing in 76, 78, 79
using, for testing 422, 424
writing, for library modules 74
writing, for scripts 72
doctest examples
dictionary, handling in 201, 203
set, handling in 201, 203
writing, for floating-point values 438
writing, for mapping values 437
writing, for setting values 437
doctest parser 427
document object model (DOM) 401
Document Type Definition (DTD) 393
documentation
including 71, 72
docutils package
reference 141
docutils tools
reference 82
Don't Repeat Yourself (DRY) principle 252
dynamic content 480

E

eager attributes
updating, settable properties used 292, 293, 294, 295, 296

eager loading 262
eager processing 286
edge cases 428, 429
Elastic
  reference 465
emergent behavior 255
encodings
  reference 55
entry documents
  creating, in entrylog collection 466, 467
except clause
  potential problems, avoiding with 102, 103
exception matching rules
  leveraging 97, 98, 99, 100
exceptions
  chaining, with raise from statement 104, 106
explicit line joining 69, 70
explicit types
  writing, on function parameters 126, 127, 128, 129, 130, 131
exponent 23
expressions 70
extension
  versus inheritance 300, 301, 302
external resources
  mocking 464, 465, 466

F

features
  using, of print() function 218, 219, 220, 221, 222
Fibonacci problem
  recursions 146, 147
file dates
  comparing 356, 357
files
  about 349
  creating 355, 356
  finding, that match 358
  OS files 349
  reading, multiple contexts used 414, 417
  reading, with context managers 361, 362, 363
  removing 357
  replacing, while preserving previous version 366, 367, 369
  writing, multiple contexts used 414, 418
  writing, with context managers 361, 362, 363
filter() function
  using 173
First Normal Form (1NF) 371
First-In-First-Out (FIFO) 278
Flask framework
  references 502
  using, for RESTful APIs 495, 498, 499, 500, 501
Flask view function decorator 549
floating-point approximations 20, 21
floating-point
  need for 23
floor division
  performing 26
fraction calculations 19, 20
fractions 17
full pathname
  elements 351
function definitions 111
function parameters
  explicit types, writing on 126, 127, 128, 129, 130, 131
  mutable default values, avoiding for 211, 212, 214
functions
  designing, with optional parameters 112
  testing, that arise exceptions 430, 433
  wrapping 134, 137

G

general domain validation 227
generator expression
  list function, using on 162
getpass()
  using, for user input 223, 224, 225, 226, 228
GIVEN-WHEN-THEN style, of test case naming 422
global objects
  managing 320, 321
global singleton variables
  sharing 324
graphs 158

H

half-open interval 32
happy path 441
hard links 352
has-a relationship 300
hashes 158
heap queue 278
heapq module 278
hints 129
HTML documents
  reading 397, 398, 400, 402
HTTP basic authentication 546
HTTP digest authentication 546
HTTP headers 490
HTTP request 483
HTTP response 483
HTTP
  about 480
  use case 480
Hungarian Notation 10
Hypertext Transfer Protocol (HTTP) 465

I

IEEE floating point
  reference 24
if-break pattern 569
immutable string
  rewriting 28, 29
implicit line joining 69, 70
indexing 33
inheritance
  versus extension 300, 301, 302
inline markup
  using 81, 82
input()
  using, for user input 223, 224, 225, 226, 228
instance
  initializing, with values 297
integer processing
  reference 17
integration testing 421
Interface Segregation Principle 259
intermediate results
  assigning, for separating variables 69
invariant 89
Inverse Power Law of Reuse 616
irrational numbers 18
is-a or inheritance 302
is-a relationship 300
ISO11179
  reference 12
items
  adding, to set 184, 185
  deleting, from list 171, 172, 174
  extracting, from tuples 56, 57
  removing, from dictionary 195, 196, 197
  removing, from set 186, 187, 188, 189
  removing, pop() method used 173
  removing, remove() method used 172
  tuples, using of 55

J

Java Developer Kit (JDK) 465
joining
  explicit line joining 70
  implicit line joining 70
JSON documents
  reading 383, 385, 386
JSON request
  client module 537
  parsing 531
  server 536
  Swagger specification 533
JSON schema specification
  reference 481
JSON
  complex data structure, deserializing 389, 390
  complex data structure, serializing 387, 388
  reference 383

K

key-and-count algorithm 193
keyword parameters
  partial function, creating with 135, 136
keyword-only arguments
  forcing, with * separator 122, 123
Komodo Edit 61
Komodo IDE
  about 61

reference 61
ksh 64

L

large integers
working with 13, 15
Last-In-First-Out (LIFO) 278
Law of the Excluded Middle 83
lazy attributes
properties, using for 286, 287, 288, 290
lazy loading 262
lazy processing 286
library modules
about 71
docstrings, writing for 74
line joining 69
Liskov Substitution Principle (LSP) 259, 624
list comprehension
working 163
writing 161, 162
list function
using, on generator expression 162
list objects
creating, ways 160
list of mappings
deleting from 342, 343, 344, 345, 347
lists
about 80
building 159
building, with append() method 160
deleting from 170, 171
dicing 165, 168
extending 164, 165
items, deleting from 171, 172
slicing 165, 168
literal blocks 80
load_tests protocol
using 477
logging package
security levels 605
logging
using, for audit output 602, 604, 606, 610, 611
using, for control 602, 604, 606, 609, 611
long lines of code
writing 65, 66
long statement
breaking, () characters used 67, 68
breaking, backslash used 67

M

man page
about 75
reference 76
mantissa 23
mapping
immutable mapping 157
mutable mapping 157
maps of lists
using 325, 326, 327
Mastering Python Regular Expressions book
reference 37
math module, Python 23, 24
meaningful names
creating 7, 8, 11
memoization 146
memory leaks 107
Method Resolution Order (MRO) 312, 592
methods
finding, Python mechanism used 306
microservice 480
mixed content model 393
mixin 300
modeline 65
module files 9
writing 60, 62, 63, 64
module global variable 321, 322
multiple contexts
used, for reading files 414, 417
used, for writing files 414, 418
multiple inheritance
concerns, separating via 309, 310, 312, 313
Multipurpose Internet Mail Extensions (MIME) 511
mutable default values
avoiding, for function parameters 211, 212, 214, 215, 216
mypy
reference 129, 132

N

names
  assigning, to objects 10
  selecting 8
no echo input 223
Notepad++
  reference 61
null hypothesis 265
number of sibling output files
  making, with distinct names 354, 355
numbered lists 80
numbers
  converting, from one type to another 21

O

object 255
object-oriented programming 255
Object-Relational Management (ORM) 614
objects
  about 8
  names, assigning to 10
OData specifications
  reference 508
ontology 301
Open/Closed Principle 259
OpenAPI (Swagger) authentication 516
OpenID
  reference 545
operators
  using 179, 180, 181, 182, 183, 184
optional parameters
  functions, designing with 112
  int (str) 112
  int (str, base) 112
order for parameters
  picking, based on partial functions 132, 133, 134
order of dict keys
  controlling 198
ordered collection
  defining 335, 337, 339, 340, 341, 342
ordinary input 223
OS environment settings
  using 248, 249, 250, 251, 252
OS files
  about 349
  block files 350
  character files 350
output filename
  making, by changing input suffix 354
output files
  directory, creating based on timestamps 369, 370

P

packages
  creating 441, 442, 443
paragraphs of text 79
parse function
  defining 378
  using 379
partial function
  creating, with keyword parameters 135, 136
  creating, with positional parameters 135
Path class
  operations, example 359
pathlib module
  about 359
  used, for working with filenames 351, 352
PEP 3333
  reference 494
performance testing 422
piton 152
pop() method
  used, for removing items 173
positional parameters
  partial function, creating with 135
predicate transformer semantics
  reference 87
print() function
  features, using of 218, 219, 220, 221, 222
priority queue 278
program output
  checking 640, 643
program
  wrapping 640, 643
properties
  using, for lazy attributes 286, 288, 291
pure paths, for specific operating system

cases 360
pymongo
reference 200
Python collections
reference 165
Python Enhancement Proposal 484
reference 65
Python Enhancement Proposal number 8 (PEP-8) 12
Python Language Reference document
reference 276
Python Package Index (PyPI) 397
Python script
writing 60, 62, 63, 64
Python Standard Library documents 64
Python standard library
reference 494
Python's duck typing
leveraging 315, 316, 318
Python
operators 22
using, for configuration files 582, 584

Q

query string
parsing, in request 502, 503, 504, 507
syntax rules 507
queue 278

R

raise from statement
exceptions, chaining with 104, 106
random walk technique 190
Rate-Time-Distance (RTD) 119
rational fraction calculations 27
rational fraction value 25
rational numbers 17
read-eval-print loop (REPL) 7, 242
recursive functions
designing, around Python's stack limits 142, 143, 144
reduction 144
references 204
regular expressions
used, for reading complex formats 376, 377
remaining numbers 84
remove() method
used, for removing items 172
Representational State Transfer (REST) 465, 480
request
query string, parsing in 502, 503, 504, 507
REST requests
making, with urllib 509, 511, 512, 513, 514
RESTful APIs
Flask framework, using for 495, 498, 499, 500, 501
ReStructuredText (RST) markup 72
reusable scripts
writing, with script library switch 148, 151
RST markup
clear documentation strings, writing with 137, 138, 139, 140
writing, in docstrings 76, 78, 79
RST syntax 75

S

script library switch
reusable scripts, writing with 148, 151
scripts
about 9, 71
designing, for composition 593, 598
docstrings, writing for 72, 73
Secure Socket Layer (SSL) 467, 515, 545
security levels, logging package
CRITICAL 605
DEBUG 605
ERROR 605
INFO 605
WARNING 605
sequence
immutable sequence 157
mutable sequence 157
server
Swagger, adding to 517, 518
set methods
using 179, 180, 181, 182, 183, 184
set
handling, in doctest examples 201, 203
immutable set 157
items, adding to 184, 185

items, removing from 186, 187, 188, 189
mutable set 157
settable properties
used, for updating eager attributes 292, 293, 294, 295, 296
shallow copy 169
shallow copy of objects
making 207, 208, 209, 210
share global state 325
signature 112
Single Responsibility Principle 259
singleton objects
managing 320, 321
slice 32
slice notation 32
slice operator
forms 169
small integers
working with 13, 15
small objects
optimizing, with __slots__ 273, 274, 275, 276
soft links 352
SOLID principle
about 259
Dependency Inversion Principle 259
Interface Segregation Principle 259
Liskov Substitution Principle 259
Open/Close Principle 259
Single Responsibility Principle 259
solid state drives (SSD) 350
sophisticated collections
using 277, 280
Sphinx Python Documentation Generator
reference 75, 82
Sphinx
reference 76
spike solution 152
SQLAlchemy 614
SSL
configuring 546
stack 278
stateful class 269
stateful objects
examples, writing for 425, 426
stateless class 269
stateless functions
examples, writing for 425
stateless objects
about 269, 270
defining, existing class used 271
defining, with new class 270
static content 480
string literal concatenation
using 68, 69
string parsing
inputting 228, 229
with regular expressions 34, 35
string patterns 36
string
changing, to lowercase 31
encoding 49, 51
extra punctuation marks, removing 31
parsing 37
piece of string, slicing 29
updating, with replacement 30, 31
versus byte 51
subdomain validation 227
super flexible keyword parameters
using 118, 121, 122
Swagger specification
providing 528, 529
using 531
swagger-spec-validator project
reference 481
Swagger
about 516
adding, to server 517, 518
reference 481
symbolic links 352
sys module 222
system testing 422

T

tail recursion 143
test modules
creating 441, 442, 443
TestCase class
assertions 447
testing
about 421

docstrings, using for 422, 424
integration testing 421
performance testing 422
system testing 422
unit testing 421
tests directory 448
text role 82
trees 158
true division operator 27
true division
performing 26
true value 25
truncated division operator 28
tuples
creating 56
items, extracting from 56, 57
using, of items 55
working 57, 58
type hints
reference 132
type inferencing 328
typical response
viewing 468

U

UDEF and NIEM Naming and Design Rules standards
reference 12
Unicode characters
reference 46
using 46, 47
Unicode encodings
reference 51
unit test 637
unit testing 421
unittest module 441
working 444, 445
unittest, and doctest tests
combining 449, 451, 452
unlinking 357
URL path
parsing 518, 521, 522
urllib
REST requests, making with 509, 511, 512, 513, 514
User class
defining 550
user input
getpass(), using for 223, 224, 225, 226, 228
input(), using for 223, 224, 225, 226, 228
UTF-8 bytes
creating 49, 51
UTF-8 encoding
reference 51

V

variables
about 204
using 7
working 206
vars() function 42
view decorator
defining 553

W

web application
client 509
server 509
web log file
searching, for particular sequence of events 281, 282
web page content
dynamic content 480
static content 480
web services
authentication, implementing for 545
implementing, with WSGI 482, 483, 484, 487, 488, 489
Werkzeug project
reference 494
while statement
designing 87, 88, 89, 90, 91, 92
with context
reference 110
with statement
reference 110
used, for managing context 107, 108, 109
wrapper 492
wrapping 303, 304
WSGI server context

wsgi.errors 491
wsgi.input 491
WSGI standard
attached content 483
environment dictionary 490
methods 483
request headers 483
WSGI
web services, implementing with 482, 483, 484, 487, 488, 489

X

XML documents
reading 390, 391, 392, 394
XML Path Language (XPath) 395
XML
reference 390

Y

YAML
additional features 579, 581
reference 577
using, for configuration files 575, 577