EXPERT INSIGHT

python™

Functional Python Programming

Use a functional approach to write succinct, expressive, and efficient Python code

Foreword by:
Ricardo Bánffy,
Software Engineer, Architect, Evangelist, and Passionate Pythonista

Third Edition

Steven F. Lott

‹packt›

Functional Python Programming

Third Edition

Use a functional approach to write succinct, expressive, and efficient Python code

Steven F. Lott

BIRMINGHAM—MUMBAI

Functional Python Programming

Third Edition

Senior Publishing Product Manager: Denim Pinto
Acquisition Editor – Peer Reviews: Gaurav Gavas
Project Editor: Parvathy Nair
Development Editor: Lucy Wan
Copy Editor: Safis Editing
Technical Editor: Karan Sonawane
Indexer: Hemangini Bari
Proofreader: Safis Editing
Presentation Designer: Sandip Tadge

First published: January 2015
Second edition: April 2018
Third edition: December 2022

Production reference: 1221222

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

ISBN 978-1-80323-257-7

www.packt.com

Foreword

Python is an incredibly versatile language that offers a lot of perks for just about every group. For the object-oriented programming fans, it has classes and inheritance. When we talk about functional programming, it has functions as a first-class type, higher-order functions such as `map` and `reduce`, and a handy syntax for comprehensions and generators. Perhaps best of all, it doesn't force any of those on the user – it's still totally OK to write a script in Python without a single class or function and not feel guilty about it.

Thinking in terms of functional programming, having in mind the goals of minimizing state and side effects, writing pure functions, reducing intermediary data, and what depends on what else will also allow you to see your code under a new light. It'll also allow you to write more compact, performant, testable, and maintainable code, where instead of writing a program to solve your problem, you "write the language up", adding new functions to it until expressing the solution you designed is simple and straightforward. This is an extremely powerful mind shift – and an exercise worth doing. It's a bit like learning a new language, such as Lisp or Forth (or German, or Irish), but without having to leave the comfort of your Python environment.

Not being a pure functional language has its costs, however. Python lacks many features functional languages can use to provide better memory efficiency and speed. Python's strongest point remains its accessibility – you can fire up your Python interpreter and start playing with the examples in this book right away. This interactive approach allows exploratory programming, where you test ideas easily, and only later need to incorporate them into a more complex program (or not – like I said, it's totally OK to write a simple

script).

This book is intended for people already familiar with Python. You don't need to know much about functional programming – the book will guide you through many common approaches, techniques, and patterns used in functional programming and how they can be best expressed in Python. Think of this book as an introduction – it'll give you the basic tools to see, think, and express your ideas in functional terms using Python.

Ricardo Bánffy

Software Engineer, Architect, Evangelist, and Passionate Pythonista

Contributors

About the author

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

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

He's a technomad, and lives on a boat that's usually located on the east coast of the US. He tries to live by the words, "Don't come home until you have a story."

About the reviewers

Alex Martelli is a Fellow of the Python Software Foundation, a winner of the Frank Willison Memorial Award for contributions to the Python community, and a top-page reputation hog on Stack Overflow. He spent 8 years with IBM Research, then 13 years at Think3 Inc., followed by 4 years as a consultant, and lately 17 years at Google. He has taught programming languages, development methods, and numerical computing at Ferrara University and other venues.

Books he has authored or co-authored include two editions of *Python Cookbook*, four editions of *Python in a Nutshell*, and a chapter in *Beautiful Teams*. Dozens of his interviews and tech talks at conferences are available on YouTube. Alex's proudest achievement are the articles that appeared in *Bridge World* (January and February 2000), which were hailed as giant steps towards solving issues that had haunted contract bridge theoreticians for decades, and still get quoted in current bridge-theoretical literature.

Tiago Antao has a BEng in Informatics and a PhD in Life Sciences. He works in the Big Data space, analyzing very large datasets and implementing complex data processing algorithms. He leverages Python with all its libraries to carry out scientific computing and data engineering tasks. He also uses low-level programming languages like C, C++, and Rust to optimize critical parts of algorithms. Tiago develops on an infrastructure based on AWS, but has used on-premises computing and scientific clusters for most of his career.

While he currently works in industry, he also has exposure to the academic side of scientific computing, with two data analysis postdocs at the universities of Cambridge and Oxford, and a research scientist position at the University of Montana, where he set up, from scratch, the scientific computing infrastructure for the analysis of biological data.

He is one of the co-authors of Biopython, a major bioinformatics package written in Python. He wrote *Bioinformatics with Python Cookbook*, which is on its third edition. He has also authored and co-authored many high-impact scientific articles in the field of bioinformatics.

Join our community Discord space

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

`https://packt.link/dHrHU`

Table of Contents

Preface xxi

Chapter 1: Understanding Functional Programming 1

The functional style of programming 3

Comparing and contrasting procedural and functional styles 4

Using the functional paradigm • 5

Using a functional hybrid • 8

The stack of turtles • 9

A classic example of functional programming 10

Exploratory data analysis 15

Summary 17

Exercises 17

Convert an imperative algorithm to functional code • 18

Convert step-wise computation to functional code • 18

Revise the sqrt() function • 20

Data cleansing steps • 20

(Advanced) Optimize this functional code • 23

Chapter 2: Introducing Essential Functional Concepts 25

Functions as first-class objects 26

Pure functions • 27

Higher-order functions • 29
Immutable data **30**
Strict and non-strict evaluation **32**
Lazy and eager evaluation **33**
Recursion instead of an explicit loop state **34**
Functional type systems **39**
Familiar territory **41**
Learning some advanced concepts **41**
Summary **42**
Exercises **43**
Apply map() to a sequence of values • 44
Function vs. lambda design question • 45
Optimize a recursion • 45

Chapter 3: Functions, Iterators, and Generators **47**

Writing pure functions **48**
Functions as first-class objects **51**
Using strings **54**
Using tuples and named tuples **56**
Using generator expressions **58**
Exploring the limitations of generators • 63
Combining generator expressions • 65
Cleaning raw data with generator functions **66**
Applying generators to built-in collections **69**
Generators for lists, dicts, and sets • 69
Using stateful mappings • 73
Using the bisect module to create a mapping • 76
Using stateful sets • 78
Summary **78**

Exercises **79**

Rewrite the some_function() function • 79

Alternative Mersenne class definition • 80

Alternative algorithm implementations • 81

Map and filter • 82

Dictionary comprehension • 82

Raw data cleanup • 82

Chapter 4: Working with Collections 85

An overview of function varieties **86**

Working with iterables **87**

Parsing an XML file • 89

Parsing a file at a higher level • 92

Pairing up items from a sequence • 95

Using the iter() function explicitly • 98

Extending an iteration • 100

Applying generator expressions to scalar functions • 104

Using any() and all() as reductions **108**

Using len() and sum() on collections **110**

Using sums and counts for statistics • 111

Using zip() to structure and flatten sequences **116**

Unzipping a zipped sequence • 117

Flattening sequences • 118

Structuring flat sequences • 120

Structuring flat sequences – an alternative approach • 124

Using sorted() and reversed() to change the order **125**

Using enumerate() to include a sequence number **127**

Summary **128**

Exercises **129**

Palindromic numbers • 129

Hands of cards • 130

Replace legs() with pairwise() • 131

Expand legs() to include processing • 132

Chapter 5: Higher-Order Functions **133**

Using max() and min() to find extrema **134**

Using Python lambda forms • 138

Lambdas and the lambda calculus • 140

Using the map() function to apply a function to a collection **141**

Working with lambda forms and map() • 143

Using map() with multiple sequences • 144

Using the filter() function to pass or reject data **146**

Using filter() to identify outliers • 149

The iter() function with a sentinel value **150**

Using sorted() to put data in order **151**

Overview of writing higher-order functions **153**

Writing higher-order mappings and filters **153**

Unwrapping data while mapping • 156

Wrapping additional data while mapping • 159

Flattening data while mapping • 161

Structuring data while filtering • 164

Building higher-order functions with callables **167**

Assuring good functional design • 168

Review of some design patterns **171**

Summary **172**

Exercises **173**

Classification of state • 173

Classification of state, Part II • 174

Optimizing a file parser • 175

Chapter 6: Recursions and Reductions **177**

Simple numerical recursions **178**

Implementing manual tail-call optimization • 180

Leaving recursion in place • 181

Handling difficult tail-call optimization • 182

Processing collections through recursion • 184

Tail-call optimization for collections • 185

Using the assignment (sometimes called the "walrus") operator in recursions • 187

Reductions and folding a collection from many items to one **188**

Tail-call optimization using deques • 190

Group-by reduction from many items to fewer **193**

Building a mapping with Counter • 195

Building a mapping by sorting • 196

Grouping or partitioning data by key values • 198

Writing more general group-by reductions • 203

Writing higher-order reductions • 205

Writing file parsers • 207

Parsing CSV files • 211

Parsing plain text files with headers • 214

Summary **218**

Exercises **218**

Multiple recursion and caching • 219

Refactor the all_print() function • 219

Parsing CSV files • 219

Classification of state, Part III • 220

Diesel engine data • 220

Chapter 7: Complex Stateless Objects **223**

Using tuples to collect data **224**

Using NamedTuple to collect data **227**
Using frozen dataclasses to collect data **233**
Complicated object initialization and property computations **237**
Using pyrsistent to collect data **240**
Avoiding stateful classes by using families of tuples **246**
Computing Spearman's rank-order correlation • 252
Polymorphism and type pattern matching **257**
Summary **261**
Exercises **262**
Frozen dictionaries • 263
Dictionary-like sequences • 264
Revise the rank_xy() function to use native types • 265
Revise the rank_corr() function • 265
Revise the legs() function to use pyrsistent • 265

Chapter 8: The Itertools Module **267**

Working with the infinite iterators **269**
Counting with count() • 269
Counting with float arguments • 271
Re-iterating a cycle with cycle() • 275
Using cycle() for data sampling • 276
Repeating a single value with repeat() • 278
Using the finite iterators **280**
Assigning numbers with enumerate() • 281
Running totals with accumulate() • 284
Combining iterators with chain() • 286
Partitioning an iterator with groupby() • 287
Merging iterables with zip_longest() and zip() • 290
Creating pairs with pairwise()) • 290
Filtering with compress() • 291

Picking subsets with islice() • 293
Stateful filtering with dropwhile() and takewhile() • 295
Two approaches to filtering with filterfalse() and filter() • 297
Applying a function to data via starmap() and map() • 298
Cloning iterators with tee() **301**
The itertools recipes **302**
Summary **304**
Exercises **305**
Optimize the find_first() function • 306
Compare Chapter 4 with the itertools.pairwise() recipe • 306
Compare Chapter 4 with itertools.tee() recipe • 307
Splitting a dataset for training and testing purposes • 307
Rank ordering • 307

Chapter 9: Itertools for Combinatorics – Permutations and Combinations **309**

Enumerating the Cartesian product **310**
Reducing a product **311**
Computing distances • 314
Getting all pixels and all colors • 318
Performance improvements **320**
Rearranging the problem • 323
Combining two transformations • 324
Permuting a collection of values **326**
Generating all combinations **329**
Combinations with replacement • 333
Recipes **334**
Summary **335**
Exercises **336**
Alternative distance computations • 336

Actual domain of pixel color values • 338
Cribbage hand scoring • 339

Chapter 10: The Functools Module **343**

Function tools **344**
Memoizing previous results with cache **345**
Defining classes with total ordering **348**
Applying partial arguments with partial() **351**
Reducing sets of data with the reduce() function **353**
Combining map() and reduce() • 356
Using the reduce() and partial() functions • 358
Using the map() and reduce() functions to sanitize raw data • 360
Using the groupby() and reduce() functions • 362
Avoiding problems with reduce() • 366
Handling multiple types with singledispatch **366**
Summary **368**
Exercises **370**
Compare string.join() and reduce() • 370
Extend the comma_fix() function • 370
Revise the clean_sum() function • 371

Chapter 11: The Toolz Package **373**

The itertools star map function **374**
Reducing with operator module functions **377**
Using the toolz package **379**
Some itertoolz functions • 379
Some dicttoolz functions • 385
Some functoolz functions • 386
Summary **390**

Exercises **391**

Replace true division with a fraction • 391

Color file parsing • 391

Anscombe's quartet parsing • 392

Waypoint computations • 393

Waypoint geofence • 393

Callable object for the row_counter() function • 394

Chapter 12: Decorator Design Techniques 397

Decorators as higher-order functions **398**

Using the functools update_wrapper() function • 403

Cross-cutting concerns **404**

Composite design **405**

Preprocessing bad data • 407

Adding a parameter to a decorator **410**

Implementing more complex decorators **414**

Complicated design considerations **415**

Summary **420**

Exercises **421**

Datetime conversions • 421

Optimize a decorator • 423

None-tolerant functions • 423

Logging • 424

Dry-run check • 425

Chapter 13: The PyMonad Library 427

Downloading and installing **428**

Functional composition and currying **428**

Using curried higher-order functions • 431

Functional composition with PyMonad • 432

Functors – making everything a function **434**

Using the lazy ListMonad() monad • 436

Monad bind() function **440**

Implementing simulation with monads **441**

Additional PyMonad features **447**

Summary **448**

Exercises **449**

Revise the arctangent series • 449

Statistical computations • 450

Data validation • 450

Multiple models • 451

Chapter 14: The Multiprocessing, Threading, and Concurrent.Futures Modules **453**

Functional programming and concurrency **454**

What concurrency really means **455**

The boundary conditions • 455

Sharing resources with process or threads • 456

Where benefits will accrue • 457

Using multiprocessing pools and tasks **459**

Processing many large files • 460

Parsing log files – gathering the rows • 462

Parsing log lines into named tuples • 463

Parsing additional fields of an Access object • 467

Filtering the access details • 471

Analyzing the access details • 472

The complete analysis process • 473

Using a multiprocessing pool for concurrent processing **474**

Using apply() to make a single request • 478

More complex multiprocessing architectures • 479

Using the concurrent.futures module • 479
Using concurrent.futures thread pools • 480
Using the threading and queue modules • 481
Using async functions • 482
Designing concurrent processing • 483
Summary **486**
Exercises **487**
Lazy parsing • 487
Filter access path details • 488
Add @cache decorators • 489
Create sample data • 489
Change the pipeline structure • 490

Chapter 15: A Functional Approach to Web Services **491**

The HTTP request-response model **492**
Injecting state through cookies • 495
Considering a server with a functional design • 496
Looking more deeply into the functional view • 497
Nesting the services • 498
The WSGI standard **499**
Raising exceptions during WSGI processing • 503
Pragmatic web applications • 505
Defining web services as functions **506**
Flask application processing • 508
The data access tier • 513
Applying a filter • 515
Serializing the results • 516
Serializing data with JSON or CSV formats • 518
Serializing data with XML and HTML • 519
Tracking usage **520**

Summary 523
Exercises 524
WSGI application: welcome • 524
WSGI application: demo • 524
Serializing data with XML • 525
Serializing data with HTML • 525

Other Books You Might Enjoy 529

Index 533

Preface

Functional programming offers a variety of techniques for creating succinct and expressive software. While Python is not a purely functional programming language, we can do a great deal of functional programming in Python.

Python has a core set of functional programming features. This lets us borrow many design patterns and techniques from other functional languages. These borrowed concepts can lead us to create elegant programs. Python's generator expressions, in particular, negate the need to create large in-memory data structures, leading to programs that may execute more quickly because they use fewer resources.

We can't easily create purely functional programs in Python. Python lacks a number of features that would be required for this. We don't have unlimited recursion, for example, we don't have lazy evaluation of all expressions, and we don't have an optimizing compiler.

There are several key features of functional programming languages that are available in Python. One of the most important ones is the idea of functions being first-class objects. Python also offers a number of higher-order functions. The built-in `map()`, `filter()`, and `functools.reduce()` functions are widely used in this role, and less obvious are functions such as `sorted()`, `min()`, and `max()`.

In some cases, a functional approach to a problem will also lead to extremely high-performance algorithms. Python makes it too easy to create large intermediate data structures, tying up memory (and processor time). With functional programming design patterns, we can often replace large lists with generator expressions that are equally expressive but take up much less memory and run much more quickly.

We'll look at the core features of functional programming from a Python point of view. Our objective is to borrow good ideas from functional programming languages and use those ideas to create expressive and succinct applications in Python.

Who this book is for

This book is for more experienced programmers who want to create succinct, expressive Python programs by borrowing techniques and design patterns from functional programming languages. Some algorithms can be expressed elegantly in a functional style; we can—and should—adapt this to make Python programs more readable and maintainable.

This is not intended as a tutorial on Python. This book assumes some familiarity with the language and the standard library. For a foundational introduction to Python, consider *Learn Python Programming, Third Edition*: `https://www.packtpub.com/product/learn-python-programming-third-edition/9781801815093`.

While we cover the foundations of functional programming, this is not a complete review of the various kinds of functional programming techniques. Having an exposure to functional programming in another language can be helpful.

What this book covers

We can decompose this book into two general kinds of topics:

- Essentials of functional programming in Python. This is the content of *Chapters 1* through *7*.
- Library modules to help create functional programs. This is the subject of the remaining chapters of the book. *Chapter 12* includes both fundamental language and library topics.

Chapter 1, *Understanding Functional Programming*, introduces some of the techniques that characterize functional programming. We'll identify some of the ways to map those features to Python. We'll also address some ways that the benefits of functional programming accrue when we use these design patterns to build Python applications.

Chapter 2, *Introducing Essential Functional Concepts*, delves into central features of the functional programming paradigm. We'll look at each in some detail to see how they're implemented in Python. We'll also point out some features of functional languages that don't apply well to Python. In particular, many functional languages have complex type-matching rules required to support compiling and optimizing.

Chapter 3, *Functions, Iterators, and Generators*, will show how to leverage immutable Python objects, and how generator expressions adapt functional programming concepts to the Python language. We'll look at some of the built-in Python collections and how we can leverage them without departing too far from functional programming concepts.

Chapter 4, *Working with Collections*, shows how you can use a number of built-in Python functions to operate on collections of data. This chapter will focus on a number of relatively simple functions, such as `any()` and `all()`, which reduce a collection of values to a single result.

Chapter 5, *Higher-Order Functions*, examines the commonly used higher-order functions such as `map()` and `filter()`. It also shows a number of other higher-order functions as well as how we can create our own functions that work with functions or return functions.

Chapter 6, *Recursions and Reductions*, teaches how to design an algorithm using recursion and then optimize it into a high-performance `for` statement. We'll also look at some other reductions that are widely used, including `collections.Counter()`.

Chapter 7, *Complex Stateless Objects*, showcases a number of ways that we can use immutable tuples, `typing.NamedTuple`, and the frozen `@dataclass` instead of stateful objects. We'll also look at the `pyrsistent` module as a way to create immutable objects. Immutable objects have a simpler interface than stateful objects: we never have to worry about abusing an attribute and setting an object into some inconsistent or invalid state.

Chapter 8, *The Itertools Module*, examines a number of functions in the `itertools` standard library module. This collection of functions simplifies writing programs that deal with collections or generator functions.

Chapter 9, *Itertools for Combinatorics – Permutations and Combinations*, covers the combinatoric functions in the `itertools` module. These functions are more specialized than those in the previous chapter. This chapter includes some examples that illustrate ill-considered use of these functions and the consequences of combinatoric explosion.

Chapter 10, *The Functools Module*, focuses on how to use some of the functions in the `functools` module for functional programming. A few functions in this module are more appropriate for building decorators, and they are left for *Chapter 12*, *Decorator Design Techniques*.

Chapter 11, *The Toolz Package*, covers the `toolz` package, a number of closely related modules that help us write functional programs in Python. The `toolz` modules parallel the built-in `itertools` and `functools` modules, providing alternatives that are often more sophisticated and make better use of curried functions.

Chapter 12, *Decorator Design Techniques*, covers how we can look at a decorator as a way to build a composite function. While there is considerable flexibility here, there are also some conceptual limitations: we'll look at ways that overly complex decorators can become confusing rather than helpful.

Chapter 13, *The PyMonad Library*, examines some of the features of the PyMonad library. This provides some additional functional programming features. It also provides a way to learn more about monads. In some functional languages, monads are an important way to force a particular order for operations that might get optimized into an undesirable order. Since Python already has strict ordering of expressions and statements, the monad feature is more instructive than practical.

Chapter 14, *The Multiprocessing, Threading, and Concurrent.Futures Modules*, points out an important consequence of good functional design: we can distribute the processing workload. Using immutable objects means that we can't corrupt an object because of poorly synchronized write operations.

Chapter 15, *A Functional Approach to Web Services*, shows how we can think of web services as a nested collection of functions that transform a request into a reply. We'll see ways to

leverage functional programming concepts for building responsive, dynamic web content.

Chapter 16, A Chi-Squared Case Study, is a bonus, online-only case study applying a number of functional programming techniques to a specific exploratory data analysis problem. We will apply a χ^2 statistical test to some complex data to see if the results show ordinary variability, or if they are an indication of something that requires deeper analysis. You can find the case study here: `https://github.com/PacktPublishing/Functional-Python-Programming-3rd-Edition/blob/main/Bonus_Content/Chapter_16.pdf`.

To get the most out of this book

This book presumes some familiarity with Python 3 and general concepts of application development. We won't look deeply at subtle or complex features of Python; we'll avoid much consideration of the internals of the language.

Some of the examples use **exploratory data analysis** (**EDA**) as a problem domain to show the value of functional programming. Some familiarity with basic probability and statistics will help with this. There are only a few examples that move into more serious data science.

Python 3.10 is required. The examples have also been tested with Python 3.11, and work correctly. For data science purposes, it's often helpful to start with the **conda** tool to create and manage virtual environments. It's not required, however, and readers should be able to use any available Python.

Additional packages are generally installed with pip. The command looks like this:

```
% python -m pip install toolz pymonad pyrsistent beautifulsoup4
```

Complete the exercises

Each chapter includes a number of exercises that help the reader apply the concepts in the chapter to real code. Most of the exercises are based on code available from the book's repository on GitHub: `https://github.com/PacktPublishing/Functional-Python-Programming-3rd-Edition`.

In some cases, the reader will notice that the code provided on GitHub includes partial solutions to some of the exercises. These serve as hints, allowing the reader to explore alternative solutions.

In many cases, exercises will need unit test cases to confirm they actually solve the problem. These are often identical to the unit test cases already provided in the GitHub repository. The reader should replace the book's example function name with their own solution to confirm that it works.

In some cases, the exercises suggest writing a response document to compare and contrast multiple solutions. It helps to find a mentor or expert who can help the reader by reviewing these small documents for clarity and completeness. A good comparison between design approaches will include performance measurements using the `timeit` module to show the performance advantages of one design over another.

Download the example code files

The code bundle for the book is hosted on GitHub at `https://github.com/PacktPublishing/Functional-Python-Programming-3rd-Edition`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://packt.link/OV1CB`.

Conventions used

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

`CodeInText`: Indicates code words in the text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: "Python has other statements, such as `global` or `nonlocal`, which modify the rules for variables in a particular namespace."

Bold: Indicates a new term, an important word, or words you see on the screen, such as in menus or dialog boxes. For example: "The **base case** states that the sum of a zero-length sequence is 0. The **recursive case** states that the sum of a sequence is the first value plus the sum of the rest of the sequence."

A block of code is set as follows:

```
print("Hello, World!")
```

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

```
% conda create -n functional3 python=3.10
```

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

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

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book we would be grateful if you would report this to us. Please visit `https://subscription.packtpub.com/help`, click on the **Submit Errata** button, search for your book, and enter the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet,

we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

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

Share your thoughts

Once you've read *Functional Python Programming, Third Edition*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback.

`https://packt.link/r/1803232579`

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

Download a free PDF copy of this book

Thanks for purchasing this book!

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

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

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

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

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/9781803232577`

2. Submit your proof of purchase

3. That's it! We'll send your free PDF and other benefits to your email directly

1

Understanding Functional Programming

Functional programming defines a computation using expressions and evaluation; often, they are encapsulated in function definitions. It de-emphasizes or avoids the complexity of state change and mutable objects. This tends to create programs that are more succinct and expressive. In this chapter, we'll introduce some of the techniques that characterize functional programming. We'll identify some of the ways to map these features to **Python**. Finally, we'll also address some ways in which the benefits of functional programming accrue when we use these design patterns to build Python applications.

This book doesn't contain a tutorial introduction to the Python language. We assume the reader knows some Python. In many cases, if the reader knows a functional programming language, then that knowledge can be applied to Python via the examples in this book. For background information on Python, see *Python in a Nutshell, 4th Edition*, or any of the Python introductions from Packt Publishing.

Python has a broad variety of programming features, including numerous ways to support functional programming. As we will see throughout this book, Python is not a *purely* functional programming language; instead, it relies on a mixture of features. We'll see that the language offers enough of the right kinds of features to provide the benefits of functional programming. It also retains all the optimization power of an imperative programming language. Further, we can mix the object-oriented and functional features to make use of the best aspects of both paradigms.

We'll also look at a problem domain that we'll use for many of the examples in this book. We'll try to stick closely to **Exploratory Data Analysis** (**EDA**). For more information, see `https://www.itl.nist.gov/div898/handbook/eda/eda.htm`. The idea of "exploratory" means doing data collection followed by analysis, with a goal of inferring what model would be appropriate to describe the data. This is a helpful domain because many of the algorithms are good examples of functional programming. Furthermore, the benefits of functional programming accrue rapidly when exploring data to locate trends and relationships.

Our goal is to establish some essential principles of functional programming. The more serious Python code will begin in *Chapter 2*, *Introducing Essential Functional Concepts*.

In this chapter, we'll focus on the following topics:

- Comparing and contrasting the functional paradigm with other ways of designing software. We'll look at how Python's approach can be called a "hybrid" between functional programming and object-oriented programming.
- We'll look in depth at a specific example extracted from the functional programming literature.
- We'll conclude with an overview of EDA and why this discipline seems to provide numerous examples of functional programming.

We'll focus on Python 3.10 features in this book. This includes the new `match` statement.

Throughout this book, we'll include Python 3 type hints in the examples. Type hints can help a reader visualize the essential purpose behind a function definition. Type hints are analyzed with the **mypy** tool. As with unit testing, **mypy** can be part of a tool chain to produce high-quality software.

The functional style of programming

We'll define functional programming through a series of examples. The distinguishing feature between these examples is the concept of **state**, specifically the state of the computation.

Python's strong imperative traits mean that the state of a computation is defined by the values of the variables in the various namespaces. Some kinds of statements make a well-defined change to the state by adding, changing, or removing a variable. We call this **imperative** because specific kinds of statements change the state.

In Python, the assignment statement is the primary way to change the state. Python has other statements, such as `global` or `nonlocal`, which modify the rules for variables in a particular namespace. Statements such as `def`, `class`, and `import` change the processing context. The bulk of the remaining statements provide ways to choose which assignment statements get executed. The focus of all these various statement types, however, is on changing the state of the variables.

In a functional language, we replace the state—the changing values of variables—with a simpler notion of evaluating functions. Each function evaluation creates a new object or objects from existing objects. Since a functional program is a composition of functions, we can design lower-level functions that are easy to understand, and then create compositions of functions that can also be easier to visualize than a complex sequence of statements.

Function evaluation more closely parallels mathematical formalisms. Because of this, we can often use simple algebra to design an algorithm that clearly handles the edge cases and boundary conditions. This makes us more confident that the functions work. It also makes

it easy to locate test cases for formal unit testing.

It's important to note that functional programs tend to be relatively succinct, expressive, and efficient compared to imperative (object-oriented or procedural) programs. The benefit isn't automatic; it requires careful design. This design effort for functional programming is often smaller than for procedural programming. Some developers experienced in imperative and object-oriented styles may find it a challenge to shift their focus from stateful designs to functional designs.

Comparing and contrasting procedural and functional styles

We'll use a tiny example program to illustrate a non-functional, or procedural, style of programming. This example computes a sum of a sequence of numbers. Each of the numbers has a specific property that makes it part of the sequence.

```
def sum_numeric(limit: int = 10) -> int:
    s = 0
    for n in range(1, limit):
        if n % 3 == 0 or n % 5 == 0:
            s += n
    return s
```

The sum computed by this function includes only numbers that are multiples of 3 or 5. We've made this program strictly procedural, avoiding any explicit use of Python's object features. The function's state is defined by the values of the variables `s` and `n`. The variable `n` takes on values such that $1 \le n < 10$. As the iteration involves an ordered exploration of values for the `n` variable, we can prove that it will terminate when the value of `n` is equal to the value of `limit`.

There are two explicit assignment statements, both setting values for the `s` variable. These state changes are visible. The value of `n` is set implicitly by the `for` statement. The state change in the `s` variable is an essential element of the state of the computation.

Now let's look at this again from a purely functional perspective. Then, we'll examine a more Pythonic perspective that retains the essence of a functional approach while leveraging a number of Python's features.

Using the functional paradigm

In a functional sense, the sum of the multiples of 3 and 5 can be decomposed into two parts:

- The sum of a sequence of numbers
- A sequence of values that pass a simple test condition, for example, being multiples of 3 and 5

To be super formal, we can define the sum as a function using simpler language components. The sum of a sequence has a recursive definition:

```
from collections.abc import Sequence
def sumr(seq : Sequence[int]) -> int:
    if len(seq) == 0:
        return 0
    return seq[0] + sumr(seq[1:])
```

We've defined the sum in two cases. The **base case** states that the sum of a zero-length sequence is 0. The **recursive case** states that the sum of a sequence is the first value plus the sum of the rest of the sequence. Since the recursive definition depends on a shorter sequence, we can be sure that it will (eventually) devolve to the base case.

Here are some examples of how this function works:

```
>>> sumr([7, 11])
18
>>> sumr([11])
11
>>> sumr([])
0
```

The first example computes the sum of a list with multiple items. The second example

shows how the recursion rule works by adding the first item, `seq[0]`, to the sum of the remaining items, `sumr(seq[1:])`. Eventually, the computation of the result involves the sum of an empty list, which is defined as 0.

The `+` operator on the last line of the `sumr` function and the initial value of 0 in the base case characterize the equation as a sum. Consider what would happen if we changed the operator to `*` and the initial value to 1: this new expression would compute a product. We'll return to this simple idea of **generalization** in the following chapters.

Similarly, generating a sequence of values with a given property can have a recursive definition, as follows:

```
from collections.abc import Sequence, Callable
def until(
        limit: int,
        filter_func: Callable[[int], bool],
        v: int
) -> list[int]:
    if v == limit:
        return []
    elif filter_func(v):
        return [v] + until(limit, filter_func, v + 1)
    else:
        return until(limit, filter_func, v + 1)
```

In this function, we've compared a given value, `v`, against the upper bound, `limit`. If `v` has reached the upper bound, the resulting list must be empty. This is the base case for the given recursion.

There are two more cases defined by an externally defined `filter_func()` function. The value of `v` is passed by the `filter_func()` function; if this returns a very small list, containing one element, this can be concatenated with any remaining values computed by the `until()` function.

If the value of `v` is rejected by the `filter_func()` function, this value is ignored and the result is simply defined by any remaining values computed by the `until()` function.

We can see that the value of `v` will increase from an initial value until it reaches `limit`, assuring us that we'll reach the base case.

Before we can see how to use the `until()` function, we'll define a small function to filter values that are multiples of 3 or 5:

```
def mult_3_5(x: int) -> bool:
    return x % 3 == 0 or x % 5 == 0
```

We could also have defined this as a lambda object to emphasize succinct definitions of simple functions. Anything more complex than a one-line expression requires the `def` statement.

This function can be combined with the `until()` function to generate a sequence of values, which are multiples of 3 and 5. Here's an example:

```
>>> until(10, mult_3_5, 0)
[0, 3, 5, 6, 9]
```

Looking back at the decomposition at the top of this section, we now have a way to compute sums and a way to compute the sequence of values.

We can combine the `sumr()` and `until()` functions to compute a sum of values. Here's the resulting code:

```
def sum_functional(limit: int = 10) -> int:
    return sumr(until(limit, mult_3_5, 0))
```

This small application to compute a sum doesn't make use of the assignment statement to set the values of variables. It is a purely functional, recursive definition that matches the mathematical abstractions, making it easier to reason about. We can be confident each piece works separately, giving confidence in the whole.

As a practical matter, we'll use a number of Python features to simplify creating functional

programs. We'll take a look at a number of these optimizations in the next version of this example.

Using a functional hybrid

We'll continue this example with a mostly functional version of the previous example to compute the sum of multiples of 3 and 5. Our **hybrid** functional version might look like the following:

```
def sum_hybrid(limit: int = 10) -> int:
    return sum(
        n for n in range(1, limit)
        if n % 3 == 0 or n % 5 == 0
    )
```

We've used a generator expression to iterate through a collection of values and compute the sum of these values. The `range(1, 10)` object is an iterable; it generates a sequence of values $\{n \mid 1 \le n < 10\}$, often summarized as "values of n such that 1 is less than or equal to n and n is less than 10." The more complex expression `n for n in range(1, 10) if n % 3 == 0 or n % 5 == 0` is also a generator. It produces a set of values, $\{n \mid 1 \le n < 10 \land (n \equiv 0 \mod 3 \lor n \equiv 0 \mod 5)\}$; something we can describe as "values of n such that 1 is less than or equal to n and n is less than 10 and n is equivalent to 0 modulo 3 or n is equivalent to 0 modulo 5." These are multiples of 3 and 5 taken from the set of values between 1 and 10. The variable `n` is bound, in turn, to each of the values provided by the `range` object. The `sum()` function consumes the iterable values, creating a final object, 23.

The bound variable, `n`, doesn't exist outside the generator expression. The variable `n` isn't visible elsewhere in the program.

The variable `n` in this example isn't directly comparable to the variable `n` in the first two imperative examples. A `for` statement (outside a generator expression) creates a proper variable in the local namespace. The generator expression does not create a variable in the

same way that a `for` statement does:

```
>>> sum(
...     n for n in range(1, 10)
...     if n % 3 == 0 or n % 5 == 0
... )
23
>>> n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

The generator expression doesn't pollute the namespace with variables, like `n`, which aren't relevant outside the very narrow context of the expression. This is a pleasant feature that ensures we won't be confused by the values of variables that don't have a meaning outside a single expression.

The stack of turtles

When we use Python for functional programming, we embark down a path that will involve a hybrid that's not strictly functional. Python is not **Haskell**, **OCaml**, or **Erlang**. For that matter, our underlying processor hardware is not functional; it's not even strictly object-oriented, as CPUs are generally procedural.

> *All programming languages rest on abstractions, libraries, frameworks and virtual machines. These abstractions, in turn, may rely on other abstractions, libraries, frameworks and virtual machines. The most apt metaphor is this: the world is carried on the back of a giant turtle. The turtle stands on the back of another giant turtle. And that turtle, in turn, is standing on the back of yet another turtle.*
>
> *It's turtles all the way down.*
>
> — *Anonymous*

There's no practical end to the layers of abstractions. Even something as concrete as circuits and electronics may be an abstraction to help designers summarize the details of quantum

electrodynamics.

More importantly, the presence of abstractions and virtual machines doesn't materially change our approach to designing software to exploit the functional programming features of Python.

Even within the functional programming community, there are both purer and less pure functional programming languages. Some languages make extensive use of monads to handle stateful things such as file system input and output. Other languages rely on a hybridized environment that's similar to the way we use Python. In Python, software can be generally functional, with carefully chosen procedural exceptions.

Our functional Python programs will rely on the following three stacks of abstractions:

- Our applications will be functions—all the way down—until we hit the objects;
- The underlying Python runtime environment that supports our functional programming is objects—all the way down—until we hit the libraries;
- The libraries that support Python are a turtle on which Python stands.

The operating system and hardware form their own stack of turtles. These details aren't relevant to the problems we're going to solve.

A classic example of functional programming

As part of our introduction, we'll look at a classic example of functional programming. This is based on the paper *Why Functional Programming Matters* by John Hughes. The article appeared in a paper called *Research Topics in Functional Programming*, edited by D. Turner, published by Addison-Wesley in 1990.

Here's a link to one of the papers in *Research Topics in Functional Programming*, "Why Functional Programming Matters": `http://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf`

This paper is a profound discussion of functional programming. There are several examples given. We'll look at just one: the **Newton-Raphson algorithm** for locating any roots of a

function. In this case, we'll define a function that will compute a square root of a number.

It's important because many versions of this algorithm rely on the explicit state managed via loops. Indeed, the Hughes paper provides a snippet of the **Fortran** code that emphasizes stateful, imperative processing.

The backbone of this approximation is the calculation of the next approximation from the current approximation. The `next_()` function takes `x`, an approximation to the `sqrt(n)` value, and calculates a next value that brackets the proper root. Take a look at the following example:

```
def next_(n: float, x: float) -> float:
    return (x + n / x) / 2
```

This function computes a series of values that will quickly converge on some value x such that $x = \frac{n}{x}$, which means $x = \sqrt{n}$.

Note that the name `next()` would collide with a built-in function. Calling it `next_()` lets us follow the original presentation as closely as possible, using Pythonic names.

Here's how the function looks when used in Python's interactive REPL:

```
>>> n = 2
>>> f = lambda x: next_(n, x)
>>> a0 = 1.0
>>> [round(x, 4)
... for x in (a0, f(a0), f(f(a0)), f(f(f(a0))),)
... ]
[1.0, 1.5, 1.4167, 1.4142]
```

We defined the `f()` function as a lambda that will converge on $\sqrt{n}$ where $n = 2$. We started with 1.0 as the initial value for a_0. Then we evaluated a sequence of recursive evaluations: $a_1 = f(a_0)$, $a_2 = f(f(a_0))$, and so on. We evaluated these functions using a

generator expression so that we could round each value to four decimal places. This makes the output easier to read and easier to use with `doctest`. The sequence appears to converge rapidly on $\sqrt{2}$. To get a more precise answer, we must continue to perform the series of steps after the first four shown above.

We can write a function that will (in principle) generate an infinite sequence of a_i values. This series will converge on the proper square root:

```
from collections.abc import Iterator, Callable
def repeat(
        f: Callable[[float], float],
        a: float
) -> Iterator[float]:
    yield a
    yield from repeat(f, f(a))
```

This function will generate a sequence of approximations using a function, `f()`, and an initial value, `a`. If we provide the `next_()` function defined earlier, we'll get a sequence of approximations to the square root of the `n` argument.

The `repeat()` function expects the `f()` function to have a single argument; however, our `next_()` function has two arguments. We've used a lambda object, `lambda x: next_(n, x)`, to create a partial version of the `next_()` function with one of two variables bound.

The Python generator functions can't be trivially recursive; they must explicitly iterate over the recursive results, yielding them individually.

Attempting to use a simple `return repeat(f, f(a))` will end the iteration, returning a generator expression instead of yielding values.

There are two ways to return all the values instead of returning a generator expression, which are as follows:

- We can write an explicit `for` statement to yield values as follows:

```
for x in some_iter: yield x
```

- We can use the `yield from` expression as follows:

```
yield from some_iter
```

Both techniques of yielding the values of a recursive generator function are will have similar results. We'll try to emphasize `yield from`.

It turns out that `yield` and `yield from` are a bit more sophisticated than we've shown here. For our purposes, we'll limit ourselves to working with recursive results. For more information on the full feature set for `yield` and `yield from`, see PEP 342 and PEP 380: `https://peps.python.org/pep-0342/` and `https://peps.python.org/pep-0380/`.

Of course, we don't want the entire infinite sequence created by the `repeat()` function. It's essential to stop generating values when we've found the square root we're looking for. The common symbol for the limit we can consider "close enough" is the Greek letter **epsilon**, ϵ.

In Python, we have to be a little clever when taking items from an infinite sequence one at a time. It works out well to use a simple interface function that wraps a slightly more complex recursion. Take a look at the following code snippet:

```
from collections.abc import Iterator
def within(
        ϵ: float,
        iterable: Iterator[float]
) -> float:
    def head_tail(
            ϵ: float,
            a: float,
```

```
            iterable: Iterator[float]
    ) -> float:
        b = next(iterable)
        if abs(a-b) <= ϵ:
            return b
        return head_tail(ϵ, b, iterable)

    return head_tail(ϵ, next(iterable), iterable)
```

We've defined an internal function, `head_tail()`, which accepts the tolerance, ϵ, an item from the iterable sequence, `a`, and the rest of the iterable sequence, `iterable`. The first item from the iterable, extracted with the `next()` function, is bound to a name, `b`. If $|a - b| \leq \epsilon$, the two values of `a` and `b` are close enough to call the value of `b` the square root; the difference is less than or equal to the very small value of ϵ. Otherwise, we use the `b` value in a recursive invocation of the `head_tail()` function to examine the next pair of values.

Our `within()` function properly initializes the internal `head_tail()` function with the first value from the `iterable` parameter.

We can use the three functions, `next_()`, `repeat()`, and `within()`, to create a square root function, as follows:

```
def sqrt(n: float) -> float:
    return within(
        ϵ=0.0001,
        iterable=repeat(
            lambda x: next_(n, x),
            1.0
        )
    )
```

We've used the `repeat()` function to generate a (potentially) infinite sequence of values based on the `next_(n,x)` function. Our `within()` function will stop generating values in the sequence when it locates two values with a difference less than ϵ.

This definition of the `sqrt()` function provides useful default values to the underlying `within()` function. It provides an ϵ value of 0.0001 and an initial a_0 value of 1.0.

A more advanced version could use default parameter values to make changes possible. As an exercise, the definition of `sqrt()` can be rewritten so an expression such as `sqrt(1.0, 0.000_01, 3)` will start with an approximation of 1.0 and compute the value of $\sqrt{3}$ to within 0.00001. For most applications, the initial a_0 value can be 1.0. However, the closer it is to the actual square root, the more rapidly this algorithm converges.

The original example of this approximation algorithm was shown in the Miranda language. It's easy to see there are some profound differences between Miranda and Python. In spite of the differences, the similarities give us confidence that many kinds of functional programming can be easily implemented in Python.

The `within` function shown here is written to match the original article's function definition. Python's `itertools` library provides a `takewhile()` function that might be better for this application than the `within()` function. Similarly, the `math.isclose()` function may be better than the `abs(a-b) <= ` ϵ expression used here. Python offers a great many pre-built functional programming features; we'll look closely at these functions in *Chapter 8*, *The Itertools Module* and *Chapter 9*, *Itertools for Combinatorics – Permutations and Combinations*.

Exploratory data analysis

Later in this book, we'll use the field of exploratory data analysis as a source for concrete examples of functional programming. This field is rich with algorithms and approaches to working with complex datasets; functional programming is often a very good fit between the problem domain and automated solutions.

While details vary from author to author, there are several widely accepted stages of EDA. These include the following:

- **Data preparation**: This might involve extraction and transformation for source applications. It might involve parsing a source data format and doing some kind of data scrubbing to remove unusable or invalid data. This is an excellent application

of functional design techniques.

David Mertz's superb book *Cleaning Data for Effective Data Science*(`https://www.packtpub.com/product/cleaning-data-for-effective-data-science/9781801071291`) provides additional information on data cleaning. This is a crucial subject for all data science and analytical work.

- **Data exploration**: This is a description of the available data. This usually involves the essential statistical functions. This is another excellent place to explore functional programming. We can describe our focus as univariate and bivariate statistics, but that sounds too daunting and complex. What this really means is that we'll focus on mean, median, mode, and other related descriptive statistics. Data exploration may also involve data visualization. We'll skirt this issue because it doesn't involve very much functional programming.

For more information on Python visualization, see *Interactive Data Visualization with Python*, `https://www.packtpub.com/product/interactive-data-visualization-with-python-second-edition/9781800200944`. See `https://www.projectpro.io/article/python-data-visualization-libraries/543` for some additional visualization libraries.

- **Data modeling and machine learning**: This tends to be prescriptive as it involves extending a model to new data. We're going to skirt around this because some of the models can become mathematically complex. If we spend too much time on these topics, we won't be able to focus on functional programming.
- **Evaluation and comparison**: When there are alternative models, each must be evaluated to determine which is a better fit for the available data. This can involve ordinary descriptive statistics of model outputs, which can benefit from functional design techniques.

One goal of EDA is often to create a model that can be deployed as a decision support

application. In many cases, a model might be a simple function. A functional programming approach can apply the model to new data and display results for human consumption.

Summary

In this chapter, we've looked at programming paradigms with an eye toward distinguishing the functional paradigm from the imperative paradigm. For our purposes, object-oriented programming is a kind of imperative programming; it relies on explicit state changes. Our objective in this book is to explore the functional programming features of Python. We've noted that some parts of Python don't allow purely functional programming; we'll be using some hybrid techniques that meld the good features of succinct, expressive functional programming with some high-performance optimizations in Python.

In the next chapter, we'll look at five specific functional programming techniques in detail. These techniques will form the essential foundation for our hybridized functional programming in Python.

Exercises

The exercises in this book are based on code available from Packt Publishing on GitHub. See `https://github.com/PacktPublishing/Functional-Python-Programming-3rd-Edition`.

In some cases, the reader will notice that the code provided on GitHub includes partial solutions to some of the exercises. These serve as hints, allowing the reader to explore alternative solutions.

In many cases, exercises will need unit test cases to confirm they actually solve the problem. These are often identical to the unit test cases already provided in the GitHub repository. The reader will need to replace the book's example function name with their own solution to confirm that it works.

Convert an imperative algorithm to functional code

The following algorithm is stated as imperative assignment statements and a while construct to indicate processing something iteratively.

Algorithm 1 Imperative iteration

Require: a list of n values, $V = \{v_0, v_1, v_2, ..., v_n\}$, and $n \geq 1$.

1: $s \longleftarrow 0$
2: $i \longleftarrow 0$
3: **while** $i \neq n$ **do**
4: $\quad s \longleftarrow s + v_i$
5: $\quad i \longleftarrow i + 1$
6: **end while**
7: $m \longleftarrow \frac{s}{n}$

What does this appear to compute? Given Python built-in functions like `sum`, can this be simplified?

It helps to write this in Python and refactor the code to be sure that correct answers are created.

A test case is the following:

$$V \longleftarrow \{7.46, 6.77, 12.74, 7.11, 7.81, 8.84, 6.08, 5.39, 8.15, 6.42, 5.73\}$$

The computed value for m is approximately `7.5`.

Convert step-wise computation to functional code

The following algorithm is stated as a long series of single assignment statements. The rad(x) function converts degrees to radians, rad(d) = $\pi \times \frac{d}{180}$. See the `math` module for an implementation.

Algorithm 2 Imperative computation

1: $rlat_1 \longleftarrow \text{rad}(lat_1)$
2: $rlat_2 \longleftarrow \text{rad}(lat_2)$
3: $dlat \longleftarrow rlat_1 - rlat_2$
4: $rlon_1 \longleftarrow \text{rad}(lon_1)$
5: $rlon_2 \longleftarrow \text{rad}(lon_2)$
6: $dlon \longleftarrow rlon_1 - rlon_2$
7: $lat_m \longleftarrow rlat_1 + rlat_2$
8: $lat_m \longleftarrow \frac{lat_m}{2}$
9: $c \longleftarrow \cos lat_m$
10: $x \longleftarrow R \times dlon$
11: $x \longleftarrow x \times c$
12: $y \longleftarrow R \times dlat$
13: $x_2 \longleftarrow x^2$
14: $y_2 \longleftarrow y^2$
15: $xy_2 \longleftarrow x_2 + y_2$
16: $d \longleftarrow \sqrt{xy_2}$

Is this code easy to understand? Can you summarize this computation as a short mathematical-looking formula?

Breaking it down into sections, lines 1 to 8 seem to be focused on some conversions, differences, and mid-point computations. Lines 9 to 12 compute two values, x and y. Can these be summarized or simplified? The final four lines do a relatively direct computation of d. Can this be summarized or simplified? As a hint, look at `math.hypot()` for a function that might be applicable in this case.

It helps to write this in Python and refactor the code.

A test case is the following:

$lat_1 \longleftarrow 32.82950$

$lon_1 \leftarrow -79.93021$

$lat_2 \leftarrow 32.74412$

$lon_2 \leftarrow -79.85226$

The computed value for d is approximately `6.4577`.

Refactoring the code can help to confirm your understanding.

Revise the sqrt() function

The `sqrt()` function defined in the *A classic example of functional programming* section has only a single parameter value, `n`. Rewrite this to create a more advanced version using default parameter values to make changes possible. An expression such as `sqrt(1.0, 0.000_01, 3)` will start with an approximation of 1.0 and compute the value to a precision of 0.00001. The final parameter value, `3`, is the value of n, the number we need to compute the square root of.

Data cleansing steps

A file of source data has US ZIP codes in a variety of formats. This problem often arises when spreadsheet software is used to collect or transform data.

- Some ZIP codes were processed as numbers. This doesn't work out well for places in New England, where ZIP codes have a leading zero. For example, one of Portsmouth, New Hampshire's codes should be stated as `03801`. In the source file, it is `3801`. For the most part, these numbers will have five or nine digits, but some codes in New England will be four or eight digits when a single leading zero was dropped. For Puerto Rico, there may be two leading zeroes.
- Some ZIP codes are stored as strings, 12345 – 0100, where a four-digit extension for a post-office box has been appended to the base five-digit code.

A CSV-format file has only text values. However, when data in the file has been processed by a spreadsheet, problems can arise. Because a ZIP code has only digits, it can be treated as numeric data. This means the original data values will have been converted to a number,

and then back to a text representation. These conversions will drop the leading zeroes. There are a number of workarounds in various spreadsheet applications to prevent this problem. If they're not used, the data can have anomalous values that can be cleansed to restore the original representation.

The objective of the exercise is to compute a histogram of the most popular ZIP codes in the source data file. The data must be cleansed to have the following two ZIP formats:

- Five characters with no post-office box, for example `03801`
- Ten characters with a hyphen, for example `03899-9876`

The essential histogram can be done with a `collections.Counter` object as follows.

```
from collections import Counter
import csv
from pathlib import Path

DEFAULT_PATH = Path.cwd() / "address.csv"

def main(source_path: Path = DEFAULT_PATH) -> None:
    frequency: Counter[str] = Counter()
    with source_path.open() as source:
        rdr = csv.DictReader(source)
        for row in rdr:
            if "-" in row['ZIP']:
                text_zip = row['ZIP']
                missing_zeroes = 10 - len(text_zip)
                if missing_zeroes:
                    text_zip = missing_zeroes*'0' + text_zip
            else:
                text_zip = row['ZIP']
                if 5 < len(row['ZIP']) < 9:
                    missing_zeroes = 9 - len(text_zip)
                else:
                    missing_zeroes = 5 - len(text_zip)
                if missing_zeroes:
                    text_zip = missing_zeroes*'0' + text_zip
```

```
            frequency[text_zip] += 1
    print(frequency)

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

This makes use of imperative processing features to read a file. The overall design, using a `for` statement to process rows of a file, is an essential Pythonic feature that we can preserve.

On the other hand, the processing of the `text_zip` and `missing_zeroes` variables through a number of state changes seems like it's a potential source for confusion.

This can be refactored through several rewrites:

1. Decompose the `main()` function into two parts. A new `zip_histogram()` function should be written to contain much of the processing detail. This function will process the opened file, and return a `Counter` object. A suggested signature is the following:

    ```
    def zip_histogram(
            reader: csv.DictReader[str]) -> Counter[str]:
        pass
    ```

    The `main()` function is left with the responsibility to open the file, create the `csv.DictReader` instance, evaluate `zip_histogram()`, and print the histogram.

2. Once the `zip_histogram()` function has been defined, the cleansing of the `ZIP` attribute can be refactored into a separate function, with a name like `zip_cleanse()`. Rather than setting the value of the `text_zip` variable, this function can return the cleansed result. This can be tested separately to be sure the various cases are handled gracefully.

3. The distinction between long ZIP codes with a hyphen and without a hyphen is something that should be fixed. Once the `zip_cleanse()` works in general, add a new

function to inject hyphens into ZIP codes with only digits. This should transform `38011234` to `03801-1234`. Note that short, five-digit ZIP codes do not need to have a hyphen added; this additional transformation only applies to nine-digit codes to make them into ten-position strings.

The final `zip_histogram()` function should look something like the following:

```
def zip_histogram(
        reader: csv.DictReader[str]) -> Counter[str]:
    return Counter(
        zip_cleanse(
            row['ZIP']
        ) for row in reader
    )
```

This provides a framework for performing a focused data cleanup in the given column. It allows us to distinguish between CSV and file processing features, and the details of how to clean up a specific column of data.

(Advanced) Optimize this functional code

The following algorithm is stated as a single "step" that has been decomposed into three separate formulae. The decomposition is more a concession to the need to fit the expression into the limits of a printed page than a useful optimization. The rad(x) function converts degrees to radians, $\text{rad}(d) = \pi \times \frac{d}{180}$.

Algorithm 3 Redundant expressions

$$x = \left[R \times (\text{rad}(lon_1) - \text{rad}(lon_2)) \times \cos\left(\frac{\text{rad}(lat_1) + \text{rad}(lat_2)}{2}\right)\right]^2 \tag{1.1}$$

$$y = [R \times (\text{rad}(lat_1) - \text{rad}(lat_2))]^2 \tag{1.2}$$

$$d \longleftarrow \sqrt{x + y} \tag{1.3}$$

There are a number of redundant expressions, like rad(lat_1) and rad(lat_2). If these are assigned to local variables, can the expression be simplified?

The final computation of d does not match the conventional understanding of computing a hypotenuse, $\sqrt{x^2 + y^2}$. Should the code be refactored to match the definition in `math.hypot`?

It helps to start by writing this in Python and then refactoring the code.

A test case is the following:

$lat_1 \leftarrow 32.82950$

$lon_1 \leftarrow -79.93021$

$lat_2 \leftarrow 32.74412$

$lon_2 \leftarrow -79.85226$

The computed value for d is approximately `6.4577`.

Refactoring the code can help to confirm your understanding of what this code really does.

2

Introducing Essential Functional Concepts

Most of the features of functional programming are already part of the Python language. Our goal in writing functional Python is to shift our focus away from imperative (procedural or object-oriented) techniques as much as possible.

We'll look at the following functional programming topics:

- In Python, functions are first-class objects.
- We can use and create higher-order functions.
- We can create pure functions very easily.
- We can work with immutable data.
- In a limited way, we can create functions that have non-strict evaluation of sub-expressions. Python generally evaluates expressions strictly. As we'll see later, a few operators are non-strict.
- We can design functions that exploit eager versus lazy evaluation.

- We can use recursion instead of an explicit loop state.
- We have a type system that can apply to functions and objects.

This expands on the concepts from the first chapter: firstly, that purely functional programming avoids the complexities of an explicit state maintained through variable assignments; and secondly, that Python is not a purely functional language.

Because Python is not a purely functional language, we'll focus on those features that are indisputably important in functional programming. We'll start by looking at functions as first-class Python objects, with properties and methods of their own.

Functions as first-class objects

Functional programming is often succinct and expressive. One way to achieve this is by providing functions as arguments and return values for other functions. We'll look at numerous examples of manipulating functions.

For this to work, functions must be first-class objects in the runtime environment. In programming languages such as C, a function is not a runtime object; because the compiled C code generally lacks internal attributes and methods, there's little runtime introspection that can be performed on a function. In Python, however, functions are objects that are created (usually) by `def` statements and can be manipulated by other Python functions. We can also create a function as a callable object or by assigning a lambda object to a variable.

Here's how a function definition creates an object with attributes:

```
>>> def example(a, b, **kw):
...     return a*b
...
>>> type(example)
<class 'function'>
>>> example.__code__.co_varnames
('a', 'b', 'kw')
>>> example.__code__.co_argcount
2
```

We've created an object, `example`, that is of the `function` class. This object has numerous attributes. The `__code__` attribute of the function object has attributes of its own. The implementation details aren't important. What is important is functions are first-class objects and can be manipulated like all other objects. The example shows the values of two of the many attributes of a function object.

Pure functions

A function free from the confusion created by side effects is often more expressive than a function that also updates state elsewhere in an application. Using pure functions can also allow some optimizations by changing evaluation order. The big win, however, stems from pure functions being conceptually simpler and much easier to test.

To write a pure function in Python, we have to write local-only code. This means we have to avoid `global` statements. We need to avoid entanglements with objects that have hidden state; often, this means avoiding input and output operations. We need to look closely at any use of `nonlocal`, also. While assigning to a non-local variable is a side effect, the state change is confined to a nested function definition. Avoiding global variables and file operations is an easy standard to meet. Pure functions are a common feature of Python programs.

There isn't a built-in tool to *guarantee* a Python function is free from side effects. For folks interested in the details, a tool like `mr-proper`, `https://pypi.org/project/mr-proper/`, can be used to confirm that a function is pure.

A Python lambda is often used to create a very small, pure function. It's possible for a lambda object to perform input or output or use an impure function. A bit of code inspection is still helpful to remove any doubts.

Here's a function created by assigning a lambda object to a variable:

```
>>> mersenne = lambda x: 2 ** x - 1
>>> mersenne(17)
131071
```

We created a pure function using `lambda` and assigned this to the variable `mersenne`. This is a callable object with a single parameter, `x`, that returns a single value.

The following example shows an impure function defined as a lambda object:

```
>>> default_zip = lambda row: row.setdefault('ZIP', '00000')
```

This function has the potential to update a dictionary in the event the key, `'ZIP'`, is not present. There are two cases, as shown in the following example:

```
>>> r_0 = {'CITY': 'Vaca Key'}
>>> default_zip(r_0)
'00000'
>>> r_0
{'CITY': 'Vaca Key', 'ZIP': '00000'}

>>> r_1 = {'CITY': 'Asheville', 'ZIP': 27891}
>>> default_zip(r_1)
27891
```

In the first case, the dictionary object `r_0` does not have the key, `'ZIP'`. The dictionary object is updated by the lambda object. This is a consequence of using the `setdefault()` method of a dictionary.

In the second case, the `r_1` object contains the key, `'ZIP'`. There's no update to the dictionary. The side effect depends on the state of the object prior to the function, making the function potentially more difficult to understand.

Higher-order functions

We can achieve expressive, succinct programs using higher-order functions. These are functions that accept a function as an argument or return a function as a value. We can use higher-order functions as a way to create composite functions from simpler functions.

Consider the Python `max()` function. We can provide a function as an argument and modify how the `max()` function behaves.

Here's some data we might want to process:

```
>>> year_cheese = [(2000, 29.87), (2001, 30.12),
...     (2002, 30.6), (2003, 30.66), (2004, 31.33),
...     (2005, 32.62), (2006, 32.73), (2007, 33.5),
...     (2008, 32.84), (2009, 33.02), (2010, 32.92)]
```

We can apply the `max()` function, as follows:

```
>>> max(year_cheese)
(2010, 32.92)
```

The default behavior is to simply compare each tuple in the sequence. This will return the tuple with the largest value on position zero of each tuple.

Since the `max()` function is a higher-order function, we can provide another function as an argument. In this case, we'll use a lambda as the function; this is used by the `max()` function, as follows:

```
>>> max(year_cheese, key=lambda yc: yc[1])
(2007, 33.5)
```

In this example, the `max()` function applies the supplied lambda and returns the tuple with the largest value in position one of each tuple.

Python provides a rich collection of higher-order functions. We'll see examples of each

of Python's higher-order functions in later chapters, primarily in *Chapter 5*, *Higher-Order Functions*. We'll also see how we can easily write our own higher-order functions.

Immutable data

Since we're not using variables to track the state of a computation, our focus needs to stay on immutable objects. We can make extensive use of tuples, `typing.NamedTuples`, and frozen `@dataclass` to provide more complex data structures that are also immutable. We'll look at these class definitions in detail in *Chapter 7*, *Complex Stateless Objects*.

The idea of immutable objects is not foreign to Python. Strings and tuples are two widely-used immutable objects. There can be a performance advantage to using immutable tuples instead of more complex mutable objects. In some cases, the benefits come from rethinking the algorithm to avoid the costs of object mutation.

As an example, here's a common design pattern that works well with immutable objects: the `wrapper()` function. A list of tuples is a fairly common data structure. We will often process this list of tuples in one of the two following ways:

- **Using higher-order functions**: As shown earlier, we provided a lambda as an argument to the `max()` function: `max(year_cheese, key=lambda yc: yc[1])`.
- **Using the wrap-process-unwrap pattern**: In a functional context, we can implement this with code that follows an `unwrap(process(wrap(structure)))` pattern.

For example, look at the following command snippet:

```
>>> max(map(lambda yc: (yc[1], yc), year_cheese))[1]
(2007, 33.5)
```

This fits the three-part pattern of wrapping a data structure, finding the maximum of the wrapped structures, and then unwrapping the structure.

The expression `map(lambda yc: (yc[1], yc), year_cheese)` will transform each item into a two-tuple with a key followed by the original item. In this example, the comparison key value is the expression `yc[1]`.

The processing is done using the `max()` function. Since each piece of the source data has been simplified to a new two-tuple, the higher-order function features of the `max()` function aren't required. To make this work, the comparison value was taken from position one of the source record and placed first into the two-tuple. The default behavior of the `max()` function uses the first item in each two-tuple to locate the largest value.

Finally, we unwrap using the subscript expression `[1]`. This will pick the second element of the two-tuple selected by the `max()` function.

This kind of wrap-and-unwrap is so common that some languages have special functions with names like `fst()` and `snd()` that we can use as function prefixes instead of a syntactic suffix of `[0]` or `[1]`. We can use this idea to modify our wrap-process-unwrap example, as follows:

```
>>> snd = lambda x: x[1]
>>> snd(max(map(lambda yc: (yc[1], yc), year_cheese)))
(2007, 33.5)
```

Here, a lambda is used to define the `snd()` function to pick the second item from a tuple. This provides an easier-to-read version of `unwrap(process(wrap()))`. As with the previous example, the `map(lambda... , year_cheese)` expression is used to wrap our raw data items, and the `max()` function does the processing. Finally, the `snd()` function extracts the second item from the tuple.

This can be simplified by using `typing.NamedTuple` or a `@dataclass`. In *Chapter 7*, *Complex Stateless Objects*, we'll look at these two alternatives.

We will—as a general design principle—avoid class definitions. It can seem like anathema to avoid objects in an **Object-Oriented Programming** (**OOP**) language, but we note that functional programming doesn't depend on stateful objects. When we use class definitions, we'll avoid designs that update attribute values.

There are a number of good reasons for using immutable objects. We can, for example, use an object as a named collection of attribute values. Additionally, callable objects can provide

some optimizations, like the caching of computed results. Caching is important because Python doesn't have an optimizing compiler. Another reason for using class definitions is to provide a namespace for closely related functions.

Strict and non-strict evaluation

Functional programming's efficiency stems, in part, from being able to defer a computation until it's required. There are two similar concepts for avoiding computation. These are:

- **Strictness**: Python operators are generally strict and evaluate all sub-expressions from left to right. This means an expression like `f(a)+f(b)+f(c)` is evaluated as if it was `(f(a)+f(b))+f(c)`. An optimizing compiler might avoid strict ordering to improve performance. Python doesn't optimize and code is mostly strict. We'll look at cases where Python is not strict below.
- **Eagerness** and **laziness**: Python operators are generally eager and evaluate all sub-expressions to compute the final answer. This means `(3-3) * f(d)` is fully evaluated even though the first part of the multiplication—the `(3-3)` sub-expression—is always zero, meaning the result is always zero, no matter what value is computed by the expression `f(d)`. Generator expressions are an example of Python doing lazy evaluation. We'll look at an example of this in the next section, *Lazy and eager evaluation.*

In Python, the logical expression operators `and`, `or`, and `if-else` are all non-strict. We sometimes call them **short-circuit** operators because they don't need to evaluate all arguments to determine the resulting value.

The following command snippet shows the `and` operator's non-strict feature:

```
>>> 0 and print("right")
0

>>> True and print("right")
right
```

When we execute the first of the preceding command snippets, the left-hand side of the `and` operator is equivalent to `False`; the right-hand side is not evaluated. In the second example, when the left-hand side is equivalent to `True`, the right-hand side is evaluated.

Other parts of Python are strict. Outside the logical operators, an expression is evaluated strictly from left to right. A sequence of statement lines is also evaluated strictly in order. Literal lists and tuples require strict evaluation. When a class is created, the methods are defined in a strict order.

Lazy and eager evaluation

Python's generator expressions and generator functions are lazy. These expressions don't create all possible results immediately. It's difficult to see this without explicitly logging the details of a calculation. Here is an example of the version of the `range()` function that has the side effect of showing the numbers it creates:

```
from collections.abc import Iterator
def numbers(stop: int) -> Iterator[int]:
    for i in range(stop):
        print(f"{i=}")
        yield i
```

To provide some debugging hints, this function prints each value as the value is yielded. If this function were eager, evaluating `numbers(1024)` would take the time (and storage) to create all 1,024 numbers. Since the `numbers()` function is lazy, it only creates a number as it is requested.

We can use this noisy `numbers()` function in a way that will show lazy evaluation. We'll write a function that evaluates some, but not all, of the values from this iterator:

```
def sum_to(limit: int) -> int:
    sum: int = 0
    for i in numbers(1_024):
        if i == limit: break
```

```
        sum += i
    return sum
```

The `sum_to()` function has type hints to show that it should accept an integer value for the `n` parameter and return an integer result. This function will not evaluate the entire result of the values produced by the `numbers()` function. It will break after only consuming a few values from the `numbers()` function. We can see this consumption of values in the following log:

```
>>> sum_to(5)
i=0
i=1
i=2
i=3
i=4
i=5
10
```

As we'll see later, Python generator functions have some properties that make them a little awkward for simple functional programming. Specifically, a generator can only be used once in Python. We have to be cautious with how we use the lazy Python generator expressions.

Recursion instead of an explicit loop state

Functional programs don't rely on loops and the associated overhead of tracking the state of loops. Instead, functional programs try to rely on the much simpler approach of recursive functions. In some languages, the programs are written as recursions, but **Tail-Call Optimization** (**TCO**) in the compiler changes them to loops. We'll introduce some recursion here and examine it closely in *Chapter 6*, *Recursions and Reductions*.

We'll look at an iteration to test whether a number is a prime number. Here's a definition from `https://mathworld.wolfram.com/PrimeNumber.html`: "A prime number ... is a

positive integer $p > 1$ that has no positive integer divisors other than 1 and p itself." We can create a naive and poorly performing algorithm to determine whether a number has any factors between 2 and the number. This is called the *Trial Division* algorithm. It has the advantage of simplicity; it works acceptably for solving some of the **Project Euler** problems. Read up on **Miller-Rabin** primality tests for a much better algorithm.

We'll use the term *coprime* to mean that two numbers have only 1 as their common factor. The numbers 2 and 3, for example, are coprime. The numbers 6 and 9, however, are not coprime because they have 3 as a common factor.

If we want to know whether a number, n, is prime, we actually ask this: is the number n coprime to all prime numbers, p, such that $p^2 < n$? We can simplify this using all integers, i, such that $2 \le i^2 < n$. The simplification does more work, but is much easier to implement.

Sometimes, it helps to formalize this as follows:

$$\text{prime}(n) = \forall x[2 \le x < \sqrt{n} + 1 \wedge n \not\equiv 0 \mod x]$$

The expression could look as follows in Python:

```
not any(
    n % p == 0
    for p in range(2, int(math.sqrt(n))+1)
)
```

An alternative conversion from mathematical formalism to Python would use `all(n % p != 0, ...)`. The `all()` function will stop when it finds the first `False` value. The `not any()` will stop when it finds the first `True` value. While the results are identical, the performance varies depending on whether or not `p` is a prime number.

This expression has a `for` iteration inside it: it's not a pure example of stateless functional programming. We can reframe this into a function that works with a collection of values. We can ask whether the number, n, is coprime within any value in the half-open interval $[2, \sqrt{n} + 1)$. This uses the symbols [) to show a half-open interval: the lower values are

included, and the upper value is not included. This is typical behavior of the Python `range()` function. We will also restrict ourselves to the domain of natural numbers. The square root values, for example, are implicitly truncated to integers.

We can think of the definition of prime as the following:

$$\text{prime}(n) = \text{coprime}\Big(n, \big[2, \sqrt{n} + 1\big)\Big)$$

given $n > 1$. We know n is prime when it is coprime to all values in the range $[2, \sqrt{n} + 1)$.

While the formal math can feel daunting, this is a search for a coprime in the given range of values. If we find a coprime, the value of n is not prime. If we fail to find a coprime, then the value of n must be prime.

When defining a recursive search over a range of values, the base case can be the empty range. Searching the empty range means no values can be found. Searching a non-empty range is handled recursively by processing one value combined with a range that's narrower by the one value processed. We could formalize it as follows:

$$\text{coprime}\Big(n, [a, b)\Big) = \begin{cases} \textbf{True} & \textbf{if } a = b\textbf{,} \\ & \textbf{the range is empty} \\ (n \not\equiv 0 \mod a) \wedge \text{coprime}\Big(n, [a+1, b)\Big) & \textbf{if } a < b \end{cases}$$

In the case where the range is non-empty, one value, a, is checked to see if it is coprime with n; then, the remaining values in the range $[a + 1, b)$ are checked. This expression can be confirmed by providing concrete examples of the two cases, which are given as follows:

- If the range is empty, $a = b$, we evaluated something like this:

$$\text{coprime}\Big(131073, [363, 363)\Big)$$

  The range contains no values, so the return is `True`. This is analogous to computing

the sum of an empty list: the sum is zero.

- If the range is not empty, we evaluated something like this:

$$\text{coprime}\Big(131073, [2, 363)\Big)$$

This decomposes into evaluating:

$$(131073 \not\equiv 0 \mod 2) \wedge \text{coprime}\Big(131073, [3, 363)\Big)$$

For this example, we can see that the first clause is `True`, and we'll evaluate the second clause recursively. Compare this with evaluating $\text{coprime}\Big(16, [2, 5)\Big)$. The value of $16 \not\equiv 0 \mod 2$ would be `False`; the values of 16 and 2 are not coprime. The evaluation of $\text{coprime}\Big(131073, [3, 363)\Big)$ becomes irrelevant, since we know the 16 is composite.

As an exercise for the reader, this recursion can be redefined to count down instead of up, using $[a, b - 1)$ in the second case. Try this revision to see what, if any, changes are required.

Some folks like to define the empty interval as $a \geq b$ instead of $a = b$. The extra > condition is needless, since a is incremented by 1 and we can easily guarantee that $a \leq b$, initially. There's no way for a to somehow magically leap past b through some error in the function; we don't need to over-specify the rules for an empty interval.

Here is a Python code snippet that implements this definition of prime:

```
def isprimer(n: int) -> bool:
    def iscoprime(k: int, a: int, b: int) -> bool:
        """Is k coprime with a value in the given range?"""
        if a == b: return True
        return (k % a != 0) and iscoprime(k, a+1, b)
```

```
    return iscoprime(n, 2, int(math.sqrt(n)) + 1)
```

This shows a recursive definition of an `iscoprime()` function. The function expects an `int` value for all three parameters. The type hints claim it will return a `bool` result.

The recursion base case is implemented as `a == b`. When this is true, the range of values from `a` to one less than `b` is empty. Because the recursive evaluation of `iscoprime()` is the tail end of the function, this is an example of **tail recursion**.

The `iscoprime()` function is embedded in the `isprimer()` function. The outer function serves to establish the boundary condition for the range of values that will be searched.

What's important in this example is that the two cases of this recursive function follow the mathematical definition in a direct way. Making the range of values an explicit argument to the internal `iscoprime()` function allows us to call the function recursively with argument values that reflect a steadily shrinking interval.

While recursion is often succinct and expressive, we have to be cautious about using it in Python. There are two problems that can arise:

- Python imposes a recursion limit to detect recursive functions with improperly defined base cases.
- Python does not have a compiler that does **Tail-Call Optimization** (**TCO**) for us.

The default recursion limit is 1,000, which is adequate for many algorithms. It's possible to change this with the `sys.setrecursionlimit()` function. It's not wise to raise this arbitrarily since it might lead to exceeding the OS memory limitations and crashing the Python runtime.

If we try a recursive `isprimer()` function on a prime number `n` over 1,000,000, we'll run afoul of the recursion limit. (Folks using IPython have a higher default limit on the size of the stack; try `isprimer(9_000_011)` to see the problem.)

Some functional programming languages can optimize these "tail call" recursive functions. An optimizing compiler will transform the recursive evaluation of the `iscoprime(k, a+1, b)`

expression into a low-overhead `for` statement. The optimization tends to make debugging optimized programs more difficult. Python doesn't perform this optimization. Performance and memory are sacrificed for clarity and simplicity. This also means we are forced to do the optimization manually.

This is the subject of *Chapter 6, Recursions and Reductions*. We'll look at several examples of doing manual TCO.

Functional type systems

Some functional programming languages, such as **Haskell** and **Scala**, are statically compiled, and depend on declared types for functions and their arguments. To provide the kind of flexibility Python already has, these languages have sophisticated type-matching rules allowing a generic function to work for a variety of related types.

In object-oriented Python, we often use the class inheritance hierarchy instead of sophisticated function type matching. We rely on Python to dispatch an operator to a proper method based on simple name-matching rules.

Python's built-in "duck typing" rules offer a great deal of type flexibility. The more complex type matching rules for a compiled functional language aren't relevant. It's common to define a `typing.Protocol` to specify the features an object must have. The actual class hierarchy doesn't matter; what matters is the presence of the appropriate methods and attributes.

Python's `match` statement offers a number of structure and type matching capabilities. Because the `match` statement has so many alternatives, we'll return to it several times. For now, we'll provide an introductory example to show the core syntax.

Here's an example that relies on literal matching, wildcard matching with _, and guard conditions:

```
import math
def isprimem(n: int) -> bool:
    match n:
        case _ if n < 2:
            prime = False
        case 2:
            prime = True
        case _ if n % 2 == 0:
            prime = False
        case _:
            for i in range(3, 1 + int(math.sqrt(n)), 2):
                if n % i == 0:
                    # Stop as soon as we know...
                    return False
            prime = True
    return prime
```

When working with a single data type, the `match` statement is not dramatically simpler than an `if-elif` chain. The `case _` blocks use the `_` pattern, which matches anything without binding any variables. Some of these are followed by additional guards, for example, `if n < 2`, to provide a more nuanced decision to these cases.

The final `case _:` matches any possible value not matched by any of the prior `case` blocks. It is analogous to the `else:` clause in an `if` statement.

The single `return` statement at the end is expected by **mypy**. We could rewrite this to use `return` statements for each case. While it would work properly, without a single, clear `return` it's difficult for the **mypy** tool to confirm that the `match` statement truly covers all possible conditions.

As we look at other examples, we'll see more of the power of the pattern and type matching capabilities. This example matches literal values of a single type. The `match` statement can do quite a bit more. In later chapters, we'll see the distinction between type hints, checked by a tool like **mypy**, and type matching that can be done by the `match` statement.

Familiar territory

One of the ideas that emerges from the previous list of topics is that many functional programming constructs are already present in Python. Indeed, elements of functional programming are already a very typical and common part of OOP.

As a very specific example, a fluent **Application Program Interface** (**API**) is a very clear example of functional programming. If we take time to create a class with `return self` in each method, we can use it as follows:

```
some_object.foo().bar().yet_more()
```

We can just as easily write several closely related functions that work as follows:

```
yet_more(bar(foo(some_object)))
```

We've switched the syntax from traditional object-oriented suffix notation to a more functional prefix notation. Python uses both notations freely, often using a prefix version of a special method name. For example, the `len()` function is generally implemented by the `__len__()` class special method.

Of course, the implementation of the preceding class might involve a highly stateful object. Even then, a small change in viewpoint may reveal a functional approach that can lead to more succinct or more expressive programming.

The point is not that imperative programming is broken in some way, or that functional programming offers a vastly superior technology. The point is that functional programming leads to a change in viewpoint that can, in many cases, be helpful for designing succinct, expressive programs.

Learning some advanced concepts

We will set some more advanced concepts aside for consideration in later chapters. These concepts are part of the implementation of a purely functional language. Since Python isn't

purely functional, our hybrid approach won't require deep consideration of these topics.

We will identify these here for the benefit of readers who already know a functional language such as **Haskell** and are learning Python. The underlying concerns are present in all programming languages, but we'll tackle them differently in Python. In many cases, we can and will drop into imperative programming rather than use a strictly functional approach.

The topics are as follows:

- **Referential transparency**: When looking at lazy evaluation and the various kinds of optimizations that are possible in a compiled language, the idea of multiple routes to the same object is important. In Python, this isn't as important because there aren't any relevant compile-time optimizations.
- **Currying**: The type systems will employ currying to reduce multiple-argument functions to single-argument functions. We'll look at currying in some depth in *Chapter 12*, *Decorator Design Techniques*.
- **Monads**: These are purely functional constructs that allow us to structure a sequential pipeline of processing in a flexible way. In some cases, we'll resort to imperative Python to achieve the same end. We'll also leverage the elegant PyMonad library for this. We'll defer this until *Chapter 13*, *The PyMonad Library*.

Summary

In this chapter, we've identified a number of features that characterize the functional programming paradigm. We started with first-class and higher-order functions. The idea is that a function can be an argument to a function or the result of a function. When functions become the object of additional programming, we can write some extremely flexible and generic algorithms.

The idea of immutable data is sometimes odd in an imperative and object-oriented programming language such as Python. When we start to focus on functional programming, however, we see a number of ways that state changes can be confusing or unhelpful. Using

immutable objects can be a helpful simplification.

Python focuses on strict evaluation: all sub-expressions are evaluated from left to right through the statement. Python, however, does perform some non-strict evaluation. The `or`, `and`, and `if-else` logical operators are non-strict: all sub-expressions are not necessarily evaluated.

Generator functions can be described as lazy. While Python is generally eager, and evaluates all sub-expressions a soon as possible, we can leverage generator functions to create lazy evaluation. With lazy evaluation, a computation is not performed until it's needed.

While functional programming relies on recursion instead of the explicit loop state, Python imposes some limitations here. Because of the stack limitation and the lack of an optimizing compiler, we're forced to manually optimize recursive functions. We'll return to this topic in *Chapter 6, Recursions and Reductions*.

Although many functional languages have sophisticated type systems, we'll rely on Python's dynamic type resolution. In some cases, this means we'll have to write manual coercion for various types.

In the next chapter, we'll look at the core concepts of pure functions and how these fit in with Python's built-in data structures. Given this foundation, we can look at the higher-order functions available in Python and how we can define our own higher-order functions.

Exercises

This chapter's exercises are based on code available from Packt Publishing on GitHub. See `https://github.com/PacktPublishing/Functional-Python-Programming-3rd-Edition`.

In some cases, the reader will notice that the code provided on GitHub includes partial solutions to some of the exercises. These serve as hints, allowing the reader to explore alternative solutions.

In many cases, exercises will need unit test cases to confirm they actually solve the problem.

These are often identical to the unit test cases already provided in the GitHub repository. The reader should replace the book's example function name with their own solution to confirm that it works.

Apply map() to a sequence of values

Some analysis has revealed a consistent measurement error in a device. The machine's revolutions per minute (RPM) as displayed on the tachometer are consistently incorrect. (Gathering the true RPM involves some heroic engineering effort, but is not a sustainable way to manage the machine.)

The result is a model that translates observed, o, to actual, a:

$$a = f(o) = 0.9 \times o - 90$$

The machine can only operate in the range of 800 to 2,500 RPM. Until the tachometer can be replaced with one that's properly calibrated, we need a table of values from observed RPM to actual RPM. This can be printed and laminated and put near the machine to help gauge fuel consumption and workload.

Because the tachometer can only be read to the nearest 100 RPM, the table only needs to show values like 800, 900, 1000, 1100, ..., 2500.

The output should be something like the following:

```
Observed  Actual
     800     630
     900     720
etc.
```

In order to provide a flexible solution, it helps to create the following two separate functions:

- A function to implement the model, which computes the actual value from the observation
- A function to display the table of values produced from the results of the model

function

These two–separate–functions will be used as part of the recalibration effort for the piece of equipment.

Test cases for the model can be isolated from test cases for the table of values, allowing new models to be used as more data is gathered.

While it is the topic of a later chapter, and has only been mentioned here, use of `map()` is encouraged, but not required.

Function vs. lambda design question

The model in the *Apply map() to a sequence of values* problem is a small function, only about one line of code. Here are three different ways this can be written:

- As a proper `def` function.
- As a lambda object.
- As a class definition that implements the `__call__()` method.

Create all three implementations. Compare and contrast them with respect to ease of understanding. Defend one as being ideal by (a) providing some criteria for software quality and (b) showing how the implementation meets those criteria.

Optimize a recursion

See *Recursion instead of an explicit loop state* earlier in this chapter.

As an exercise for the reader, this recursion can be redefined to count down instead of up, using $[a, b - 1)$ in the second case. Implement this change to see what, if any, changes are required. Measure the performance to see if there is any performance consequence.

Join our community Discord space

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

`https://packt.link/dHrHU`

3

Functions, Iterators, and Generators

The core of functional programming is the use of pure functions to map values from the input domain to the output range. Avoiding side effects reduces any dependence on variable assignment to maintain the state of a computation. We can't purge the assignment statement from the Python language, but we can reduce our dependence on stateful objects. This means choosing among the available Python built-in functions and data structures to select those that don't require stateful operations.

This chapter will present several Python features from a functional viewpoint, as follows:

- Pure functions, free of side effects
- Functions as objects that can be passed as arguments or returned as results
- The use of Python's object-oriented suffix notation and prefix notation
- Using tuples as a way to create immutable objects, which avoid the confusion of state changes

- Using iterable collections as our primary design tool for functional programming

We'll look at generators and generator expressions, since these are ways to work with collections of objects. As we noted in *Chapter 2*, *Introducing Essential Functional Concepts*, there are some boundary issues when trying to replace all generator expressions with recursions. Python imposes a recursion limit and doesn't automatically handle **Tail-Call Optimization** (**TCO**): we must optimize recursions manually using a generator expression.

We'll write generator expressions that will perform the following tasks:

- Conversions
- Restructuring
- Complex calculations

We'll take a quick survey of many of the built-in Python collections and how we can work with collections while pursuing a functional paradigm. This may change our approach to working with lists, dicts, and sets. Writing functional Python encourages us to focus on tuples and immutable collections. In the next chapter, we'll emphasize more functional ways to work with specific kinds of collections.

Writing pure functions

In *Chapter 2*, *Introducing Essential Functional Concepts*, we looked at pure functions. In this section, we'll look at a common problem with non-functional programming: a function that has a reference to a global variable. When a global variable is assigned, the `global` statement will be used. When a global variable is read, however, this is called a **free variable**, and there's no obvious marker in the Python code.

Any references to values in the Python global namespace (using a free variable) is something we can rework into a proper parameter. In most cases, it's quite easy. Here is an example that depends on a free variable:

```
global_adjustment: float

def some_function(a: float, b: float, t: float) -> float:
    return a+b*t+global_adjustment
```

After refactoring the function, we would need to change each reference to this function. This may have a ripple effect through a complex application. We'll leave the refactoring as an exercise.

There are many internal Python objects that are stateful. Objects used for input and output are generally called file objects or file-like objects; these are examples of stateful objects in common use. See the `io` module for more information on file objects. We observe that some of the commonly used stateful objects in Python generally behave as context managers. In a few cases, stateful objects don't completely implement the context manager interface; in these cases, there's often a `close()` method. We can use the `contextlib.closing()` function to provide these objects with the proper context manager interface.

A context manager provides a way to perform operations on entry to and exit from a block of code. The `with` statement uses a context manager to perform the entry operation, execute the indented block of code, and perform the exit operation. It's important to note the exit operation is *always* performed, even if an exception is raised in the indented block of code. This makes for a tidy way to perform state-changing operations, making the code easier to reason about. In practice, it looks like the following example:

```
from pathlib import Path

def write_file(some_path: Path) -> None:
    result = "Hello, world!"
    with some_path.open('w') as output_file:
        output_file.write(result + "\n")
```

The file is only open for writing inside the `with` statement. This makes it easier to see where state-changing operations are performed.

We can't easily eliminate all stateful Python objects. Therefore, we must strike a balance between managing state while still exploiting the strengths of functional design. To this end, we should always use the `with` statement to encapsulate stateful file objects into a well-defined scope.

Always use file objects in a `with` context. This defines a context in which state-changing operations will be performed.

We should always avoid global file objects, global database connections, and the associated stateful object issues. A global file object is a common pattern for handling open files or databases. We may have a function as shown in the following example:

```
from typing import TextIO
ifile: TextIO
ofile: TextIO

def open_files(iname: str, oname: str) -> None:
    """A bad idea..."""
    global ifile, ofile
    ifile = open(iname, "r")
    ofile = open(oname, "w")
```

This function creates an easily overlooked pair of global variables. Other functions can use the `ifile` and `ofile` variables, hoping they properly refer to the global files, which are left open and will endure a difficult-to-understand series of state changes.

This is not a very functional design, and we need to avoid it. The files should be proper parameters to functions, and the open files should be nested in a `with` statement to ensure that their stateful behavior is handled properly. This is an important rewrite to change these variables from globals to formal parameters: it makes the file operations more visible.

The rewrite will involve locating every function using `ifile` or `ofile` as free variables. For example, we might have a function like the following:

```
def next_line_with(prefix: str) -> str | None:
    """Also a bad idea..."""
    line = ifile.readline()
    while (line is not None and not line.startswith(prefix)):
        line = ifile.readline()
    return line
```

We'll need to make the `ifile` global variable reference into a parameter to this function. This will create ripples of change to the functions that call `next_line_with()`. This can become an extensive rewrite to identify and localize the state changes. This may lead to rethinking the design to replace a function like `next_line_with()`.

This context manager design pattern also applies to databases. A database connection object should generally be provided as a formal argument to an application's functions. This is contrary to the way some popular web frameworks work: some frameworks rely on a global database connection in an effort to make the database a transparent feature of the application. This transparency obscures a dependency between a web operation and the database; it can make unit testing more complex than necessary. Additionally, a multithreaded web server may not benefit from sharing a single database connection: a connection pool is often better. This suggests that there are some benefits of a hybrid approach that uses functional design with a few isolated stateful features.

Functions as first-class objects

In *Chapter 2, Introducing Essential Functional Concepts*, we looked at ways in which Python functions are first-class objects. In Python, function objects have a number of attributes. The reference manual lists a number of special member names that apply to functions. Since functions are objects with attributes, we can extract the docstring or the name of a function, using special attributes such as `__doc__` or `__name__`. We can also extract the body of the function through the `__code__` attribute. In compiled languages, this introspection can be either impossible or quite complicated.

Additionally, a callable object helps us to create functions. We can consider the callable

class definition as a higher-order function. We do need to be judicious in how we use the `__init__()` method of a callable object; we should avoid setting stateful class variables. One common application is to use an `__init__()` method to create objects that fit the **Strategy** design pattern.

A class following the Strategy design pattern depends on other objects to provide an algorithm or parts of an algorithm. This allows us to inject algorithmic details at runtime, rather than compiling the details into the class.

To focus on the overall design principles, we'll look at a function that does a tiny computation. This computes one of the Mersenne prime numbers. See `https://www.mersenne.org/primes/` for ongoing research into this topic.

Here is an example of the definition of a callable object class with an embedded Strategy object:

```
from collections.abc import Callable

class Mersenne1:

    def __init__(
            self,
            algorithm : Callable[[int], int]
    ) -> None:
        self.pow2 = algorithm

    def __call__(self, arg: int) -> int:
        return self.pow2(arg) - 1
```

This class uses `__init__()` to save a reference to another function, `algorithm`, as `self.pow2`. We're not creating any stateful instance variables; the value of `self.pow2` isn't expected to change. It's a common practice to use a name like `_pow2` to suggest this attribute isn't expected to be used by a client of this class. The `algorithm` parameter has a type hint of `Callable[[int], int]`, which describes a function that takes an integer argument and returns an integer value.

We've used the `Callable` type hint from the `collections.abc` module, where it is defined. An alias is available in the `typing` module, but since PEP 585 was implemented, the use of the `typing.Callable` is deprecated. We'll use a number of generic types from the `collections.abc` module throughout this chapter.

The function given as a Strategy object must raise 2 to the given power. We can plug in any function that performs this computation. Three candidate objects that we can plug into this class are as follows:

```
def shifty(b: int) -> int:
    return 1 << b

def multy(b: int) -> int:
    if b == 0: return 1
    return 2 * multy(b - 1)

def faster(b: int) -> int:
    if b == 0: return 1
    if b % 2 == 1: return 2 * faster(b-1)
    t = faster(b // 2)
    return t * t
```

The `shifty()` function raises 2 to the desired power using a left shift of the bits. The `multy()` function uses a naive recursive multiplication. The `faster()` function uses a **divide and conquer** strategy that will perform $\log_2(b)$ multiplications instead of b multiplications.

All three of these functions have identical function signatures. Each of them can be summarized as `Callable[[int], int]`, which matches the parameter, `algorithm`, of the `Mersenne1.__init__()` method.

We can create instances of our `Mersenne1` class with an embedded Strategy algorithm, as follows:

```
m1s = Mersenne1(shifty)

m1m = Mersenne1(multy)

m1f = Mersenne1(faster)
```

Each of the resulting functions, `m1s()`, `m1m()`, and `m1f()`, is built from another function. The functions `shifty()`, `multy()`, and `faster()` are incorporated into resulting functions. This shows how we can define alternative functions that produce the same result but use different algorithms.

The callable objects created by this class behave as ordinary Python functions, as shown in the following example:

```
>>> m1s(17)
131071
>>> m1f(89)
618970019642690137449562111
```

Python allows us to compute $M_{89} = 2^{89} - 1$, since this doesn't even come close to the recursion limits in Python. This is quite a large prime number, as it has 27 digits. In order to exceed the limits of the `multy()` function, we'd have to ask for the value of $M_{1,279}$, a number with 386 digits.

Using strings

Since Python strings are immutable, they're an excellent example of functional programming objects. A Python `str` object has a number of methods, all of which produce a new string as the result. These methods are pure functions with no side effects.

The syntax for methods is postfix, where most functions are prefix. This mixture of syntax styles means complex string operations can be hard to read when they're co-mingled with conventional functions. For example, in this expression, `len(variable.title())`, the

`title()` method is in postfix notation and the `len()` function is in prefix notation. (We touched on this in *Chapter 2*, *Introducing Essential Functional Concepts*, in the *Familiar territory* section.)

When scraping data from a web page, we may have a function to clean the data. This could apply a number of transformations to a string to clean up the punctuation and return a `Decimal` object for use by the rest of the application. This will involve a mixture of prefix and postfix syntax.

It could look like the following code snippet:

```
from decimal import Decimal

def clean_decimal(text: str | None) -> Decimal | None:
    if text is None: return None
    return Decimal(
        text.replace("$", "").replace(",", "")
    )
```

This function does two replacements on the string to remove `$` and `,` string values. The resulting string is used as an argument to the `Decimal` class constructor, which returns the desired object. If the input value is `None`, this is preserved; this is why the `str | None` type hint is used.

To make the syntax look more consistent, we can consider defining our own prefix functions for the string methods, as follows:

```
def replace(text: str, a: str, b: str) -> str:
    return text.replace(a, b)
```

This can allow us to use `Decimal(replace(replace(text, "$", ""), ",", ""))` with consistent-looking prefix syntax. It's not clear whether this kind of consistency is a significant improvement over the mixed prefix and postfix notation. This may be an example of a foolish consistency.

A slightly better approach may be to define a more meaningful prefix function to strip punctuation, such as the following code snippet:

```
def remove(str: str, chars: str) -> str:
    if chars:
        return remove(
            str.replace(chars[0], ""),
            chars[1:]
        )
    return str
```

This function will recursively remove each of the characters from the `chars` variable. We can use it as `Decimal(remove(text, "$,"))` to make the intent of our string cleanup more clear.

Using tuples and named tuples

Since Python tuples are immutable objects, they're another excellent example of objects suitable for functional programming. A Python tuple has very few methods, so almost everything is done using prefix syntax. There are a number of use cases for tuples, particularly when working with list-of-tuple, tuple-of-tuple, and generator-of-tuple constructs.

The `typing.NamedTuple` class adds an essential feature to a tuple: names to use instead of cryptic index numbers. We can exploit named tuples to create objects that are accretions of data. This allows us to write pure functions based on stateless objects, yet keep data bound into tidy object-like packages. The `collections.namedtuple()` can also be used to define an immutable class of objects. This lacks a mechanism for providing type hints, making it less desirable than the `typing.NamedTuple` class.

The decision to use a tuple or `typing.NamedTuple` object is entirely a matter of convenience. As an example, consider working with a sequence of color values as a three-tuple of the form `(number, number, number)`. It's not clear that these are in red, green, blue order. We have a number of approaches for making the tuple structure explicit.

One purely functional approach to expose the triple structure is by creating functions to pick a three-tuple apart, as shown in the following code snippet:

```
from collections.abc import Callable
from typing import TypeAlias

Extractor: TypeAlias = Callable[[tuple[int, int, int, str]], int]

red: Extractor = lambda color: color[0]

green: Extractor = lambda color: color[1]

blue: Extractor = lambda color: color[2]
```

Given a tuple, `item`, we can use `red(item)` to select the item that has the red component. This style is used in a number of purely functional languages; it has a structure that matches mathematical abstractions nicely.

In Python, it can sometimes help to provide a more formal type hint on each variable, as follows:

```
from collections.abc import Callable
from typing import TypeAlias

RGB: TypeAlias = tuple[int, int, int, str]

redt: Callable[[RGB], int] = lambda color: color[0]
```

This defines a new type alias, `RGB`, as a four-tuple. The `redt()` function is provided with a type hint of `Callable[[RGB], int]` to indicate it should be considered to be a function that accepts an argument value of the `RGB` class and produces an integer result. This follows other styles of functional programming and adds type hints that can be checked by **mypy**.

A somewhat better technique is to use Python's `typing.NamedTuple` class. This uses a class definition instead of function definitions and looks like this:

```
from typing import NamedTuple
class Color(NamedTuple):
    """An RGB color."""
    red: int
    green: int
    blue: int
    name: str
```

The `Color` class defines a tuple with specific names and type hints for each position within the tuple. This preserves the advantages of performance and immutability. It adds the ability for the **mypy** program to confirm that the tuple is used properly.

This also means we'll use `color.red` instead of `red(color)`. Using an attribute name to access a member of a tuple seems to add clarity.

There are a number of additional approaches for working with immutable tuples. We'll look at all of these immutable class techniques in *Chapter 7, Complex Stateless Objects*.

Using generator expressions

We've shown some examples of generator expressions already in the *Lazy and eager evaluation* section of *Chapter 2, Introducing Essential Functional Concepts*. We'll show some more later in this chapter. In this section, we'll introduce some more generator techniques.

Python collections are described as iterable. We can use a `for` statement to iterate over the values. The key mechanism is the ability of a collection to create an iterator object to be used by the `for` statement. This concept generalizes to encompass a function that is an iterator over values. We call these generator functions. We can also write generator expressions.

It's common to see generator expressions used to create the `list` or `dict` literals through list comprehension or a dictionary comprehension syntax. This is a list comprehension example, `[x**2 for x in range(10)]`, a kind of list display. A list comprehension is one of several places in Python where generator expressions are used. In this example, the list

literal `[]` characters wrap the generator expression, `x**2 for x in range(10)`. This list comprehension creates a list object from the enclosed generator expression.

The underlying `x**2 for x in range(10)` expression yields a sequence of values. These must be consumed by a client function. The `list()` function can consume the values. This means we have two ways to create a list object from a generator expression, as shown in the following example:

```
>>> list(x**2 for x in range(10)) == [x**2 for x in range(10)]
True
```

There are other kinds of comprehensions to create dictionaries and sets. When the enclosing characters are `{}`, this is a set comprehension. When the enclosing characters are `{}`, and there are `:` to separate keys and values, this is a dictionary comprehension. In this section, we're going to focus on the generator expressions, separate from the specific kind of collection object they might create.

A collection object and a generator expression have some similar behaviors because both are iterable. They're not equivalent, as we'll see in the following code. Using a display object has the disadvantage of creating a (potentially large) collection of objects. A generator expression is lazy and creates objects only as required; this can improve performance.

We have to provide two important caveats on generator expressions, as follows:

- Generators have some of the same internal methods as lists. This means we can apply functions like `sorted()` and `iter()` to either generators or lists. An exception is the `len()` function, which needs to know the size of the collection and won't work for a generator.
- Generators can be used only once. After that, they appear empty.

A generator function is a function that has a `yield` expression in it. This makes the function act as an iterator. Each individual `yield` value must be individually consumed by a client function. For a tutorial introduction, see `https://wiki.python.org/moin/Generators`.

Also, see `https://docs.python.org/3/howto/functional.html}generator-expressions-and-list-comprehensions`.

We can use something like the following to create a sequence of possible prime numbers:

```
from collections.abc import Iterator

def candidates() -> Iterator[int]:
    for i in range(2, 1024):
        yield m1f(i)
```

This function iterates through 1,024 result values. It doesn't compute them eagerly, however. It is lazy, and computes values as they're requested. The built-in `next()` function is one way to consume values. Here's an example of consuming values from a generator function:

```
>>> c = candidates()
>>> next(c)
3
>>> next(c)
7
>>> next(c)
15
>>> next(c)
31
```

When the `candidates()` function is evaluated, it creates a generator object, which is saved in the variable `c`. Each time we use `next(c)`, the generator function computes one more value and yields it. In this example, it will get a new value from the `range` object, and evaluate the `m1f()` function to compute a new value.

The `yield from` expression extends the `yield` expression. This will consume values from some iterator, yielding each of the values it consumes. As a small example, consider the following function:

```
from collections.abc import Iterator

def bunch_of_numbers() -> Iterator[int]:
    for i in range(5):
        yield from range(i)
```

Each time a value is requested, it will be produced by the `yield from` nested inside the `for` statement. This will yield `i` distinct values, one from each request. Since `i` is set by the containing `for` statement, this will be used to produce ever longer sequences of numbers.

Here's what the result looks like:

```
>>> list(bunch_of_numbers())
[0, 0, 1, 0, 1, 2, 0, 1, 2, 3]
```

Here is a generator function that we'll use for some more examples:

```
from collections.abc import Iterator
import math

def pfactorsl(x: int) -> Iterator[int]:
    if x % 2 == 0:
        yield 2
        if x // 2 > 1:
            yield from pfactorsl(x // 2)
        return
    for i in range(3, int(math.sqrt(x) + .5) + 1, 2):
        if x % i == 0:
            yield i
            if x // i > 1:
                yield from pfactorsl(x // i)
            return
    yield x
```

We're locating the prime factors of a number. If the number, `x`, is even, we'll yield 2 and then recursively yield all prime factors of `x // 2`.

For odd numbers, we'll step through odd values greater than or equal to 3 to locate a candidate factor of the number. When we locate a factor, `i`, we'll yield that factor, and then recursively yield all prime factors of `x // i`.

In the event that we can't locate a factor, the number, `x`, must be prime, so we can yield the number.

We handle 2 as a special case to cut the number of iterations in half. All prime numbers, except 2, are odd.

We've used one important `for` statement in addition to recursion. This is an optimization, and it's a teaser for the content of *Chapter 6, Recursions and Reductions*. This optimization allows us to easily handle numbers that have as many as 1,000 factors. (As an example, $2^{1,000}$, a number with 300 digits, will have 1,000 factors.) Since the `for` variable, `i`, is not used outside the indented body of the statement, the stateful nature of the `i` variable won't lead to confusion if we make any changes to the body of the `for` statement.

Because the function, as a whole, is a generator, the `yield from` statement is used to consume values from the recursive call and yield them to the caller. It provides an iterable sequence of values as a result.

In a recursive generator function, be careful of the `return` statement.

Do not use the following statement: `return recursive_iter(args)`. It returns only a generator object; it doesn't evaluate the `recursive_iter()` function to return the generated values. Use any of the following alternatives:

- A `yield` expression:

```
for result in recursive_iter(args):
    yield result
```

- A `yield from` expression:

```
yield from recursive_iter(args)
```

A function that implements the `Iterator` protocol is often described as being a **generator function**. There is a separate `Generator` protocol, which extends the essential `Iterator` definition. We often find that functional Python programs can be structured around the generator expression construct. This tends to focus the design effort on functions and stateless objects.

Exploring the limitations of generators

We noted that there are some limitations of generator expressions and generator functions. The limitations can be observed by executing the following command snippet:

```
>>> pfactorsl(1560)
<generator object pfactorsl at ...>

>>> list(pfactorsl(1560))
[2, 2, 2, 3, 5, 13]

>>> len(pfactorsl(1560))
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: object of type 'generator' has no len()
```

In the first example, we saw the generator function, `pfactors1()`, created a generator. The generator is lazy, and doesn't have a proper value until we consume the results yielded by the generator. By itself, this isn't a limitation; lazy evaluation is an important reason why generator expressions fit with functional programming in Python.

In the second example, we materialized a list object from the results yielded by the generator function. This is handy for seeing the output and writing unit test cases.

In the third example, we saw one limitation of generator functions: there's no `len()`.

Because the generator is lazy, the size can't be known until after all of the values are consumed.

The other limitation of a generator object is that they can only be used once. For example, look at the following command snippet:

```
>>> result = pfactorsl(1560)
>>> sum(result)
27

>>> sum(result)
0
```

The first evaluation of the `sum()` function performed evaluation of the generator object, `result`. All of the values were consumed. The second evaluation of the `sum()` function found that the generator object was now empty. We can only consume the values of a generator object once.

The generator function, `pfactorsl()`, can produce an indefinite number of generator objects. In many cases, we'll define generator functions that consume the results yielded by other generators. In these cases, we may not be able to trivially create generators, but must create a whole pipeline of generators.

Generators have a stateful life in Python. While they're very nice for some aspects of functional programming, they're not quite perfect.

We can try to use the `itertools.tee()` function to overcome the once-only limitation. We'll look at this in depth in *Chapter 8, The Itertools Module*. It's not a great idea because it can consume a great deal of memory.

Here is a quick example of its usage:

```
import itertools
from typing import Any
from collections.abc import Iterable
```

```
def limits(iterable: Iterable[Any]) -> Any:
    max_tee, min_tee = itertools.tee(iterable, 2)
    return max(max_tee), min(min_tee)
```

We created two clones of the parameter generator expression, `max_tee` and `min_tee`. We can consume these two clones to get maximum and minimum values from the iterable. Interestingly, because the two clones are used serially, this leads to consuming a great deal of memory to cache items. This specific example often works out better using a list object instead of using `tee()` to clone an iterator.

Once consumed, a generator object will not provide any more values. When we want to compute multiple kinds of reductions—for example, sums and counts, or minimums and maximums—we need to design with this one-pass-only limitation in mind.

Combining generator expressions

The essence of functional programming comes from the ways we can easily combine generator expressions and generator functions to create very sophisticated composite processing sequences. When working with generator expressions, we can combine generators in several ways.

One common way to combine generator functions is when we create a composite function. We may have a generator that computes `(f(x) for x in some_iterable)`. If we want to compute `g(f(x))`, we have several ways to combine two generators.

We can tweak the original generator expression as follows:

```
g_f_x = (g(f(x)) for x in some_iterable)
```

While technically correct, this defeats any idea of reuse. Rather than reusing an expression, we rewrote it.

We can also substitute one expression within another expression, as follows:

```
g_f_x = (g(y) for y in (f(x) for x in some_iterable))
```

This has the advantage of allowing us to use simple substitution. We can revise this slightly to emphasize reuse, using the following commands:

```
f_x = (f(x) for x in some_iterable)
g_f_x = (g(y) for y in f_x)
```

This has the advantage of leaving the initial expression, `(f(x) for x in some_iterable)`, essentially unchanged. All we did was assign the expression to a variable without altering the syntax.

The resulting composite function is also a generator expression, which is also lazy. This means that extracting the next value from `g_f_x` will extract one value from `f_x`, which will extract one value from the source `some_iterable` object.

Cleaning raw data with generator functions

One of the tasks that arise in exploratory data analysis is cleaning up raw source data. This is often done as a composite operation applying several scalar functions to each piece of input data to create a usable dataset.

Let's look at a simplified set of data. This data is commonly used to show techniques in exploratory data analysis. It's called **Anscombe's quartet**, and it comes from the article *Graphs in Statistical Analysis*, by F. J. Anscombe, that appeared in *American Statistician* in 1973. The following are the first few rows of a downloaded file with this dataset:

```
Anscombe's quartet
I II III IV
x y x y x y x y
10.0 8.04 10.0 9.14 10.0 7.46 8.0 6.58
8.0 6.95 8.0 8.14 8.0 6.77 8.0 5.76
13.0 7.58 13.0 8.74 13.0 12.74 8.0 7.71
```

Since the data is properly tab-delimited, we can use the `csv.reader()` function to iterate through the various rows. Sadly, we can't trivially process this with the `csv` module. We have to do a little bit of parsing to extract the useful information from this file. We can define a function to iterate over the raw data as follows:

```
import csv
from typing import TextIO
from collections.abc import Iterator, Iterable

def row_iter(source: TextIO) -> Iterator[list[str]]:
    return csv.reader(source, delimiter="\t")
```

We wrapped a file in a `csv.reader()` function to create an iterator over the rows of raw data. The `typing` module provides a handy definition, `TextIO`, for file objects that read (or write) string values. Each row is a list of text values. It can be helpful to define an additional type, `Row = list[str]`, to make this more explicit.

We can use this `row_iter()` function in the following context:

```
>>> from pathlib import Path
>>> source_path = Path("Anscombe.txt")
>>> with source_path.open() as source:
...     print(list(row_iter(source)))
```

While this will display useful information, the problem is the first three items in the resulting iterable aren't data. The Anscombe's quartet file starts with the following header rows:

```
[["Anscombe's quartet"],
 ['I', 'II', 'III', 'IV'],
 ['x', 'y', 'x', 'y', 'x', 'y', 'x', 'y'],
```

We need to filter these three non-data rows from the iterable. There are several possible approaches. Here is a function that will excise three expected title rows, validate they are

the expected headers, and return an iterator over the remaining rows:

```
from collections.abc import Iterator

def head_split_fixed(
        row_iter: Iterator[list[str]]
) -> Iterator[list[str]]:
    title = next(row_iter)
    assert (len(title) == 1
        and title[0] == "Anscombe's quartet")
    heading = next(row_iter)
    assert (len(heading) == 4
        and heading == ['I', 'II', 'III', 'IV'])

    columns = next(row_iter)
    assert (len(columns) == 8
        and columns == ['x','y', 'x','y', 'x','y', 'x','y'])
    return row_iter
```

This function plucks three rows from the source data, an iterator. It asserts that each row has an expected value. If the file doesn't meet these basic expectations, it's a sign that the file was damaged or perhaps our analysis is focused on the wrong file.

Since both the `row_iter()` and the `head_split_fixed()` functions expect an iterator as an argument value, they can be combined, as follows:

```
from pathlib import Path
from collections.abc import Iterator

def get_rows(path: Path) -> Iterator[list[str]]:
    with path.open() as source:
        yield from head_split_fixed(row_iter(source))
```

We've applied one iterator to the results of another iterator. In effect, this defines a composite function. We're not done, of course; we still need to convert the string values to the float values, and we also need to pick apart the four parallel series of data in each row.

The final conversions and data extractions are more easily done with higher-order functions, such as `map()` and `filter()`. We'll return to those in *Chapter 5*, *Higher-Order Functions*.

Applying generators to built-in collections

We'll now look at ways to apply generator expressions to a number of Python's built-in collections. This section will cover the following topics:

- Generators for lists, dicts, and sets
- Working with stateful collections
- Using the `bisect` module to create a mapping
- Using stateful sets

Each of these looks at some specialized cases of Python collections and generator functions. In particular, we'll look at ways to produce a collection, and consume the collection in later processing.

This is a lead-in to the next chapter, *Chapter 4*, *Working with Collections*, which covers the Python collections in considerably more detail.

Generators for lists, dicts, and sets

A Python sequence object, such as a list, is iterable. However, it has some additional features. We can think of a list as a materialized iterable. We've used the `tuple()` function in several examples to collect the output of a generator expression or generator function into a single tuple object. We can use the `list()` function to materialize a sequence to create a list object.

In Python, a **list display**, or **list comprehension**, offers simple syntax to materialize a generator: we add the `[]` brackets. This is ubiquitous to the point where the distinction between generator expression and list comprehension can be lost. We need to disentangle the idea of a generator expression from a list display that uses a generator expression.

The following is an example to enumerate the cases:

```
>>> range(10)
range(0, 10)

>>> [range(10)]
[range(0, 10)]

>>> [x for x in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The first example is the `range` object, which is a type of generator function. It doesn't produce any values because it's lazy.

The second example shows a list composed of a single instance of the generator function. The `[]` syntax creates a list literal of the `range()` object without consuming any values created by the iterator.

The third example shows a list comprehension built from a generator expression that includes a generator function. The function, `range(10)`, is evaluated by a generator expression, `x for x in range(10)`. The resulting values are collected into a list object.

We can also use the `list()` function to build a list from an iterable or a generator expression. This also works for `set()`, `tuple()`, and `dict()`.

The `list(range(10))` function evaluates the generator object. The `[range(10)]` list literal does not evaluate the `range(10)` generator object.

While there's shorthand syntax for `list`, `dict`, and `set` using `[]` and `{}`, there's no shorthand syntax for a tuple. To materialize a tuple, we must use the `tuple()` function. For this reason, it often seems most consistent to use the `list()`, `tuple()`, and `set()` functions as the preferred syntax.

In the data cleansing code in the previous section, we used a composite function to create

a list of four tuples. The function looked as follows:

```
>>> data = list(get_rows(Path("Anscombe.txt")))
>>> data[0]
['10.0', '8.04', '10.0', '9.14', '10.0', '7.46', '8.0', '6.58']
>>> data[1]
['8.0', '6.95', '8.0', '8.14', '8.0', '6.77', '8.0', '5.76']
>>> data[-1]
['5.0', '5.68', '5.0', '4.74', '5.0', '5.73', '8.0', '6.89']
```

We assigned the results of the `get_rows()` composite function to a name, `data`. Each of these rows is a collection of four (x, y) pairs.

To extract one of the (x, y) pairs, we'll need to do a little bit more processing to make this useful. First, we need to pick pairs of columns from the eight-tuple. Since the pairs are always adjacent, we can select a pair of columns with a slice operation of the form `row[2 * n: 2 * n + 2]`. The idea is that pair n is in positions $2 \times n$ and $2 \times n + 1$. The slice expression `2 * n: 2 * n + 2` includes the start element, `2 * n`, and stops just before the stop element, `2 * n + 2`. We can wrap this with a reusable function, as shown in the following definition:

```
from typing import cast, TypeVar
from collections.abc import Iterator, Iterable

SrcT = TypeVar("SrcT")

def series(
        n: int,
        row_iter: Iterable[list[SrcT]]
) -> Iterator[tuple[SrcT, SrcT]]:
    for row in row_iter:
        yield cast(tuple[SrcT, SrcT], tuple(row[n * 2: n * 2 + 2]))
```

This function picks two adjacent columns based on a number between 0 and 3. It creates a tuple object from those two columns. The `cast()` function is a type hint to in-

form the **mypy** tool that the result will be a two-tuple where both items are strings. This is required because it's difficult for the **mypy** tool to determine that the expression `tuple(row[n * 2: n * 2 + 2])` will select exactly two elements from the row collection.

This example uses a type variable, `SrcT`, to make a deeper claim about the transformation. Specifically, the type variable tells people reading the code (and tools like **mypy**) that the input object types will be the resulting object types. If the source is, for example, an iterable of lists of `str`, then `SrcT = str`, and the output will be an iterator over tuples with two `str` values.

We can use the `series()` function to extract collections from the source file as follows:

```
>>> from pathlib import Path
>>> source_path = Path("Anscombe.txt")
>>> with source_path.open() as source:
...     data = tuple(head_split_fixed(row_iter(source)))
>>> series_I = tuple(series(0, data))
>>> series_II = tuple(series(1, data))
>>> series_III = tuple(series(2, data))
>>> series_IV = tuple(series(3, data))
```

We applied the `tuple()` function to a composite function based on the `series()`, `head_split_fixed()`, and `row_iter()` functions. Each of these expressions will create an object that we can reuse in several other functions. We can then do analysis on subsets of the source data.

The `series_I` sequence looks as follows:

```
>>> series_I
(('10.0', '8.04'), ('8.0', '6.95'), ... ('5.0', '5.68'))
```

The other three sequences are similar in structure. The values, however, are quite different.

The final thing we'll need to do is create proper numeric values from the strings that we've accumulated so that we can compute some statistical summary values. We can apply the

`float()` function conversion as the last step. There are many alternative places to apply the `float()` function, and we'll look at some choices in *Chapter 5, Higher-Order Functions*.

To reduce memory use and increase performance, we prefer to use generator expressions and functions as much as possible. These iterate through collections in a lazy manner, computing values only when required. Since iterators can only be used once, we're sometimes forced to materialize a collection as a tuple (or list) object. Materializing a collection costs memory and time, so we do it reluctantly.

Programmers familiar with **Clojure** can match Python's lazy generators with the `lazy-seq` and `lazy-cat` functions. The idea is that we can specify a potentially infinite sequence, but only take values from it as needed.

Using stateful mappings

Python offers several stateful collections; the various mappings include the `dict` class and a number of related mappings defined in the `collections` module. We need to emphasize the stateful nature of these mappings and use them carefully.

For our purposes, learning functional programming techniques in Python, there are two use cases for mapping: a stateful dictionary that accumulates a mapping and a frozen dictionary that can't be updated. Python doesn't provide an easy-to-use definition of an immutable mapping. We can use the abstract base class `Mapping` from the `collections.abc` module. We can also create an immutable `MappingProxyType` object from a mutable mapping. For more information, see the `types` module.

The stateful dictionary can be further decomposed into the following two typical use cases:

- A dictionary built once and never updated. In this case, we want to exploit the hashed keys feature of the `dict` class to optimize performance. We can create a dictionary from any iterable sequence of `(key, value)` two-tuples using the expression `dict(sequence)`.
- A dictionary built incrementally. This is an optimization we can use to avoid materializing and sorting a list object. We'll look at this in *Chapter 6, Recursions and*

Reductions, where we'll look at the `collections.Counter` class as a sophisticated reduction. Incremental building is particularly helpful for memoization. We'll defer memoization until *Chapter 10, The Functools Module*.

The first example, building a dictionary once, stems from an application with three operating phases: gather some input, create a `dict` object, and then process input based on the mappings in the dictionary. As an example of this kind of application, we may be doing some image processing and have a specific palette of colors, represented by names and `(R, G, B)` tuples. If we use the **GNU Image Manipulation Program** (**GIMP**) file format, the color palette may look like the following command snippet:

```
GIMP Palette
Name: Small
Columns: 3
#
0 0 0 Black
255 255 255 White
238 32 77 Red
28 172 120 Green
31 117 254 Blue
```

The details of parsing this file are the subject of *Chapter 6, Recursions and Reductions*. What's important is the results of the parsing.

First, we'll define a `typing.NamedTuple` class `Color` as follows:

```
from typing import NamedTuple

class Color(NamedTuple):
    red: int
    green: int
    blue: int
    name: str
```

Second, we'll assume that we have a parser that produces an iterable of `Color` objects. If

we materialized it as a tuple, it would look like the following:

```
>>> palette = [
...     Color(red=239, green=222, blue=205, name='Almond'),
...     Color(red=205, green=149, blue=117, name='Antique Brass'),
...     Color(red=253, green=217, blue=181, name='Apricot'),
...     Color(red=197, green=227, blue=132, name='Yellow Green'),
...     Color(red=255, green=174, blue=66, name='Yellow Orange')
... ]
```

To locate a given color name quickly, we will create a frozen dictionary from this sequence. This is not the only way to get fast lookups of a color by name. We'll look at another option later.

To create a mapping from an iterable sequence of tuples, we will use the `process(wrap(iterable))` design pattern. The following command shows how we can create the color name mapping:

```
>>> name_map = dict((c.name, c) for c in palette)
```

Here are three three parts to the design pattern:

- The source *iterable* is the `palette`. We could formalize this with the hint `Iterable[Color]`.
- The *wrap* is the `(c.name, c)` expression to transform a `Color` object to a `tuple[str, Color]` pair.
- The *process* is the `dict()` function to create a mapping.

The resulting dictionary looks as follows:

```
>>> name_map['Antique Brass']
Color(red=205, green=149, blue=117, name='Antique Brass')
>>> name_map['Yellow Orange']
Color(red=255, green=174, blue=66, name='Yellow Orange')
```

This can also be done using a dictionary comprehension. We leave that as an exercise for the reader.

Now that we've materialized the mapping, we can use this `dict()` object in some later processing for repeated transformations from color name to `(R, G, B)` color numbers. The lookup will be blazingly fast because a dictionary does a rapid transformation from key to hash value followed by lookup in the dictionary.

Using the bisect module to create a mapping

In the previous example, we created a `dict` object to achieve a fast mapping from a color name to a `Color` object. This isn't the only choice; we can use the `bisect` module instead. Using the `bisect` module means that we have to create a sorted sequence, which we can then search. To be perfectly compatible with the dictionary implementation, we can use `collections.Mapping` as the base class.

The `dict` class uses a hash computation to locate items almost immediately. However, this requires allocating a fairly large block of memory. The `bisect` mapping does a search, which doesn't require as much memory, but performance cannot be described as immediate. The performance drops from $O(1)$ to $O(\log n)$. While this is dramatic, the savings in memory can be critical for processing large collections of data.

A static mapping class looks like the following command snippet:

```
import bisect
from collections.abc import Mapping, Iterable
from typing import Any

class StaticMapping(Mapping[str, Color]):
    def __init__(self,
            iterable: Iterable[tuple[str, Color]]
    ) -> None:
        self._data: tuple[tuple[str, Color], ...] = tuple(iterable)
        self._keys: tuple[str, ...] = tuple(sorted(key for key, _ in
        self._data))
```

```
    def __getitem__(self, key: str) -> Color:
        ix = bisect.bisect_left(self._keys, key)
        if (ix != len(self._keys) and self._keys[ix] == key):
            return self._data[ix][1]
        raise ValueError(f"{key!r} not found")

    def __iter__(self) -> Iterator[str]:
        return iter(self._keys)

    def __len__(self) -> int:
        return len(self._keys)
```

This class extends the abstract superclass `collections.Mapping`. It provides an initialization and implementations for three functions missing from the abstract definition. The type of `tuple[str, Color]` defines a specific kind of two-tuple expected by this mapping.

The `__getitem__()` method uses the `bisect.bisect_left()` function to search the collection of keys. If the key is found, the appropriate value is returned. The `__iter__()` method returns an iterator, as required by the superclass. The `__len__()` method, similarly, provides the required length of the collection.

This class may not seem to embody too many functional programming principles. Our goal here is to support a larger application that minimizes the use of stateful variables. This class saves a static collection of key-value pairs. As an optimization, it materializes two objects.

An application would create an instance of this class to perform relatively rapid lookups of values associated with keys. The superclass does not support updates to the object. The collection, as a whole, is stateless. It's not as fast as the built-in `dict` class, but it uses less memory and, through the formality of being a subclass of the `Mapping` class, we can be assured that this object is not used to contain a processing state.

Using stateful sets

Python offers several stateful collections, including the `set` collection. For our purposes, there are two use cases for a set:

- A stateful `set` that accumulates items
- `frozenset`, which can be used to optimize searches for an item

We can create `frozenset` from an iterable in the same way we create a tuple object from an iterable, via a `frozenset(some_iterable)` expression; this will create a structure that has the advantage of a very fast `in` operator. This can be used in an application that gathers data, creates a set, and then uses a `frozenset` to match other data items against the set.

We may have a set of colors that we will use as a kind of **chroma-key**: we will use this color to create a mask that will be used to combine two images. Pragmatically, a single color isn't appropriate, but a small set of very similar colors works best. In this case, we can examine each pixel of an image file to see if the pixel is in the chroma-key set or not. For this kind of processing, the chroma-key colors can be loaded into `frozenset` before processing the target images. The set lookup is extremely fast.

As with mappings—specifically the `Counter` class—there are some algorithms that can benefit from a memoized set of values. Some functions benefit from memoization because a function is a mapping between domain values and range values, a job where mapping works well. A few algorithms benefit from a memoized set, which is stateful and grows as data is processed.

We'll return to memoization in *Chapter 10, The Functools Module*.

Summary

In this chapter, we looked again at writing pure functions free of side effects. We looked at generator functions and how we can use these as the backbone of functional programming to process collections of items. We also examined a number of the built-in collection classes to show how they're used in the functional paradigm. While the general idea behind functional programming is to limit the use of stateful variables, the collection objects have

a stateful implementation. For many algorithms, they're often essential. Our goal is to be judicious in our use of Python's non-functional features.

In the next two chapters, we'll look at functions for processing collections. After that, we'll look closely at higher-order functions: functions that accept functions as arguments as well as returning functions. In later chapters, we'll look at techniques for defining our own higher-order functions. We'll also look at the `itertools` and `functools` modules and their higher-order functions in later chapters.

Exercises

This chapter's exercises are based on code available from Packt Publishing on GitHub. See `https://github.com/PacktPublishing/Functional-Python-Programming-3rd-Editi` on.

In some cases, the reader will notice that the code provided on GitHub includes partial solutions to some of the exercises. These serve as hints, allowing the reader to explore alternative solutions.

In many cases, exercises will need unit test cases to confirm they actually solve the problem. These are often identical to the unit test cases already provided in the GitHub repository. The reader should replace the book's example function name with their own solution to confirm that it works.

Rewrite the some_function() function

In the *Writing pure functions* section, a function was shown that relied on a global variable.

Create a small application that sets the global variable and calls the function. The application can expand on the following example:

```
def some_function ...

def main():
    """
```

```
    >>> main()
    some_function(2, 3, 5)=30
    some_function(2, 3, 5)=34
    """
    global global_adjustment
    global_adjustment = 13
    print(f"{some_function(2, 3, 5)=}")
    global_adjustment = 17
    print(f"{some_function(2, 3, 5)=}")

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

First, create a test suite for `some_function()` and `main()`. A `doctest` suite embedded in docstring is shown in the example.

Second, rewrite `some_function()` to make `global_adjustment` into a parameter. This will lead to revising `main()` and all of the test cases.

Alternative Mersenne class definition

The examples in the *Functions as first-class objects* section show a `Mersenne1` class that accepts a function as a parameter to the `__init__()` method.

An alternative is to provide a plug-in **Strategy** function as part of the class definition.

This would permit the following kinds of object definitions:

```
>>> class ShiftyMersenne(Mersenne2):
...     pow2 = staticmethod(shifty)

>>> m2s = ShiftyMersenne()
>>> m2s(17)
131071
```

The use of `staticmethod()` is essential because the `shifty()` function does not expect the `self` argument when it is evaluated. It's essential to make sure this function is understood

as being "static"—that is, not using a `self` parameter.

Alternative algorithm implementations

Consider the following algorithm:

Algorithm 4 Imperative iteration

Require: $V = \{v_0, v_1, v_2, ..., v_n\}$ and $n \geq 1$.

Require: A function, $f(v)$, that offsets and scales each value.

1: $s \longleftarrow 0$

2: $i \longleftarrow 0$

3: **while** $i \neq n$ **do**

4: $\quad s \longleftarrow s + f(v_i)$

5: $\quad i \longleftarrow i + 1$

6: **end while**

7: $m \longleftarrow \frac{s}{n}$

As we've seen in this chapter, there are three ways to write this in Python:

- As a `for` statement that updates stateful variables
- As a generator expression
- As a `map()` operation to apply the function

Write all three versions in Python.

A test case is the following data:

$$V \longleftarrow \{7.46, 6.77, 12.74, 7.11, 7.81, 8.84, 6.08, 5.39, 8.15, 6.42, 5.73\}$$

And the following scaling function:

$$f(v) = \frac{(v - 7.5)}{2.031}$$

The computed value for m is approximately zero.

Map and filter

The built-in `map()` and `filter()` functions always have an equivalent generator expression. In order to have consistent-looking code, a project team is wrestling with the idea of insisting that all code use generator expressions, avoiding the built-in `map()` and `filter()` functions.

1. Take the side of only using generator expressions and provide reasons for which this is advantageous.
2. Take the side of only using built-in `map()` and `filter()` functions and provide reasons why this alternative might be advantageous.
3. In looking at the reasons for the first two parts of this exercise, is there a clearly articulated decision on which approach is better? If not, why not? If so, what rule should the team use?

Dictionary comprehension

In the *Using stateful mappings* section, we built a mapping from a list of two-tuples. We can also build a mapping using a dictionary comprehension. Rewrite the expression `dict((c.name, c) for c in palette)` as a dictionary comprehension.

Raw data cleanup

A file, `Anscombe.txt`, is almost a valid CSV format file. The problem is there are three lines of useless text at the beginning. The lines are easy to recognize because applying the `float()` function to the values in those header rows will raise a `ValueError` exception.

Some team members suggest using a regular expression to examine the values to see if they are valid numbers. This can be called Look Before You Leap (LBYL):

Algorithm 5 LBYL

Require: R is a CSV reader.

1: **for all** $row \in R$ **do**
2: **if** $\forall\{c$ **is valid** $\mid c \in row\}$ **then**
3: $clean \leftarrow \{float(c) \mid c \in row\}$
4: **yield** $clean$
5: **else**
6: Write a log message
7: **end if**
8: **end for**

Other team members suggest using a simpler `try:` statement to uncover the invalid non-numeric headers and discard them. This can be called Easier to Ask Forgiveness Than Permission (EAFP):

Algorithm 6 EAFP

Require: R is a CSV reader.

1: **for all** $row \in R$ **do**
2: **try**
3: $clean \leftarrow \{float(c) \mid c \in row\}$
4: **yield** clean
5: **except** `ValueError`
6: Write a log message
7: **end for**

Both algorithms work. It's instructive to implement each one in Python to compare them. Here are a few starting points for comparing the algorithms:

- The LBYL variant can rely entirely on generator expressions. However, it requires writing a regular expression that recognizes all possible floating-point values. Is this a responsibility that should be part of this application?

- The EAFP variant needs a separate function to implement the `try:` statement processing. Otherwise, it seems amenable to also being written via generator expressions or the `map()` function.

After building the two variants, which seems to be more expressive of the purpose of filtering and acquiring data?

4

Working with Collections

Python offers a number of functions that process whole collections. They can be applied to sequences (lists or tuples), sets, mappings, and iterable results of generator expressions. We'll look at Python's collection-processing features from a functional programming viewpoint.

We'll start out by looking at iterables and some simple functions that work with iterables. We'll look at some design patterns to handle iterables and sequences with recursive functions as well as explicit `for` statements. We'll look at how we can apply a scalar function to a collection of data with a generator expression.

In this chapter, we'll show you examples of how to use the following functions with collections:

- `any()` and `all()`
- `len()`, `sum()`, and some higher-order statistical processing related to these functions
- `zip()` and some related techniques to structure and flatten lists of data
- `sorted()` and `reversed()` to impose an ordering on a collection

- `enumerate()`

The first four functions can be called **reductions**: they reduce a collection to a single value. The other three functions, `zip()`, `reversed()`, and `enumerate()`, are **mappings**; they produce new collections from existing collections. In the next chapter, we'll look at some more mapping and reduction functions that use an additional function as an argument to customize their processing.

In this chapter, we'll start by looking at ways to process data using generator expressions. Then, we'll apply different kinds of collection-level functions to show how they can simplify the syntax of iterative processing. We'll also look at some different ways of restructuring data.

In the next chapter, we'll focus on using higher-order collection functions to do similar kinds of processing.

An overview of function varieties

We need to distinguish between two broad species of functions, as follows:

- **Scalar functions**: These apply to individual values and compute an individual result. Functions such as `abs()`, `pow()`, and the entire `math` module are examples of scalar functions.
- **Collection functions**: These work with iterable collections.

We can further subdivide these collection functions into three subspecies:

- **Reduction**: This uses a function to fold values in the collection together, resulting in a single final value. For example, if we fold + operations into a sequence of integers, this will compute the sum. This can be also be called an **aggregate function**, as it produces a single aggregate value for an input collection. Functions like `sum()` and `len()` are examples of reducing a collection to a single value.
- **Mapping**: This applies a scalar function to each individual item of a collection; the result is a collection of the same size. The built-in `map()` function does this; a function like `enumerate()` can be seen as a mapping from items to pairs of values.

- **Filter**: This applies a scalar function to all items of a collection to reject some items and pass others. The result is a subset of the input. The built-in `filter()` function does this.

Some functions, for example, `sorted()` and `reversed()`, don't fit this framework in a simple, tidy way. Because these two "reordering" functions don't compute new values from existing values, it seems sensible to set them off to one side.

We'll use this conceptual framework to characterize ways in which we use the built-in collection functions.

Working with iterables

As noted in the previous chapters, Python's `for` statement works with iterables, including Python's rich variety of collections. When working with materialized collections such as tuples, lists, maps, and sets, the `for` statement involves the explicit management of state.

While this strays from purely functional programming, it reflects a necessary optimization for Python. The state management is localized to an iterator object that's created as a part of the `for` statement evaluation; we can leverage this feature without straying too far from pure, functional programming. If, for example, we use the `for` statement's variable outside the indented body of the statement, we've strayed from purely functional programming by leveraging this state control variable.

We'll return to this in *Chapter 6, Recursions and Reductions*. It's an important topic, and we'll just scratch the surface in this section with a quick example of working with generators.

One common application of iterable processing with the `for` statement is the `unwrap(process(wrap(iterable)))` design pattern. A `wrap()` function will first transform each item of an iterable into a two-tuple with a derived sort key and the original item. We can then process these two-tuple items as a single, wrapped value. Finally, we'll use an `unwrap()` function to discard the value used to wrap, which recovers the original item.

This happens so often in a functional context that two functions are used heavily for this; they are the following:

```
from collections.abc import Callable, Sequence
from typing import Any, TypeAlias

Extractor: TypeAlias = Callable[[Sequence[Any]], Any]

fst: Extractor = lambda x: x[0]
snd: Extractor = lambda x: x[1]
```

These two functions pick the first and second values from a two-tuple, and both are handy for the `process()` and `unwrap()` phases of the processing.

Another common pattern is `wrap3(wrap2(wrap1()))`. In this case, we're starting with simple tuples and then wrapping them with additional results to build up larger and more complex tuples. We looked at an example in *Chapter 2*, *Introducing Essential Functional Concepts*, in the *Immutable data* section. A common variation on this theme builds new, more complex named tuple instances from source objects. We might call this the **Accretion** design pattern—an item that accretes derived values.

As an example, consider using the **Accretion** pattern to work with a simple sequence of latitude and longitude values. The first step will convert the simple point represented as a `(lat, lon)` pair on a path into pairs of legs `(begin, end)`. Each pair in the result will be represented as `((lat, lon), (lat, lon))`. The value of `fst(item)` is the starting position; the value of `snd(item)` is the ending position for each value of each item in the collection. We'll expose this design through a series of examples.

In the next sections, we'll show you how to create a generator function that will iterate over the content of a source file. This iterable will contain the raw input data that we will process. Once we have the raw data, later sections will show how to decorate each leg with the haversine distance along the leg. The final result of a `wrap(wrap(iterable()))` design will be a sequence of three tuples: `((lat, lon), (lat, lon), distance)`. We can then analyze the results for the longest and shortest distance, bounding rectangle, and other summaries.

The haversine formula is long-ish, but computes the distance along the surface of a sphere between two points:

$$a = \sqrt{\sin^2\left(\frac{\phi_1 - \phi_2}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\lambda_1 - \lambda_2}{2}\right)}$$

$$d = R \times 2\arcsin(a)$$

The first part, a, is the angle between the two points. The distance, d, is computed from the angle, using the radius of the sphere, R, in the desired units. For a distance in nautical miles, we can use $R = \frac{360 \times 60}{2 \times \pi} \approx 3437.7$. For a distance in kilometers, we can use $R = 6371$.

Parsing an XML file

We'll start by parsing an **Extensible Markup Language** (**XML**) file to get the raw latitude and longitude pairs. This will show you how we can encapsulate some not-quite-functional features of Python to create an iterable sequence of values.

We'll make use of the `xml.etree` module. After parsing, the resulting `ElementTree` object has an `iterfind()` method that will iterate through the available values.

We'll be looking for constructs such as the following XML example:

```
<Placemark><Point>
<coordinates>-76.33029518659048, 37.54901619777347,0</coordinates>
</Point></Placemark>
```

The file will have a number of `<Placemark>` tags, each of which has a point and coordinate structure within it. The coordinate tag's values are East-West longitude, North-South latitude, and altitude above mean sea level. This means there are two tiers of parsing: the XML tier, and then the details of each coordinate. This is typical of **Keyhole Markup Language** (**KML**) files that contain geographic information. (For more information, see `https://developers.google.com/kml/documentation`.)

Extracting data from an XML file can be approached at two levels of abstraction:

- At the lower level, we need to locate the various tags, attribute values, and content within the XML file.
- At a higher level, we want to make useful objects out of the text and attribute values.

The lower-level processing can be approached in the following way:

```
from collections.abc import Iterable
from typing import TextIO
import xml.etree.ElementTree as XML

def row_iter_kml(file_obj: TextIO) -> Iterable[list[str]]:
    ns_map = {
        "ns0": "http://www.opengis.net/kml/2.2",
        "ns1": "http://www.google.com/kml/ext/2.2"
    }
    path_to_points = (
      "./ns0:Document/ns0:Folder/ns0:Placemark/"
      "ns0:Point/ns0:coordinates"
    )
    doc = XML.parse(file_obj)
    text_blocks = (
        coordinates.text
        for coordinates in doc.iterfind(path_to_points, ns_map)
    )
    return (
        comma_split(text)
        for text in text_blocks
        if text is not None
    )
```

This function requires text; generally this will come from a file opened via a `with` statement. The result of this function is a generator that creates list objects from the latitude/longitude pairs. As a part of the XML processing, this function uses a simple static `dict` object, `ns_map`, that provides the namespace mapping information for the XML tags being parsed. This dictionary will be used by the `ElementTree.iterfind()` method to locate only the `<coordinates>` tags in the XML source document.

The essence of the parsing is a generator function that uses the sequence of tags located by `doc.iterfind()`. This sequence of tags is then processed by a `comma_split()` function to tease the text value into its comma-separated components.

The `path_to_points` object is a string that defines how to navigate through the XML structure. It describes the location of the `<coordinates>` tag within the other tags of the document. Using this path means the generator expression will avoid the values of other, irrelevant tags.

The `if text is not None` clause reflects the definition of the `text` attribute of an element tree tag. If there's no body in the tag, the text value will be `None`. While it is extremely unlikely to see an empty `<coordinates/>` tag, the type hints require we handle this case.

The `comma_split()` function has a more functional syntax than the the `split()` method of a string. This function is defined as follows:

```
def comma_split(text: str) -> list[str]:
    return text.split(",")
```

We've used a wrapper to emphasize a slightly more uniform syntax. We've also added explicit type hints to make it clear that a string is converted to a list of `str` values. Without the type hint, there are two potential definitions of `split()` that could be meant. It turns out, this method applies to `bytes` as well as `str`. We've used the `str` type name to narrow the domain of types.

The result of the `row_iter_kml()` function is an iterable sequence of rows of data. Each row will be a list composed of three strings: latitude, longitude, and altitude of a waypoint along this path. This isn't directly useful yet. We'll need to do some more processing to get latitude and longitude as well as converting these two strings into useful floating-point values.

This idea of an iterable sequence of tuples (or lists) allows us to process some kinds of data files in a simple and uniform way. In *Chapter 3*, *Functions, Iterators, and Generators*, we looked at how **Comma-Separated Values** (**CSV**) files are easily handled as rows of

tuples. In *Chapter 6, Recursions and Reductions*, we'll revisit the parsing idea to compare these various examples.

The output from the `row_iter_kml()` function can be collected by the `list()` function. The following interactive example will read the file and extract the details. The `list()` function will create a single list from each `<coordinate>` tag. The accumulated result object looks like the following example:

```
>>> from pprint import pprint
>>> source_url = "file:./Winter%202012-2013.kml"
>>> with urllib.request.urlopen(source_url) as source:
...     v1 = list(row_iter_kml(source))
>>> pprint(v1)
[['-76.33029518659048', '37.54901619777347', '0'],
 ['-76.27383399999999', '37.840832', '0'],
 ['-76.459503', '38.331501', '0'],
 ...
 ['-76.47350299999999', '38.976334', '0']]
```

These are all string values. To be more useful, it's important to apply some additional functions to the output of this function that will create a usable subset of the data.

Parsing a file at a higher level

After parsing the low-level syntax to transform XML to Python, we can restructure the raw data into something usable in our Python program. This kind of structuring applies to XML, **JavaScript Object Notation** (**JSON**), CSV, YAML, TOML, and any of the wide variety of physical formats in which data is serialized.

We'll aim to write a small suite of generator functions that transforms the parsed data into a form our application can use. The generator functions include some simple transformations on the text that are found by the `row_iter_kml()` function, which are as follows:

- Discarding altitude, which can also be stated as keeping only latitude and longitude
- Changing the order from `(longitude, latitude)` to `(latitude, longitude)`

We can make these two transformations have more syntactic uniformity by defining a utility function, as follows:

```
def pick_lat_lon(
        lon: str, lat: str, alt: str
) -> tuple[str, str]:
    return lat, lon
```

We've created a function to take three argument values and create a tuple from two of them. The type hints are more complex than the function itself. The conversion of source data to usable data often involves selecting a subset of fields, as well as conversion from strings to numbers. We've separated the two problems because these aspects often evolve separately.

We can use this function as follows:

```
from collections.abc import Iterable
from typing import TypeAlias

Rows: TypeAlias = Iterable[list[str]]
LL_Text: TypeAlias = tuple[str, str]

def lat_lon_kml(row_iter: Rows) -> Iterable[LL_Text]:
    return (pick_lat_lon(*row) for row in row_iter)
```

This function will apply the `pick_lat_lon()` function to each row from a source iterator. We've used `*row` to assign each element of the row's three-tuple to separate parameters of the `pick_lat_lon()` function. The function can then extract and reorder the two relevant values from each three-tuple.

To simplify the function definition, we've defined two type aliases: `Rows` and `LL_Text`. These type aliases can simplify a function definition. They can also be reused to ensure that several related functions are all working with the same types of objects. This kind of functional design allows us to freely replace any function with its equivalent, which makes

refactoring less risky.

These functions can be combined to parse the file and build a structure we can use. Here's an example of some code that could be used for this purpose:

```
>>> import urllib
>>> source_url = "file:./Winter%202012-2013.kml"
>>> with urllib.request.urlopen(source_url) as source:
...     v1 = tuple(lat_lon_kml(row_iter_kml(source)))
>>> v1[0]
('37.54901619777347', '-76.33029518659048')
>>> v1[-1]
('38.976334', '-76.47350299999999')
```

This script uses the `request.urlopen()` function to open a source. In this case, it's a local file. However, we can also open a KML file on a remote server. Our objective in using this kind of file opening is to ensure that our processing is uniform no matter what the source of the data is.

The script is built around the two functions that do low-level parsing of the KML source. The `row_iter_kml(source)` expression produces a sequence of text columns. The `lat_lon_kml()` function will extract and reorder the latitude and longitude values. This creates an intermediate result that sets the stage for further processing. The subsequent processing can be designed to be independent of the original format.

The final function provides the latitude and longitude values from a complex XML file using an almost purely functional approach. As the result is iterable, we can continue to use functional programming techniques to process each point that we retrieve from the file.

Purists will sometimes argue that using a `for` statement introduces a non-functional element. To be pure, the iteration should be defined recursively. Since a recursion isn't a good use of Python language features, we'll prefer to sacrifice some purity for a more Pythonic approach.

This design explicitly separates low-level XML parsing from higher-level reorganization of the data. The XML parsing produced a generic tuple of string structure. This is compatible with parsers for other file formats. As one example, the result value is compatible with the output from the CSV parser. When working with SQL databases, it can help to use a similar iterable of tuple structures. This permits a design for higher-level processing that can work with data from a variety of sources.

We'll show you a series of transformations to re-arrange this data from a collection of strings to a collection of waypoints along a route. This will involve a number of transformations. We'll need to restructure the data as well as convert from strings to floating-point values. We'll also look at a few ways to simplify and clarify the subsequent processing steps. We'll use this dataset in later chapters because it's quite complex.

Pairing up items from a sequence

A common restructuring requirement is to make start-stop pairs out of points in a sequence. Given a sequence, $S = \{s_0, s_1, s_2, ..., s_n\}$, we would also want to create a paired sequence, $S = \{(s_0, s_1), (s_1, s_2), (s_2, s_3), ..., (s_{n-1}, s_n)\}$. The first and second items form a pair. The second and third items form the next pair. Note that the pairs overlap; each point (other than the first or last) will be the end of one pair and the start of the next pair.

These overlapping pairs are used to compute distances from point to point using a trivial application of a haversine function. This technique is also used to convert a path of points into a series of line segments in a graphics application.

Why pair up items? Why not insert a few additional lines of code into a function such as this:

```
begin = next(iterable)
for end in iterable:
    compute_something(begin, end)
    begin = end
```

This code snippet will process each leg of the data as a `begin, end` pair. However, the

processing function and the `for` statement to restructure the data are tightly bound, making reuse more complex than necessary. The algorithm for pairing is hard to test in isolation when it is one part of a more complex `compute_something()` function.

Creating a combined function also limits our ability to reconfigure the application. There's no easy way to inject an alternative implementation of the `compute_something()` function. Additionally, we've got a piece of an explicit state, the `begin` variable, which makes life potentially complex. If we try to add features to the body of the `for` statement, we can easily fail to set the `begin` variable correctly when an item in the `iterable` source is filtered out from processing.

We achieve better reuse by separating this pairing function from other processing. Simplification, in the long run, is one of our goals. If we build up a library of helpful primitives such as this pairing function, we can tackle larger problems more quickly and confidently.

Indeed, the `itertools` library (the subject of *Chapter 8*, *The Itertools Module*) includes a `pairwise()` function we can also use to perform this pairing of values from a source iterator. While we can use this function, we'll also look at how to design our own.

There are many ways to pair up the points along the route to create start and stop information for each leg. We'll look at a few here and then revisit this in *Chapter 5*, *Higher-Order Functions*, and again in *Chapter 8*, *The Itertools Module*. Creating pairs can be done in a purely functional way using a recursion:

$$\text{pairs}(l) = \begin{cases} [] & \textbf{if } |l| \leq 1 \\ [(l_0, l_1)] + \text{pairs}(l_{[1:]}) & \textbf{if } |l| > 1 \end{cases}$$

While the mathematical formalism seems simple, it doesn't account for the way item l_1 is both part of the first pair and also the head of the remaining items in $l_{[1:]}$.

The functional ideal is to avoid assigning this value to a variable. Variables—and the

resulting stateful code—can turn into a problem when we try to make a "small" change and misuse the variable's value.

An alternative is to somehow "peek" at the upcoming item in the iterable source of data. This doesn't work out well in Python. Once we've used `next()` to examine the value, it can't be put back into the iterable. This makes a recursive, functional version of creating overlapping pairs a bit too complex to be of any real value.

Our strategy for performing tail-call optimization is to replace the recursion in the mathematical formalism with a `for` statement. In some cases, we can further optimize this into a generator expression. Because this works with an explicit variable to track the state of the computation, it's a better fit for Python, while being less purely functional.

The following code is an optimized version of a function to pair up the points along a route:

```
from collections.abc import Iterator, Iterable
from typing import Any, TypeVar

LL_Type = TypeVar('LL_Type')

def legs(lat_lon_iter: Iterator[LL_Type]) -> Iterator[tuple[LL_Type,
LL_Type]]:
    begin = next(lat_lon_iter)
    for end in lat_lon_iter:
        yield begin, end
        begin = end
```

The version is simpler, quite fast, and free from the stack limits of a recursive definition. It's independent of any particular type of sequence, as it will pair up anything emitted by a sequence generator. As there's no processing function inside the loop, we can reuse the `legs()` function as needed. We could also redesign this function slightly to accept a processing function as a parameter value, and apply the given function to each `(begin, end)` pair that's created.

The type variable, `LL_Type`, is used to clarify precisely how the `legs()` function restructures

the data. The hint says that the input type is preserved on output. The input type is an `Iterator` of some arbitrary type, `LL_Type`; the output will include tuples of the same type, `LL_Type`. No other conversion is implied by the function.

The `begin` and `end` variables maintain the state of the computation. The use of stateful variables doesn't fit the ideal of using immutable objects for functional programming. The optimization, however, is important in Python. It's also invisible to users of the function, making it a Pythonic-functional hybrid.

Note that this function requires an iterable source of individual values. This can be an iterable collection or a generator.

We can think of this function as one that yields the following kind of sequence of pairs:

```
[items[0:2], items[1:3], items[2:4], ..., items[-2:]]
```

Another view of this function using the built-in `zip()` function is as follows:

```
list(zip(items, items[1:]))
```

While informative, this `zip()`-based example only works for sequence objects. The `pairs()` function shown earlier will work for any iterable, including sequence objects. The `legs()` function only works for an `Iterator` object as the source of data. The good news is we can make an iterator object from an iterable collection with the built-in `iter()` function.

Using the iter() function explicitly

From a purely functional viewpoint, all of our iterables can be processed with recursive functions, where the state is managed by the recursive call stack. Pragmatically, processing iterables in Python will often involve evaluation of `for` statements. There are two common situations: collection objects and iterables. When working with a collection object, an `Iterator` object is created by the `for` statement. When working with a generator function, the generator function is an iterator and maintains its own internal state. Often, these

are equivalent, from a Python programming perspective. In rare cases—generally those situations where we have to use the `next()` function explicitly—the two won't be precisely equivalent.

The `legs()` function shown previously has an explicit `next()` evaluation to get the first value from the iterable. This works wonderfully well with generator functions, expressions, and other iterables. It doesn't work with sequence objects such as tuples or lists.

The following code contains three examples to clarify the use of the `next()` and `iter()` functions:

```
# Iterator as input:
>>> list(legs(x for x in range(3)))
[(0, 1), (1, 2)]

# List object as input:
>>> list(legs([0, 1, 2]))
Traceback (most recent call last):
...
TypeError: 'list' object is not an iterator

# Explicit iterator created from list object:
>>> list(legs(iter([0,1,2])))
[(0, 1), (1, 2)]
```

In the first case, we applied the `legs()` function to an iterable. In this case, the iterable was a generator expression. This is the expected behavior based on our previous examples in this chapter. The items are properly paired up to create two legs from three waypoints.

In the second case, we tried to apply the `legs()` function to a sequence. This resulted in an error. While a list object and an iterable are equivalent when used in a `for` statement, they aren't equivalent everywhere. A sequence isn't an iterator; a sequence doesn't implement the `__next__()` special method allowing it to be used by the `next()` function. The `for` statement handles this gracefully, however, by creating an iterator from a sequence automatically.

To make the second case work, we need to explicitly create an iterator from a list object. This permits the `legs()` function to get the first item from the iterator over the list items. The `iter()` function will create an iterator from a list.

Extending an iteration

We have two kinds of extensions we could factor into a `for` statement that processes iterable data. We'll look first at a filter extension. In this case, we may be rejecting values from further consideration. They may be data outliers, or perhaps source data that's improperly formatted. Then, we'll look at mapping source data by performing a simple transformation to create new objects from the original objects. In our case, we'll be transforming strings to floating-point numbers. The idea of extending a simple `for` statement with a mapping, however, applies to many situations. We'll look at refactoring the above `legs()` function. What if we need to adjust the sequence of points to discard a value? This will introduce a filter extension that rejects some data values.

The iterative process we're designing returns pairs without performing any additional application-related processing—the complexity is minimal. Simplicity means we're somewhat less likely to confuse the processing state.

Adding a filter extension to this design could look something like the following code snippet:

```
from collections.abc import Iterator, Iterable, Callable
from typing import TypeAlias

Waypoint: TypeAlias = tuple[float, float]
Pairs_Iter: TypeAlias = Iterator[Waypoint]
Leg: TypeAlias = tuple[Waypoint, Waypoint]
Leg_Iter: TypeAlias = Iterable[Leg]

def legs_filter(
        lat_lon_iter: Pairs_Iter,
        rejection_rule: Callable[[Waypoint, Waypoint], bool]) ->
        Leg_Iter:
```

```
    begin = next(lat_lon_iter)
    for end in lat_lon_iter:
        if rejection_rule(begin, end):
            pass
        else:
            yield begin, end
        begin = end
```

We have plugged in a processing rule to reject certain values. As the `for` statement remains succinct and expressive, we are confident that the processing will be done properly. We have, on the other hand, cluttered up a relatively simple function with two separate collections of features. This kind of clutter is not an ideal approach to functional design.

We haven't really provided much information about the `rejection_rule()` function. This needs to be a kind of condition that applies to a `Leg` tuple to reject the point from further consideration. For example, it may reject `begin == end` to avoid zero-length legs. A handy default value for `rejection_rule` is `lambda s, e: False`. This will preserve all of the legs.

The next refactoring will introduce additional mapping to an iteration. Adding mappings is common when a design is evolving. In our case, we have a sequence of string values. We need to convert these to float values for later use.

The following is one way to handle this data mapping, through a generator expression that wraps a generator function:

```
>>> import urllib
>>> source_url = "file:./Winter%202012-2013.kml"
>>> with urllib.request.urlopen(source_url) as source:
...     trip = list(
...         legs(
...             (float(lat), float(lon))
...             for lat, lon in lat_lon_kml(row_iter_kml(source))
...         )
...     )
```

We've applied the `legs()` function to a generator expression that creates float values from the output of the `lat_lon_kml()` function. We can read this in an inside-out order as well. The `lat_lon_kml()` function's output is transformed into a pair of float values, which is then transformed into a sequence of legs.

This is starting to get complex. We've got a large number of nested functions here. We're applying `float()`, `legs()`, and `list()` to a data generator. One common way of refactoring complex expressions is to separate the generator expression from any materialized collection. We can do the following to simplify the expression:

```
>>> import urllib
>>> source_url = "file:./Winter%202012-2013.kml"
>>> with urllib.request.urlopen(source_url) as source:
...     ll_iter = (
...         (float(lat), float(lon))
...         for lat, lon in lat_lon_kml(row_iter_kml(source))
...     )
...     trip = list(
...         legs(ll_iter)
...     )
```

We've assigned the generator function to a variable named `ll_iter`. This variable isn't a collection object; it's a generator of item two-tuples. We're not using a list comprehension to create an object. We've merely assigned the generator expression to a variable name. We've then used the `ll_iter` variable in a subsequent expression.

The evaluation of the `list()` function actually leads to a proper object being built so that we can print the output. The `ll_iter` variable's items are created only as needed.

There is yet another refactoring we might like to do. In general, the source of the data is something we often want to change. In our example, the `lat_lon_kml()` function is tightly bound in the rest of the expression. This makes reuse difficult when we have a different data source.

In the case where the `float()` operation is something we'd like to parameterize so that

we can reuse it, we can define a function around the generator expression. We'll extract some of the processing into a separate function merely to group the operations. In our case, the string-pair to float-pair is unique to particular source data. We can rewrite a complex float-from-string expression into a simpler function, such as:

```
from collections.abc import Iterator, Iterable
from typing import TypeAlias

Text_Iter: TypeAlias = Iterable[tuple[str, str]]
LL_Iter: TypeAlias = Iterable[tuple[float, float]]

def floats_from_pair(lat_lon_iter: Text_Iter) -> LL_Iter:
    return (
        (float(lat), float(lon))
        for lat, lon in lat_lon_iter
    )
```

The `floats_from_pair()` function applies the `float()` function to the first and second values of each item in the iterable, yielding a two-tuple of floats created from an input value. We've relied on Python's `for` statement to decompose the two-tuple.

The type hints detail the transformation from an iterable sequence of `tuple[str, str]` items to `tuple[float, float]` items. The `LL_Iter` type alias can then be used elsewhere in a complex set of function definitions to show how the float pairs are processed.

We can use this function in the following context:

```
>>> import urllib
>>> source_url = "file:./Winter%202012-2013.kml"
>>> with urllib.request.urlopen(source_url) as source:
...     trip = list(
...         legs(
...             floats_from_pair(
...                 lat_lon_kml(
...                     row_iter_kml(source))))
...     )
```

We're going to create legs that are built from float values that come from a KML file. It's fairly easy to visualize the processing, as each stage in the process is a prefix function. Each function's input is the output from the next function in the nested processing steps. This seems like a natural way to express a pipeline of processing.

When parsing, we often have sequences of string values. For numeric applications, we'll need to convert strings to float, int, or `Decimal` values. This often involves inserting a function such as the `floats_from_pair()` function into a sequence of expressions that clean up the source data.

Our previous output was all strings; it looked like the following code snippet:

```
(('37.54901619777347', '-76.33029518659048'),
 ('37.840832', '-76.27383399999999'),
 ...
 ('38.976334', '-76.47350299999999'))
```

We'll want data like the following code snippet, where we have floats:

```
(((37.54901619777347, -76.33029518659048),
(37.840832, -76.273834)), ((37.840832, -76.273834),
...
((38.330166, -76.458504), (38.976334, -76.473503)))
```

After building this processing pipeline, there are some simplifications available. We'll look at some refactoring in *Chapter 5*, *Higher-Order Functions*. We will revisit this in *Chapter 6*, *Recursions and Reductions*, to see how to apply these simplifications to the file parsing problem.

Applying generator expressions to scalar functions

We'll look at a more complex kind of generator expression to map data values from one kind of data to another. In this case, we'll apply a fairly complex function to individual data values created by a generator.

We'll call these non-generator functions **scalar**, as they work with simple atomic values. To work with collections of data, a scalar function will be embedded in a generator expression.

To continue the example started earlier, we'll provide a haversine function to compute the distance between latitude and longitude values. Technically, these are angles, and some spherical trigonometry is required to convert angles to distances on the surface of the sphere. We can use a generator expression to apply a scalar `haversine()` function to a sequence of pairs from our KML file.

The important part of the `haversine()` function is to compute a distance between two points following the proper spherical geometry of the earth. It can involve some tricky-looking math, but we've provided the whole definition here. We also mentioned this function at the beginning of the *Working with iterables* section.

The `haversine()` function is implemented by the following code:

```
from math import radians, sin, cos, sqrt, asin
from typing import TypeAlias

MI = 3959
NM = 3440
KM = 6371

Point: TypeAlias = tuple[float, float]

def haversine(p1: Point, p2: Point, R: float=NM) -> float:
    lat_1, lon_1 = p1
    lat_2, lon_2 = p2
    Δ_lat = radians(lat_2 - lat_1)
    Δ_lon = radians(lon_2 - lon_1)
    lat_1 = radians(lat_1)
    lat_2 = radians(lat_2)

    a = sqrt(
        sin(Δ_lat / 2) ** 2 +
        cos(lat_1) * cos(lat_2) * sin(Δ_lon / 2) ** 2
```

```
    )
    c = 2 * asin(a)
    return R * c
```

The start and end points, `p1` and `p2`, have type hints to show their structure. The return value is also provided with a hint. The explicit use of a type alias for `Point` makes it possible for the **mypy** tool to confirm that this function is used properly.

For short distances covered by coastal sailors, the equirectangular distance computation is more useful:

$$x = R \times \Delta\lambda \times \cos\frac{\Delta\phi}{2}$$

$$y = R \times \Delta\phi$$

$$d = \sqrt{x^2 + y^2}$$

Where R is the earth's mean radius, $R = \frac{360 \times 60}{2\pi}$ nautical miles. The ϕ values are N-S latitude, and the λ values are E-W longitude. This means (ϕ_0, λ_0) and (ϕ_1, λ_1) are the two points we're navigating between.

See `https://edwilliams.org/avform147.htm` for more information.

The following code is how we could use our collection of functions to examine some KML data and produce a sequence of distances:

```
>>> import urllib
>>> source_url = "file:./Winter%202012-2013.kml"
>>> with urllib.request.urlopen(source_url) as source:
...     trip = (
...         (start, end, round(haversine(start, end), 4))
...         for start,end in
...             legs(
...                 floats_from_pair(
```

```
...                     lat_lon_kml(row_iter_kml(source))
...                 )
...             )
...     )
...     for start, end, dist in trip:
...         print(f"({start} to {end} is {dist:.1f}")
```

The essence of the processing is the generator expression assigned to the `trip` variable. We've assembled three-tuples with a start, end, and the distance from start to end. The start and end pairs come from the `legs()` function. The `legs()` function works with floating-point data built from the latitude-longitude pairs extracted from a KML file.

The output looks like the following command snippet:

```
((37.54901619777347, -76.33029518659048) to (37.840832, -76.273834)
is 17.7
((37.840832, -76.273834) to (38.331501, -76.459503) is 30.7
((38.331501, -76.459503) to (38.845501, -76.537331) is 31.1
((38.845501, -76.537331) to (38.992832, -76.451332) is 9.7
...
```

Each individual processing step has been defined succinctly. The overview, similarly, can be expressed succinctly as a composition of functions and generator expressions.

Clearly, there are several further processing steps we may like to apply to this data. The first, of course, is to use the `format()` method of a string to produce better-looking output.

More importantly, there are a number of aggregate values we'd like to extract from this data. We'll call these values *reductions* of the available data. We'd like to reduce the data to get the maximum and minimum latitude, for example, to show the extreme north and south ends of this route. We'd like to reduce the data to get the maximum distance in one leg as well as the total distance for all legs.

The problem we'll have using Python is that the output generator in the `trip` variable can be used only once. We can't easily perform several reductions of this detailed data. While

we can use `itertools.tee()` to work with the iterable several times, it takes a fair amount of memory. It can be wasteful, also, to read and parse the KML file for each reduction. We can make our processing more efficient by materializing intermediate results as a list object.

In the next section, we look at two specific kinds of reductions that compute a single boolean result from a collection of booleans.

Using any() and all() as reductions

The `any()` and `all()` functions provide boolean reduction capabilities. Both functions reduce a collection of values to a single `True` or `False`. The `all()` function ensures that all items have a true value; the `any()` function ensures that at least one item has a true value. In both cases, these functions rely on the Pythonic concept of "truish", or truthy: values for which the built-in `bool()` function returns `true`. Generally, "falsish" values include `False` and `None`, as well as zero, an empty string, and empty collections. Non-false values are true.

These functions are closely related to a universal quantifier and an existential quantifier used to express mathematical logic. We may, for example, want to assert that all elements in a given collection have a property. One formalism for this could look like the following:

$$(\forall_{x \in S}) \, \text{Prime}(x)$$

We read this as *for all x in S, the function, Prime(x), is true.* We've used the universal quantifier, for all, ∀, in front of the logical expression.

In Python we switch the order of the items slightly to transcribe the logic expression as follows:

```
all(isprime(x) for x in someset)
```

The `all()` function will evaluate the `isprime(x)` function for each distinct value of `x` and

reduce the collection of values to a single `True` or `False`.

The `any()` function is related to the existential quantifier. If we want to assert that no value in a collection is prime, we could use one of these two equivalent expressions:

$$\neg\,(\forall_{x \in S})\,\text{Prime}(x) \equiv (\exists_{x \in S})\,\neg\,\text{Prime}(x)$$

The left side states that it is not the case that all elements in S are prime. The right side asserts that there exists one element in S that is not prime. These two are equivalent; that is, if not all elements are prime, then one element must be non-prime.

This rule of equivalence is called **De Morgan's Law**. It can be stated generally as $\forall x\, P(x) \equiv \neg\exists x\, \neg P(x)$. If some proposition, $P(x)$, is true for all x, there is no x for which $P(x)$ is false.

In Python, we can switch the order of the terms and transcribe these to working code in either of these two forms:

```
not_p_1 = not all(isprime(x) for x in someset)
```

```
not_p_2 = any(not isprime(x) for x in someset)
```

As these two lines are equivalent, there are two common reasons for choosing one over the other: performance and clarity. The performance is nearly identical, so it boils down to clarity. Which of these states the condition the most clearly?

The `all()` function can be described as an *and reduction* of a set of values. The result is similar to folding the `and` operator between the given sequence of values. The `any()` function, similarly, can be described as an *or reduction*. We'll return to this kind of general-purpose reducing when we look at the `reduce()` function in *Chapter 10, The Functools Module*. There's no best answer here; it's a question of what seems most readable to the intended audience.

We also need to look at the degenerate case of these functions. What if the sequence has no elements? What are the values of `all(())` or `all([])`?

Consider a list, `[1, 2, 3]`. The expression `[] + [1, 2, 3] == [1, 2, 3]` is true because the empty list is the identity value for list concatenation. This also works for the `sum(())` function: `sum([]) + sum([1, 2, 3]) == sum([1, 2, 3])`. The sum of an empty list must be the additive identity value for addition, zero.

The `and` identity value is `True`. This is because `True and whatever == whatever`. Similarly, the `or` identity value is `False`. The following code demonstrates that Python follows these rules:

```
>>> all(())
True
>>> any(())
False
```

Python gives us some very nice tools to perform processing that involves logic. We have the built-in `and`, `or`, and `not` operators. However, we also have these collection-oriented `any()` and `all()` functions.

Using len() and sum() on collections

The `len()` and `sum()` functions provide two simple reductions—a count of the elements and the sum of the elements in a sequence. These two functions are mathematically similar, but their Python implementation is quite different.

Mathematically, we can observe this cool parallelism:

- The `len()` function returns the sum of ones for each value in a collection, X: $\sum_{x \in X} 1 = \sum_{x \in X} x^0$.
- The `sum()` function returns the sum of each value in a collection, X: $\sum_{x \in X} x = \sum_{x \in X} x^1$.

The `sum()` function works for any iterable. The `len()` function doesn't apply to iterables; it

only applies to sequences. This little asymmetry in the implementation of these functions is a little awkward around the edges of statistical algorithms.

As noted above, for empty sequences, both of these functions return a proper additive identity value of zero:

```
>>> sum(())
0
>>> len(())
0
```

While `sum(())` returns an integer zero, this isn't a problem when working with float values. When other numeric types are used, the integer zero can be used along with values of the types of the available data. Python's numeric types generally have rules for performing operations with values of other numeric types.

Using sums and counts for statistics

In this section, we'll implement a number of functions useful for statistics. The point is show how functional programming can be applied to the kinds of processing that are common in statistical functions.

Several common functions are described as "measure of central tendency". Functions like the arithmetic mean or the standard deviation provide a summary of a collection of values. A transformation called "normalization" shifts and scales values around a population mean and standard deviation. We'll also look at how to compute a coefficient of correlation to show to what extent two sets of data are related to each other.

Readers might want to look at `https://towardsdatascience.com/descriptive-statistics-f2beeaf7a8df` for more information on descriptive statistics.

The arithmetic mean seems to have an appealingly trivial definition based on `sum()` and `len()`. It looks like the following might work:

```
def mean(items):
    return sum(items) / len(items)
```

This simple-looking function doesn't work for `Iterable` objects. This definition only works for collections that support the `len()` function. This is easy to discover when trying to write proper type annotations. The definition of `mean(items: Iterable[float]) -> float` won't work because more general `Iterable[float]` types don't support `len()`.

Indeed, we have a hard time performing a computation like standard deviation based on iterables. In Python, we must either materialize a sequence object or resort to somewhat more complex processing that computes multiple sums on a single pass through the data. To use simpler functions means using a `list()` to create a concrete sequence that can be processed multiple times.

To pass muster with **mypy**, the definition needs to look like this:

```
from collections.abc import Sequence

def mean(items: Sequence[float]) -> float:
    return sum(items)/len(items)
```

This includes the appropriate type hints to ensure that `sum()` and `len()` will both work for the expected types of data. The **mypy** tool is aware of the arithmetic type matching rules: any value that could be treated as a float would be considered valid. This means that `mean([1, 2, 3])` will be accepted by the **mypy** tool in spite of the values being all integers.

We have some alternative and elegant expressions for mean and standard deviation in the following definitions:

```
import math
from collections.abc import Sequence
```

```
def stdev(data: Sequence[float]) -> float:

    s0 = len(data)  # sum(1 for x in data)
    s1 = sum(data)  # sum(x for x in data)
    s2 = sum(x**2 for x in data)

    mean = s1 / s0
    stdev = math.sqrt(s2 / s0 - mean ** 2)
    return stdev
```

These three sums, `s0`, `s1`, and `s2`, have a tidy, parallel structure. We can easily compute the mean from two of the sums. The standard deviation is a bit more complex, but it's based on the three available sums.

This kind of pleasant symmetry also works for more complex statistical functions, such as correlation and even least-squares linear regression.

The moment of correlation between two sets of samples can be computed from their standardized value. The following is a function to compute the standardized value:

```
def z(x: float, m_x: float, s_x: float) -> float:
    return (x - m_x) / s_x
```

The calculation subtracts the mean, μ_x, from each sample, x, and divides it by the standard deviation, σ_x. This gives us a value measured in units of sigma, σ. For normally-distributed data, a value $\pm 1\sigma$ is expected about two-thirds of the time. More extreme values should be less common. A value outside $\pm 3\sigma$ should happen less than one percent of the time.

We can use this scalar function as follows:

```
>>> d = [2, 4, 4, 4, 5, 5, 7, 9]
>>> list(z(x, mean(d), stdev(d)) for x in d)
[-1.5, -0.5, -0.5, -0.5, 0.0, 0.0, 1.0, 2.0]
```

We've built a list that consists of normalized scores based on some raw data in the variable,

d. We used a generator expression to apply the scalar function, `z()`, to the sequence object.

The `mean()` and `stdev()` functions are based on the examples shown previously:

```
from math import sqrt
from collections.abc import Sequence

def mean(samples: Sequence[float]) -> float:
    return s1(samples)/s0(samples)

def stdev(samples: Sequence[float]) -> float:
    N = s0(samples)
    return sqrt((s2(samples) / N) - (s1(samples) / N) ** 2)
```

The three sum functions, similarly, can be defined as shown in the following code:

```
def s0(samples: Sequence[float]) -> float:
    return sum(1 for x in samples)  # or len(data)

def s1(samples: Sequence[float]) -> float:
    return sum(x for x in samples)  # or sum(data)

def s2(samples: Sequence[float]) -> float:
    return sum(x*x for x in samples)
```

While this is very expressive and succinct, it's a little frustrating because we can't use an iterable here. When evaluating the `mean()` function, for example, both a sum of the iterable and a count of the iterable are required. For the standard deviation, two sums and a count of the iterable are all required. For this kind of statistical processing, we must materialize a sequence object (in other words, create a `list`) so that we can examine the data multiple times.

The following code shows how we can compute the correlation between two sets of samples:

```
def corr(samples1: Sequence[float], samples2: Sequence[float]) ->
float:
    m_1, s_1 = mean(samples1), stdev(samples1)
    m_2, s_2 = mean(samples2), stdev(samples2)
    z_1 = (z( x, m_1, s_1 ) for x in samples1)
    z_2 = (z( x, m_2, s_2 ) for x in samples2)
    r = (
        sum(zx1 * zx2 for zx1, zx2 in zip(z_1, z_2))
        / len(samples1)
    )
    return r
```

This correlation function, `corr()`, gathers basic statistical summaries of the two sets of samples: the mean and standard deviation. Given these summaries, we define two generator functions that will create normalized values for each set of samples. We can then use the `zip()` function (see the next example) to pair up items from the two sequences of normalized values and compute the product of those two normalized values. The average of the product of the normalized scores is the correlation.

The following code is an example of gathering the correlation between two sets of samples:

```
>>> xi = [1.47, 1.50, 1.52, 1.55, 1.57, 1.60, 1.63, 1.65,
... 1.68, 1.70, 1.73, 1.75, 1.78, 1.80, 1.83,]

>>> yi = [52.21,53.12,54.48,55.84,57.20,58.57,59.93,61.29,
... 63.11, 64.47, 66.28, 68.10, 69.92, 72.19, 74.46,]

>>> round(corr(xi, yi), 5)
0.99458
```

We've shown two sequences of data points, `xi` and `yi`. The correlation is over `0.99`, which shows a very strong relationship between the two sequences.

This shows one of the strengths of functional programming. We've created a handy statistical module using a half-dozen functions with definitions that are single expressions.

Interestingly, the the `corr()` function can't easily be reduced to a single expression. (It can be reduced to a single very long expression, but it would be terribly hard to read.) Each internal variable in this function's implementation is used only once. This shows us that the `corr()` function has a functional design, even though it's written out in six separate lines of Python.

Using zip() to structure and flatten sequences

The `zip()` function interleaves values from several iterators or sequences. It will create *n* tuples from the values in each of the *n* input iterables or sequences. We used it in the previous section to interleave data points from two sets of samples, creating two-tuples.

The `zip()` function is a generator. It does not materialize a resulting collection.

The following is an example of code that shows what the `zip()` function does:

```
>>> xi = [1.47, 1.50, 1.52, 1.55, 1.57, 1.60, 1.63, 1.65,
... 1.68, 1.70, 1.73, 1.75, 1.78, 1.80, 1.83,]
>>> yi = [52.21, 53.12, 54.48, 55.84, 57.20, 58.57, 59.93, 61.29,
... 63.11, 64.47, 66.28, 68.10, 69.92, 72.19, 74.46,]

>>> zip(xi, yi)
<zip object at ...>

>>> pairs = list(zip(xi, yi))
>>> pairs[:3]
[(1.47, 52.21), (1.5, 53.12), (1.52, 54.48)]
>>> pairs[-3:]
[(1.78, 69.92), (1.8, 72.19), (1.83, 74.46)]
```

There are a number of edge cases for the `zip()` function. We must ask the following questions about its behavior:

- What happens where then are no arguments at all?

- What happens where there's only one argument?
- What happens when the sequences are different lengths?

As with other functions, such as `any()`, `all()`, `len()`, and `sum()`, we want an identity value as a result when applying the reduction to an empty sequence. For example, `sum(())` should be zero. This concept tells us what the identity value for `zip()` should be.

Clearly, each of these edge cases must produce some kind of iterable output. Here are some examples of code that clarify the behaviors. First, the empty argument list:

```
>>> list(zip())
[]
```

The production of an empty list fits with the idea of a list identity value of `[]`. Next, we'll try a single iterable:

```
>>> list(zip((1,2,3)))
[(1,), (2,), (3,)]
```

In this case, the `zip()` function emitted one tuple from each input value. This, too, makes considerable sense.

Finally, we'll look at the different-length list approach used by the `zip()` function:

```
>>> list(zip((1, 2, 3), ('a', 'b')))
[(1, 'a'), (2, 'b')]
```

This result is debatable. Why truncate the longer list? Why not pad the shorter list with `None` values? This alternate definition of the `zip()` function is available in the `itertools` module as the `zip_longest()` function. We'll look at this in *Chapter 8, The Itertools Module*.

Unzipping a zipped sequence

We can use `zip()` to create a sequence of tuples. We also need to look at several ways to unzip a collection of tuples into separate collections.

We can't fully unzip an iterable of tuples, since we might want to make multiple passes over the data. Depending on our needs, we may need to materialize the iterable to extract multiple values.

The first way to unzip tuples is something we've seen many times: we can use a generator function to unzip a sequence of tuples. For example, assume that the following pairs are a sequence object with two-tuples:

```
>>> p0 = list(x[0] for x in pairs)
>>> p0[:3]
[1.47, 1.5, 1.52]
>>> p1 = list(x[1] for x in pairs)
>>> p1[:3]
[52.21, 53.12, 54.48]
```

This snippet created two sequences. The `p0` sequence has the first element of each two-tuple; the `p1` sequence has the second element of each two-tuple.

Under some circumstances, we can use the multiple assignment of a `for` statement to decompose the tuples. The following is an example that computes the sum of the products:

```
>>> round(sum(p0*p1 for p0, p1 in pairs), 3)
1548.245
```

We used the `for` statement to decompose each two-tuple into `p0` and `p1`.

Flattening sequences

Sometimes, we'll have zipped data that needs to be flattened. That is, we need to turn a sequence of sub-sequences into a single list. For example, our input could be a file that has rows of columnar data. It looks like this:

```
2 3 5 7 11 13 17 19 23 29
31 37 41 43 47 53 59 61 67 71
...
```

We can use `(line.split() for line in file)` to create a sequence from the lines in the source file. Each item within that sequence will be a nested 10-item tuple from the values on a single line.

This creates data in blocks of 10 values. It looks as follows:

```
>>> blocked = list(line.split() for line in file)
>>> from pprint import pprint
>>> pprint(blocked)
[['2', '3', '5', '7', '11', '13', '17', '19', '23', '29'],
 ['31', '37', '41', '43', '47', '53', '59', '61', '67', '71'],
 ...
 ['179', '181', '191', '193', '197', '199', '211', '223', '227',
'229']]
```

This is a start, but it isn't complete. We want to get the numbers into a single, flat sequence. Each item in the input is a 10-tuple; we'd rather not deal with decomposing this one item at a time.

We can use a two-level generator expression, as shown in the following code snippet, for this kind of flattening:

```
>>> len(blocked)
5
>>> (x for line in blocked for x in line)
<generator object <genexpr> at ...>
>>> flat = list(x for line in blocked for x in line)
>>> len(flat)
50
>>> flat[:10]
['2', '3', '5', '7', '11', '13', '17', '19', '23', '29']
```

The first `for` clause assigns each item—a list of 10 values—from the blocked list to the line variable. The second `for` clause assigns each individual string from the line variable to the `x` variable. The final generator is this sequence of values assigned to the `x` variable.

We can understand this via a rewrite as follows:

```
from collections.abc import Iterable
from typing import Any

def flatten(data: Iterable[Iterable[Any]]) -> Iterable[Any]:
    for line in data:
        for x in line:
            yield x
```

This transformation shows us how the generator expression works. The first `for` clause (`for line in data`) steps through each 10-tuple in the data. The second `for` clause (`for x in line`) steps through each item in the first `for` clause.

This expression flattens a sequence-of-sequence structure into a single sequence. More generally, it flattens any iterable that contains iterables into a single, flat iterable. It will work for list-of-list as well as list-of-set or any other combination of nested iterables.

Structuring flat sequences

Sometimes, we'll have raw data that is a flat list of values that we'd like to bunch up into subgroups. In the *Pairing up items from a sequence* section earlier in this chapter, we looked at overlapping pairs. In this section, we're looking at non-overlapping pairs.

One approach is to use the `itertools` module's `groupby()` function to implement this. This will have to wait until *Chapter 8, The Itertools Module*.

Let's say we have a flat list, as follows:

```
>>> flat = ['2', '3', '5', '7', '11', '13', '17', '19', '23', '29',
... '31', '37', '41', '43', '47', '53', '59', '61', '67', '71',
... ]
```

We can write nested generator functions to build a sequence-of-sequence structure from flat data. To do this, we'll need a single iterator that we can use multiple times. The expression looks like the following code snippet:

```
>>> flat_iter = iter(flat)
>>> (tuple(next(flat_iter) for i in range(5))
...     for row in range(len(flat) // 5)
... )
<generator object <genexpr> at ...>

>>> grouped = list(_)
>>> from pprint import pprint
>>> pprint(grouped)
[('2', '3', '5', '7', '11'),
 ('13', '17', '19', '23', '29'),
 ('31', '37', '41', '43', '47'),
 ('53', '59', '61', '67', '71')]
```

First, we create an iterator that exists outside either of the two loops that we'll use to create our sequence-of-sequences. The generator expression uses `tuple(next(flat_iter) for i in range(5))` to create five-item tuples from the iterable values in the `flat_iter` variable. This expression is nested inside another generator that repeats the inner loop the proper number of times to create the required sequence of values.

This works only when the flat list is divided evenly. If the last row has partial elements, we'll need to process them separately.

We can use this kind of function to group data into same-sized tuples, with an odd-sized tuple at the end, using the following definitions:

```
from collections.abc import Sequence
from typing import TypeVar

ItemType = TypeVar("ItemType")
# Flat = Sequence[ItemType]
```

```
# Grouped = list[tuple[ItemType, ...]]

def group_by_seq(n: int, sequence: Sequence[ItemType]) ->
list[tuple[ItemType,...]]:
    flat_iter = iter(sequence)
    full_sized_items = list(
        tuple(next(flat_iter) for i in range(n))
        for row in range(len(sequence) // n)
    )
    trailer = tuple(flat_iter)
    if trailer:
        return full_sized_items +  [trailer]
    else:
        return full_sized_items
```

Within the `group_by_seq()` function, an initial list is built and assigned to the variable `full_sized_items`. Each tuple in this list is of size `n`. If there are leftovers, the trailing items are used to build a tuple with a non-zero length that we can append to the list of full-sized items. If the `trailer` tuple is of the length zero, it can be safely ignored.

The type hints include a generic definition of `ItemType` as a type variable. The intent of a type variable is to show that whatever type is an input to this function will be returned from the function. A sequence of strings or a sequence of floats would both work properly.

The input is summarized as a `Sequence` of items. The output is a `List` of tuples of items. The items are all of a common type, described with the `ItemType` type variable.

This isn't as delightfully simple and functional-looking as other algorithms we've looked at. We can rework this into a simpler generator function that yields an iterable instead of a list.

The following code uses a `while` statement as part of tail recursion optimization:

```
from collections.abc import Iterator
from typing import TypeVar

ItemT = TypeVar("ItemT")

def group_by_iter(n: int, iterable: Iterator[ItemT]) ->
Iterator[tuple[ItemT, ...]]:
    def group(n: int, iterable: Iterator[ItemT]) -> Iterator[ItemT]:
        for i in range(n):
            try:
                yield next(iterable)
            except StopIteration:
                return

    while row := tuple(group(n, iterable)):
        yield row
```

We've created a row of the required length from the input iterable. At the end of the input iterable, the value of `tuple(next(iterable) for i in range(n))` will be a zero-length tuple. This can be the base case of a recursive definition. This was manually optimized into the terminating condition of the `while` statement.

The walrus operator, `:=`, is used to assign the result of the expression `tuple(group(n, iterable))` to a variable, `row`. If this is a non-empty tuple, it will be the output from the `yield` statement. If this is an empty tuple, the loop will terminate.

The type hints have been modified to reflect the way this works with an iterator. These iteration processing techniques are not limited to sequences. Because the internal `group()` function uses `next()` explicitly, it has to be used like this: `group_by_iter(7, iter(flat))`. The `iter()` function must be used to create an iterator from a collection.

We can, as an alternative, use the `iter()` function inside the `group()` function. When presented with a collection, this will create a fresh, new iterator. When presented with an iterator, it will do nothing. This makes the function easier to use.

Structuring flat sequences – an alternative approach

Let's say we have a simple, flat list and we want to create non-overlapping pairs from this list. The following is the data we have:

```
>>> flat = ['2', '3', '5', '7', '11', '13', '17', '19', '23', '29',
...     '31', '37', '41', '43', '47', '53', '59', '61', '67', '71',
... ]
```

We can create pairs using list slices, as follows:

```
>>> pairs = list(zip(flat[0::2], flat[1::2]))
>>> pairs[:3]
[('2', '3'), ('5', '7'), ('11', '13')]
>>> pairs[-3:]
[('47', '53'), ('59', '61'), ('67', '71')]
```

The slice `flat[0::2]` is all of the even positions. The slice `flat[1::2]` is all of the odd positions. If we zip these together, we get a two-tuple. The item at index `[0]` is the value from the first even position, and then the item at index `[1]` is the value from the first odd position. If the number of elements is even, this will produce pairs nicely. If the total number of items is odd, the final item will be dropped. This is a problem with a handy solution.

The `list(zip(...))` expression has the advantage of being quite short. We can follow the approach in the previous section and define our own functions to solve the same problem.

We can also build a solution using Python's built-in features. Specifically, the `*(args)` approach to generate a sequence-of-sequences that must be zipped together. It looks like the following:

```
>>> n = 2
>>> pairs = list(
...     zip(*(flat [i::n] for i in range(n)))
```

```
... )
>>> pairs[:5]
[('2', '3'), ('5', '7'), ('11', '13'), ('17', '19'), ('23', '29')]
```

This will generate n slices: `flat[0::n]`, `flat[1::n]`, `flat[2::n]`, and so on, and `flat[n-1::n]`. This collection of slices becomes the arguments to `zip()`, which then interleaves values from each slice.

Recall that `zip()` truncates the sequence at the shortest list. This means that if the list is not an even multiple of the grouping factor, `n`, items will be dropped. When the list's length, `len(flat)`, isn't a multiple of *n*, we'll see `len(flat) % n` is not zero; this will be the size of the final slice.

If we switch to using the `itertools.zip_longest()` function, then we'll see that the final tuple will be padded with enough `None` values to make it have a length of `n`.

We have two approaches to structuring a list into groups. We need to select the approach based on what will be done if the length of the list is not a multiple of the group size. We can use `zip()` to truncate or `zip_longest()` to add a "padding" constant to make the final group the expected size.

The list slicing approach to grouping data is another way to approach the problem of structuring a flat sequence of data into blocks. As it is a general solution, it doesn't seem to offer too many advantages over the functions in the previous section. As a solution specialized for making two-tuples from a flat list, it's elegantly simple.

Using sorted() and reversed() to change the order

Python's `sorted()` function produces a new list by rearranging the order of items in a list. This is similar to the way the `list.sort()` method changes the order of list.

Here's the important distinction between `sorted(aList)` and `aList.sort()`:

- The `aList.sort()` method modifies the `aList` object. It can only be meaningfully applied to a `list` object.

- The `sorted(aList)` function creates a new list from an existing collection of items. The source object is not changed. Further, a variety of collections can be sorted. A `set` or the keys of a `dict` can be put into order.

There are times when we need a sequence reversed. Python offers us two approaches to this: the `reversed()` function, and slices with reversed indices.

For example, consider performing a base conversion to hexadecimal or binary. The following code is a simple conversion function:

```
from collections.abc import Iterator

def digits(x: int, base: int) -> Iterator[int]:
    if x == 0: return
    yield x % base
    yield from digits(x // base, base)
```

This function uses a recursion to yield the digits from the least significant to the most significant. The value of `x % base` will be the least significant digits of `x` in the base `base`.

We can formalize it as follows:

$$\text{digits}(x, b) = \begin{cases} [] & \textbf{if } x = 0 \\ [x \mod b] + \text{digits}(\lfloor \frac{x}{b} \rfloor, b) & \textbf{if } x > 0 \end{cases}$$

In Python, we can use a long name like `base`. This is uncommon in conventional mathematics, so a single letter, b, is used.

In some cases, we'd prefer the digits to be yielded in the reverse order; most significant first. We can wrap this function with the `reversed()` function to swap the order of the digits:

```
def to_base(x: int, base: int) -> Iterator[int]:
    return reversed(tuple(digits(x, base)))
```

The `reversed()` function produces an iterable, but the argument value must be a collection object. The function then yields the items from that object in the reverse order. While a dictionary can be reversed, the operation is an iterator over the keys of the dictionary.

We can do a similar kind of thing with a slice, such as `tuple(digits(x, base))[::-1]`. The slice, however, is not an iterator. A slice is a materialized object built from another materialized object. In this case, for such small collections of values, the allocation of extra memory for the slice is minor. As the `reversed()` function uses less memory than creating slices, it can be advantageous for working with larger collections.

The "Martian Smiley", `[:]`, is an edge case for slicing. The expression `some_list[:]` is a copy of the list made by taking a slice that includes all the items in order.

Using enumerate() to include a sequence number

Python offers the `enumerate()` function to apply index information to values in a sequence or iterable. It performs a specialized kind of wrap that can be used as part of an `unwrap(process(wrap(data)))` design pattern.

It looks like the following code snippet:

```
>>> xi[:3]
[1.47, 1.5, 1.52]
>>> len(xi)
15

>>> id_values = list(enumerate(xi))
>>> id_values[:3]
[(0, 1.47), (1, 1.5), (2, 1.52)]
```

```
>>> len(id_values)
15
```

The `enumerate()` function transformed each input item into a pair with a sequence number and the original item. It's similar to the following:

```
zip(range(len(source)), source)
```

An important feature of `enumerate()` is that the result is an iterable and it works with any iterable input.

When looking at statistical processing, for example, the `enumerate()` function comes in handy to transform a single sequence of values into a more proper time series by prefixing each sample with a number.

Summary

In this chapter, we saw detailed ways to use a number of built-in reductions.

We've used `any()` and `all()` to do essential logic processing. These are tidy examples of reductions using a simple operator, such as `or` or `and`. We've also looked at numeric reductions such as `len()` and `sum()`. We've applied these functions to create some higher-order statistical processing. We'll return to these reductions in *Chapter 6, Recursions and Reductions*.

We've also looked at some of the built-in mappings. The `zip()` function merges multiple sequences. This leads us to look at using this in the context of structuring and flattening more complex data structures. As we'll see in examples in later chapters, nested data is helpful in some situations and flat data is helpful in others. The `enumerate()` function maps an iterable to a sequence of two-tuples. Each two-tuple has the sequence number at index `[0]` and the original value at index `[1]`.

The `reversed()` function iterates over the items in a sequence object, with their original

order reversed. Some algorithms are more efficient at producing results in one order, but we'd like to present these results in the opposite order. The `sorted()` function imposes an order either based on the direct comparisons of objects, or by using a key function to compare a derived value of each object.

In the next chapter, we'll look at the mapping and reduction functions that use an additional function as an argument to customize their processing. Functions that accept a function as an argument are our first examples of higher-order functions. We'll also touch on functions that return functions as a result.

Exercises

This chapter's exercises are based on code available from Packt Publishing on GitHub. See `https://github.com/PacktPublishing/Functional-Python-Programming-3rd-Edition`.

In some cases, the reader will notice that the code provided on GitHub includes partial solutions to some of the exercises. These serve as hints, allowing the reader to explore alternative solutions.

In many cases, exercises will need unit test cases to confirm they actually solve the problem. These are often identical to the unit test cases already provided in the GitHub repository. The reader should replace the book's example function name with their own solution to confirm that it works.

Palindromic numbers

See Project Euler problem number 4, `https://projecteuler.net/problem=4`. The idea here is to locate a number that has a specific property. In this exercise, we want to look at the question of a number being (or not being) a palindrome.

One way to handle this is to decompose a number into a sequence of decimal digits. We can then examine the sequence of decimal digits to see if it forms a proper palindrome.

See the *Using sorted() and reversed() to change the order* section for a snippet of code to

extract digits in base 10 from a given number. Do we need the digits in the conventional order of most-significant digits first? Does it matter if the digits are generated in reverse order?

We can leverage this function in two separate ways to check for a palindrome:

- Compare positions within a sequence of digits, `d[0] == d[-1]`. We only need to compare the first half of the digits with the second half. Be sure your algorithm handles an odd number of digits correctly.
- Use `reversed()` to create a second sequence of digits and compare the two sequences. This is waste of time and memory, but may be easier to understand.

Implement both alternatives and compare the resulting code for clarity and expressiveness.

Hands of cards

Given five cards, there are a number of ways the five cards form groups. The full set of hands for a game like poker is fairly complex. A simplified set of hands, however, provides a tool for establishing whether data is random or not. Here are the hand types we are interested in:

1. All five cards match.
2. Four of the five cards match.
3. Three of the five cards match. Unlike poker, we'll ignore whether or not the other two cards match.
4. There are two separate matching pairs.
5. Two cards match, forming a single pair.
6. No cards match.

For truly random data, the probabilities can be computed with some clever math. A good random number generator allows us to build a simulation that provides expected values.

To get started, we need a function to distinguish which flavor of hand is represented by five random values in the domain 1 to 13 (inclusive). The input is a list of five values. The output should be a numeric code for which of the six kinds of hands has been found.

The broad outline of this function is the following:

Algorithm 7 Classify hands

Require: $H = \{c_0, c_1, c_2, c_3, c_4\}$

1: **if** H has 5 of the same value **then return** 1
2: **else if** H has 4 of the same value **then return** 2
3: **else if** H has 3 of the same value **then return** 3
4: **else if** H has 2 of the one value and 2 of another value **then return** 4
5: **else if** H has 2 of the same value **then return** 5
6: **else return** 6
7: **end if**

Hint: For the more general poker-hand identification, it can help to sort the values into ascending order. For this simplified algorithm, it helps to convert the list into a `Counter` object and examine the frequencies of the various card ranks. The `Counter` class is defined in the `collections` module with many other useful collection classes.

Each of the hands can be recognized by a function of the form:

```
    def hand_flavor(cards: Sequence[int]) -> bool:
        examine the cards
```

This lets us write each individual hand-detection algorithm separately. We can then test them in isolation. This gives us confidence the overall hand classifier will work. It means you'll need to write test cases on the individual classifiers to be sure they work properly.

Replace legs() with pairwise()

In the *Pairing up items from a sequence* section, we looked at the design for a `legs()` function to create pairs of legs from a sequence of waypoints.

This function can be replaced with `itertools.pairwise()`. After making this change, determine which implementation is faster.

Expand legs() to include processing

In the *Pairing up items from a sequence* section, we looked at the design for a `legs()` function to create pairs of legs from a sequence of waypoints.

A design alternative is to incorporate a function into the processing of `legs()` to perform a computation on each pair that's created.

The function might look the following:

```
RT = TypeVar("RT")

def legs(transform: Callable[[LL_Type, LL_Type], RT], lat_lon_iter:
Iterator[LL_Type]) -> Iterator[RT]:
    begin = next(lat_lon_iter)
    for end in lat_lon_iter:
        yield transform(begin, end)
        begin = end
```

This changes the design of subsequent examples. Follow this design change through subsequent examples to see if this leads to simpler, easier-to-understand Python function definitions.

5

Higher-Order Functions

A very important feature of the functional programming paradigm is higher-order functions. We'll look at these three varieties of higher-order functions:

- Functions that accept functions as one (or more) of their arguments
- Functions that return a function
- Functions that accept a function and return a function, a combination of the preceding two features

We'll look at the built-in higher-order functions in this chapter. Separate from these functions, we'll look at a few of the library modules that offer higher-order functions in later chapters after introducing the concepts here.

Functions that accept functions and create functions include complex callable classes as well as function decorators. We'll defer consideration of decorators until *Chapter 12*, *Decorator Design Techniques*.

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

- `max()` and `min()`

- `map()`
- `filter()`
- `iter()`
- `sorted()`

Additionally, we'll look at the `itemgetter()` function in the `operator` module. This function is useful for extracting an item from a sequence.

We'll also look at lambda forms that we can use to simplify using higher-order functions.

The `max()` and `min()` functions are reductions; they create a single value from a collection. The other functions are mappings. They don't reduce the input to a single value.

The `max()`, `min()`, and `sorted()` functions have both a default behavior as well as a higher-order function behavior. If needed, a function can be provided via the `key=` argument. There is a meaningful default behavior for these functions.

The `map()` and `filter()` functions take the function as the first positional argument. Here, the function is required because there is no default behavior.

There are a number of higher-order functions in the `itertools` module. We'll look at this module in *Chapter 8*, *The Itertools Module*, and *Chapter 9*, *Itertools for Combinatorics – Permutations and Combinations*.

Additionally, the `functools` module provides a general-purpose `reduce()` function. We'll look at this in *Chapter 10*, *The Functools Module*, because it requires a bit more care to use. We need to avoid transforming an inefficient algorithm into a nightmare of excessive processing.

Using max() and min() to find extrema

The `max()` and `min()` functions each have a dual life. They are simple functions that apply to collections. They are also higher-order functions. We can see their default behavior as

follows:

```
>>> max(1, 2, 3)
3
>>> max((1,2,3,4))
4
```

Both functions will accept an indefinite number of arguments. The functions are designed to also accept a sequence or an iterable as the only argument and locate the max (or min) of that iterable. When applied to a mapping collection, they will locate the maximum (or minimum) key value.

They also do something more sophisticated. Let's say we have our trip data from the examples in *Chapter 4*, *Working with Collections*. We have a function that will generate a sequence of tuples that looks as follows:

```
[
 ((37.54901619777347, -76.33029518659048), (37.840832, -76.273834),
17.7246),
 ((37.840832, -76.273834), (38.331501, -76.459503), 30.7382),
 ((38.331501, -76.459503), (38.845501, -76.537331), 31.0756),
 ((36.843334, -76.298668), (37.549, -76.331169), 42.3962),
 ((37.549, -76.331169), (38.330166, -76.458504), 47.2866),
 ((38.330166, -76.458504), (38.976334, -76.473503), 38.8019)
]
```

Each tuple in this collection has three values: a starting location, an ending location, and a distance. The locations are given in latitude and longitude pairs. The east latitude is positive; these are points along the US East Coast, about 76° west. The distances between points are in nautical miles.

We have three ways of getting the maximum and minimum distances from this sequence of values. They are as follows:

- Extract the distance with a generator function. This will give us only the distances,

as we've discarded the other two attributes of each leg. This won't work out well if we have any additional processing requirements based on the latitude or longitude.

- Use the `unwrap(process(wrap()))` pattern. This will give us the legs with the longest and shortest distances. From these, we can extract the distance or the point as needed.
- Use the `max()` and `min()` functions as higher-order functions, inserting a function that does the extraction of the important distance values. This will also preserve the original objects with all of their attributes.

To provide context, the following script builds the overall trip:

```
>>> from Chapter04.ch04_ex1 import (
...     floats_from_pair, float_lat_lon, row_iter_kml, haversine, legs
... )
>>> import urllib.request
>>> data = "file:./Winter%202012-2013.kml"

>>> with urllib.request.urlopen(data) as source:
...     path = floats_from_pair(float_lat_lon(row_iter_kml(source)))
...     trip = list(
...         (start, end, round(haversine(start, end), 4))
...         for start, end in legs(path)
...     )
```

The resulting `trip` object is a list object, containing the individual legs. Each leg is a three-tuple with the starting point, the ending point, and the distance, computed with the `haversine()` function. The `leg()` function creates start-end pairs from the overall path of points in the original KML file. The `list()` function consumes values from the lazy generator to materialize the list of legs.

Once we have the `trip` object, we can extract distances and compute the maximum and minimum of those distances. The code to do this with a generator function looks as follows:

```
>>> longest = max(dist for start, end, dist in trip)
>>> shortest = min(dist for start, end, dist in trip)
```

We've used a generator function to extract the relevant item from each leg of the trip tuple. We've had to repeat the generator expression because the expression `dist for start, end, dist in trip` can be consumed only once.

Here are the results based on a larger set of data than was shown previously:

```
>>> longest
129.7748
>>> shortest
0.1731
```

It may help to refer to *Chapter 2*, *Introducing Essential Functional Concepts*, for examples of the wrap-process-unwrap design pattern.

The following is a version of the `unwrap(process(wrap()))` pattern applied to this data:

```
from collections.abc import Iterator, Iterable
from typing import Any

def wrap(leg_iter: Iterable[Any]) -> Iterable[tuple[Any, Any]]:
    return ((leg[2], leg) for leg in leg_iter)

def unwrap(dist_leg: tuple[Any, Any]) -> Any:
    distance, leg = dist_leg
    return leg
```

We can use these functions as follows:

```
>>> longest = unwrap(max(wrap(trip)))
>>> longest
((27.154167, -80.195663), (29.195168, -81.002998), 129.7748)

>>> short = unwrap(min(wrap(trip)))
>>> short
((35.505665, -76.653664), (35.508335, -76.654999), 0.1731)
```

The final and most important form uses the higher-order function feature of the `max()` and

`min()` functions. We'll define a helper function first and then use it to reduce the collection of legs to the desired summaries by executing the following code snippet:

```
def by_dist(leg: tuple[Any, Any, Any]) -> Any:
    lat, lon, dist = leg
    return dist
```

We can use this function as the `key=` argument value to the built-in `max()` function. It looks like the following:

```
>>> longest = max(trip, key=by_dist)
>>> longest
((27.154167, -80.195663), (29.195168, -81.002998), 129.7748)

>>> short = min(trip, key=by_dist)
>>> short
((35.505665, -76.653664), (35.508335, -76.654999), 0.1731)
```

The `by_dist()` function picks apart the three items in each leg tuple and returns the distance item. We'll use this with the `max()` and `min()` functions.

The `max()` and `min()` functions both accept an iteratable and a function as arguments. The keyword parameter `key=` is used by many of Python's higher-order functions to provide a function that will be used to extract the necessary key value.

Using Python lambda forms

In many cases, the definition of a helper function seems to require too much code. Often, we can digest the `key=` function to a single expression. It can seem wasteful to have to write both `def` and `return` statements to wrap a single expression.

Python offers the lambda form as a way to simplify using higher-order functions. A lambda form allows us to define a small, anonymous function. The function's body is limited to a single expression.

The following is an example of using a simple lambda expression as the `key=` function:

```
>>> longest = max(trip, key=lambda leg: leg[2])
>>> shortest = min(trip, key=lambda leg: leg[2])
```

The lambda we've used will be given an item from the sequence; in this case, each leg three-tuple will be given to the lambda. The lambda argument variable, `leg`, is assigned and the expression, `leg[2]`, is evaluated, plucking the distance from the three-tuple.

In cases where a lambda is used exactly once, this form is ideal. When reusing a lambda, it's important to avoid copy and paste. In the above example, the lambda is repeated, a potential software maintenance nightmare. What's the alternative?

We can assign lambdas to variables, by doing something like this:

```
start = lambda x: x[0]
end = lambda x: x[1]
dist = lambda x: x[2]
```

Each of these lambda forms is a callable object, similar to a defined function. They can be used like a function.

The following is an example at the interactive prompt:

```
>>> longest = ((27.154167, -80.195663), (29.195168, -81.002998),
129.7748)
>>> dist(longest)
129.7748
```

Here are two reasons for avoiding this technique:

- PEP 8, the style guide for Python code, advises against assigning lambda objects to variables. See `https://peps.python.org/pep-0008/` for more information.
- The `operator` module provides a generic item getter, `itemgetter()`. This is a higher-order function that returns a function we can use instead of a lambda object.

To extend this example, we'll look at how we get the latitude or longitude value of the

starting or ending point.

The following is a continuation of the interactive session:

```
>>> from operator import itemgetter
>>> start = itemgetter(0)
>>> start(longest)
(27.154167, -80.195663)

>>> lat = itemgetter(0)
>>> lon = itemgetter(1)
>>> lat(start(longest))
27.154167
```

We've imported the `itemgetter()` function from the `operator` module. The value returned by this function is a function that will grab the requested item from a sequence. In the first part of the example, the `start()` function will extract item `0` from a sequence.

The `lat()` and `lon()` functions, similarly, are created by the `itemgetter()` function. Note that the complexity of the nested tuples in the data structure must be carefully paralleled with the `itemgetter()` functions.

There's no clear advantage to using lambda objects or `itemgetter()` functions as a way to extract fields over defining a `typing.NamedTuple` class or a dataclass. Using lambdas (or better, the `itemgetter()` function) does allow the code to rely on prefix function notation, which might be easier to read in a functional programming context. We can gain a similar advantage by using the `operator.attrgetter` function to extract a specific attribute from a `typing.NamedTuple` class or dataclass. Using `attrgetter` duplicates a name. For example, a `typing.NamedTuple` class with an attribute of `lat` may also use `attrgetter('lat')`; this can make it slightly harder to locate all references to an attribute when refactoring.

Lambdas and the lambda calculus

If Python were a purely functional programming language, it would be necessary to explain Church's lambda calculus, and the technique invented by Haskell Curry that we call

currying. Python, however, doesn't stick closely to the lambda calculus. Functions are not curried to reduce them to single-argument lambda forms.

Python lambda forms are not restricted to single-argument functions. They can have any number of arguments. They are restricted to a single expression, however.

We can, using the `functools.partial` function, implement currying. We'll save this for *Chapter 10, The Functools Module*.

Using the map() function to apply a function to a collection

A scalar function maps values from a domain to a range. When we look at the `math.sqrt()` function, as an example, we're looking at a mapping from a float value, `x`, to another float value, `y = sqrt(x)`, such that $y^2 = x$. The domain is limited to non-negative values for the `math` module. When using the `cmath` module, any number can be used, and the results can be complex numbers.

The `map()` function expresses a similar concept; it maps values from one collection to create another collection. It assures that the given function is used to map each individual item from the domain collection to the range collection—this is the ideal way to apply a built-in function to a collection of data.

Our first example involves parsing a block of text to get a sequence of numbers. Let's say we have the following chunk of text:

```
>>> text= """\
... 2 3 5 7 11 13 17 19 23 29
... 31 37 41 43 47 53 59 61 67 71
... 73 79 83 89 97 101 103 107 109 113
... 127 131 137 139 149 151 157 163 167 173
... 179 181 191 193 197 199 211 223 227 229
... """
```

We can restructure this text using the following generator function:

```
>>> data = list(
...     v
...     for line in text.splitlines()
...         for v in line.split()
... )
```

This will split the text into lines. For each line, it will split the line into space-delimited words and iterate through each of the resulting strings. The results look as follows:

```
['2', '3', '5', '7', '11', '13', '17', '19', '23', '29',
'31', '37', '41', '43', '47', '53', '59', '61', '67', '71',
'73', '79', '83', '89', '97', '101', '103', '107', '109', '113',
'127', '131', '137', '139', '149', '151', '157', '163', '167',
'173', '179', '181', '191', '193', '197', '199', '211', '223',
'227', '229']
```

We still need to apply the `int()` function to each of the string values. This is where the `map()` function excels. Take a look at the following code snippet:

```
>>> list(map(int, data))
[2, 3, 5, 7, 11, 13, 17, 19, ..., 229]
```

The `map()` function applied the `int()` function to each value in the collection. The result is a sequence of numbers instead of a sequence of strings.

The `map()` function's results are iterable. The `map()` function can process any type of iterable.

The idea here is that any Python function can be applied to the items of a collection using the `map()` function. There are a lot of built-in functions that can be used in this map-processing context.

Working with lambda forms and map()

Let's say we want to convert our trip distances from nautical miles to statute miles. We want to multiply each leg's distance by `6076.12/5280`, which is `1.150780`.

We'll rely on a number of `itemgetter` functions to extract data from the data structure. We can combine extractions with computation of new values. We can do this calculation with the `map()` function as follows:

```
>>> from operator import itemgetter
>>> start = itemgetter(0)
>>> end = itemgetter(1)
>>> dist = itemgetter(2)
>>> sm_trip = map(
...     lambda x: (start(x), end(x), dist(x) * 6076.12 / 5280),
...     trip
... )
```

We've defined a lambda that will be applied to each leg in the trip by the `map()` function. The lambda will use the `itemgetter` function to separate the start, end, and distance values from each leg's tuple. It will compute a revised distance and assemble a new leg tuple from the start, end, and statute mile distances.

This is precisely like the following generator expression:

```
>>> sm_trip = (
...     (start(x), end(x), dist(x) * 6076.12 / 5280)
...     for x in trip
... )
```

We've done the same processing on each item in the generator expression.

Using the built-in `map()` function or a generator expression will produce identical results and have nearly identical performance. The choice of using lambdas, named tuples, defined functions, the `operator.itemgetter()` function, or generator expressions is entirely a matter of how to make the resulting application program succinct and expressive.

Using map() with multiple sequences

Sometimes, we'll have two collections of data that need to be parallel to each other. In *Chapter 4*, *Working with Collections*, we saw how the `zip()` function can interleave two sequences to create a sequence of pairs. In many cases, we're really trying to do something like the following:

```
map(function, zip(one_iterable, another_iterable))
```

We're creating argument tuples from two (or more) parallel iterables and applying a function to the argument tuple. This can be awkward because the parameters to the given function, `function()`, will be a single two-tuple; the argument values will not be applied to each parameter.

As a consequence, we can think about using the following technique to decompose the tuple into two individual parameters:

```
(
    function(x, y)
    for x, y in zip(one_iterable, another_iterable)
)
```

Here, we've replaced the `map()` function with an equivalent generator expression. `for x, y` decomposes the two-tuples so we can apply them to each parameter of the function.

There is a better approach that is already available to us. Let's look at a concrete example of the alternate approach.

In *Chapter 4*, *Working with Collections*, we looked at trip data that we extracted from an XML file as a series of waypoints. We needed to create legs from this list of waypoints that show the start and end of each leg.

The following is a simplified version that uses the `zip()` function applied to two slices of a sequence:

```
>>> waypoints = range(4)
>>> zip(waypoints, waypoints[1:])
<zip object at ...>

>>> list(zip(waypoints, waypoints[1:]))
[(0, 1), (1, 2), (2, 3)]
```

We've created a sequence of pairs drawn from a single flat list. Each pair will have two adjacent values. The `zip()` function stops when the shorter list is exhausted. This `zip(x, x[1:])` pattern only works for materialized sequences and the iterable created by the `range()` function. It won't work for iterable objects because the slicing operation isn't implemented.

We created pairs so that we can apply the `haversine()` function to each pair to compute the distance between the two points on the path. The following is how it looks in one sequence of steps:

```
>>> from Chapter04.ch04_ex1 import (
...     floats_from_pair, float_lat_lon, row_iter_kml, haversine
... )
>>> import urllib.request

>>> data = "file:./Winter%202012-2013.kml"
>>> with urllib.request.urlopen(data) as source:
...     path_gen = floats_from_pair(
...         float_lat_lon(row_iter_kml(source)))
...     path = list(path_gen)

>>> distances_1 = map(
...     lambda s_e: (s_e[0], s_e[1], haversine(*s_e)),
...     zip(path, path[1:])
... )
```

We've built a list of waypoints, and labeled this with the `path` variable. This is an ordered sequence of latitude-longitude pairs. As we're going to use the `zip(path, path[1:])`

design pattern, we must have a materialized sequence and not an iterable.

The results of the `zip()` function will be pairs that have a start and end. We want our output to be a triple with the start, end, and distance. The lambda we're using will decompose the original start-end two-tuple and create a new three-tuple from the start, end, and distance.

We can simplify this by using a clever feature of the `map()` function, which is as follows:

```
>>> distances_2 = map(
...     lambda s, e: (s, e, haversine(s, e)),
...     path, path[1:]
... )
```

Note that we've provided a lambda object and two iterables to the `map()` function. The `map()` function will take the next item from each iterable and apply those two values as the arguments to the given function. In this case, the given function is a lambda that creates the desired three-tuple from the start, end, and distance.

The formal definition for the `map()` function states that it will do **star-map** processing with an indefinite number of iterables. It will take items from each iterable to create a tuple of argument values for the given function. This saves us from having to add the `zip` function to combine sequences.

Using the filter() function to pass or reject data

The job of the `filter()` function is to use and apply a decision function called a **predicate** to each value in a collection. When the predicate function's result is true, the value is passed; otherwise, the value is rejected. The `itertools` module includes `filterfalse()` as a variation on this theme. Refer to *Chapter 8*, *The Itertools Module*, to understand the usage of the `itertools` module's `filterfalse()` function.

We might apply this to our trip data to create a subset of legs that are over 50 nautical miles long, as follows:

```
>>> long_legs = list(
...     filter(lambda leg: dist(leg) >= 50, trip)
... )
```

The predicate lambda will be `True` for long legs, which will be passed. Short legs will be rejected. The output contains the 14 legs that pass this distance test.

This kind of processing clearly segregates the filter rule `(lambda leg: dist(leg) >= 50)` from any other processing that creates the trip object or analyzes the long legs.

For another simple example, look at the following code snippet:

```
>>> filter(lambda x: x % 3 == 0 or x % 5 == 0, range(10))
<filter object at ...>
>>> sum(_)
23
```

We've defined a small lambda to check whether a number is a multiple of three or a multiple of five. We've applied that function to an iterable, `range(10)`. The result is an iterable sequence of numbers that are passed by the decision rule.

The numbers for which the lambda is `True` are `[0, 3, 5, 6, 9]`, so these values are passed. As the lambda is `False` for all other numbers, they are rejected.

The _ variable is a special feature of Python's REPL. It is implicitly set to the result of an expression. In the previous example, the `filter(...)` result was assigned to _. On the next line, `sum(_)` consumed the values from the `filter(...)` result.

This is only available in the REPL, and exists to save us a little bit of typing when we're exploring complex functions interactively.

This can also be done with a generator expression by executing the following code:

```
>>> list(x for x in range(10) if x % 3 == 0 or x % 5 == 0)
[0, 3, 5, 6, 9]
```

We can formalize this using the following set comprehension notation:

$$\{x \mid 0 \le x < 10 \wedge (x \equiv 0 \mod 3 \vee x \equiv 0 \mod 5)\}$$

This says that we're building a collection of x values such that x is in `range(10)` and `x % 3 == 0 or x % 5 == 0`. There's a very elegant symmetry between the `filter()` function and formal mathematical set comprehensions.

We often want to use the `filter()` function with defined functions instead of lambda forms. The following is an example of reusing a predicate defined earlier:

```
>>> from Chapter02.ch02_ex1 import isprimeg

>>> list(filter(isprimeg, range(100)))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
67, 71, 73, 79, 83, 89, 97]
```

In this example, we imported a function from another module called `isprimeg()`. We then applied this function to a collection of values to pass the prime numbers and reject any non-prime numbers from the collection.

This can be a remarkably inefficient way to generate a table of prime numbers. The superficial simplicity of this is the kind of thing lawyers call an *attractive nuisance*. It looks like it might be fun, but it doesn't scale well at all. The `isprimeg()` function duplicates all of the testing effort for each new value. Some kind of cache is essential to provide redoing the testing of primality. A better algorithm is the **Sieve of Eratosthenes**; this algorithm retains the previously located prime numbers and uses them to prevent recalculation.

For more information on primality testing, and this algorithm for finding small prime numbers, see `https://primes.utm.edu/prove/prove2_1.html`.

Using filter() to identify outliers

In the previous chapter, we defined some useful statistical functions to compute mean and standard deviation and normalize a value. We can use these functions to locate outliers in our trip data. What we can do is apply the `mean()` and `stdev()` functions to the distance value in each leg of a trip to get the population mean and standard deviation.

We can then use the `z()` function to compute a normalized value for each leg. If the normalized value is more than 3, the data is potentially far from the mean. If we reject these outliers, we have a more uniform set of data that's less likely to harbor reporting or measurement errors.

The following is how we can tackle this:

```
>>> from Chapter04.ch04_ex3 import mean, stdev, z

>>> dist_data = list(map(dist, trip))
>>> μ_d = mean(dist_data)
>>> σ_d = stdev(dist_data)

>>> outlier = lambda leg: abs(z(dist(leg), μ_d, σ_d)) > 3

>>> list(filter(outlier, trip))
```

We've mapped the distance function to each leg in the `trip` collection. The `dist()` function is the function created by `itemgetter(2)`. As we'll do several things with the result, we must materialize a `list` object. We can't rely on the iterator, as the first function in this sequence of steps will consume all of the iterator's values. We can then use this extraction to compute population statistics μ_d and σ_d with the mean and standard deviation.

Given the mean and standard deviation values, we used the `outlier` lambda to filter our data. If the normalized value is too large, the data is an outlier. The threshold for "too far from the mean" can vary based on the kind of distribution. For a normal distribution, the probability of a value being within three standard deviations from the mean is 0.997.

The result of `list(filter(outlier, trip))` is a list of two legs that are quite long compared to the rest of the legs in the population. The average distance is about 34 nm, with a standard deviation of 24 nm.

We're able to decompose a fairly complex problem into a number of independent functions, each of which can be easily tested in isolation. Our processing is a composition of simpler functions. This can lead to succinct, expressive functional programming.

The iter() function with a sentinel value

The built-in `iter()` function creates an iterator over an object of a collection class. The `list`, `dict`, and `set` classes all work with the `iter()` function to provide an iterator object for the items in the underlying collection. In most cases, we'll allow the `for` statement to do this implicitly. In a few cases, however, we need to create an iterator explicitly. One example of this is to separate the head from the tail of a collection.

Other uses of the `iter()` function include building iterators to consume the values created by a callable object (for example, a function) until a sentinel value is found. This feature is sometimes used with the `read()` method of a file to consume items until some end-of-line or end-of-file sentinel value is found. An expression such as `iter(file.read, '\n')` will evaluate the given function until the sentinel value, `'\n'`, is found. This must be used carefully: if the sentinel is not found, it can continue reading zero-length strings forever.

Providing a callable function to `iter()` can be a bit challenging because the function we provide must maintain some state internally. This is generally looked at as undesirable in functional programs.

However, hidden state is a feature of an open file: each `read()` or `readline()` method of a file advances the internal state to the next character or the next line.

Another example of explicit iteration is the way that a mutable collection object's `pop()` method makes a stateful change to a collection object. The following is an example of

using the `pop()` method:

```
>>> source = [1, 2, 3, None, 4, 5, 6]
>>> tail = iter(source.pop, None)
>>> list(tail)
[6, 5, 4]
```

The `tail` variable was set to an iterator over the list `[1, 2, 3, None, 4, 5, 6]` that will be traversed by the `pop()` function. The default behavior of `pop()` is `pop(-1)`; that is, the elements are popped in the reverse order. This makes a stateful change to the list object: each time `pop()` is called, the item is removed, mutating the list. When the sentinel value is found, the iterator stops returning values. If the sentinel is not found, this will break with an `IndexError` exception.

This kind of internal state management is something we'd like to avoid. Consequently, we won't try to contrive a use for this feature.

Using sorted() to put data in order

When we need to produce results in a defined order, Python gives us two choices. We can create a list object and use the `list.sort()` method to put items in an order. An alternative is to use the `sorted()` function. This function works with any iterable, but it creates a final `list` object as part of the sorting operation.

The `sorted()` function can be used in two ways. It can be simply applied to collections. It can also be used as a higher-order function using the `key=` argument.

Let's say we have our trip data from the examples in *Chapter 4*, *Working with Collections*. We have a function that will generate a sequence of tuples with the starting location, end location, and distance for each leg of a trip. The data looks as follows:

```
[
 ((37.54901619777347, -76.33029518659048), (37.840832, -76.273834),
17.7246),
```

```
 ((37.840832, -76.273834), (38.331501, -76.459503), 30.7382),
 ((38.331501, -76.459503), (38.845501, -76.537331), 31.0756),
 ((36.843334, -76.298668), (37.549, -76.331169), 42.3962),
 ((37.549, -76.331169), (38.330166, -76.458504), 47.2866),
 ((38.330166, -76.458504), (38.976334, -76.473503), 38.8019)
]
```

We can see the default behavior of the `sorted()` function using the following interaction:

```
>>> sorted(dist(x) for x in trip)
[0.1731, 0.1898, 1.4235, 4.3155, ... 86.2095, 115.1751, 129.7748]
```

We used a generator expression `(dist(x) for x in trip)` to extract the distances from our trip data. The `dist()` function is the function created by `itemgetter(2)`. We then sorted this iterable collection of numbers to get the distances from 0.17 nm to 129.77 nm.

If we want to keep the legs and distances together in their original three-tuples, we can have the `sorted()` function apply a `key=` function to determine how to sort the tuples, as shown in the following code snippet:

```
>>> sorted(trip, key=dist)
[((35.505665, -76.653664), (35.508335, -76.654999), 0.1731), ...
```

We've sorted the trip data, using the `dist()` function to extract the distance from each tuple. The `dist()` function, shown earlier, is created by the `itemgetter()` function as follows:

```
>>> from operator import itemgetter
>>> dist = itemgetter(2)
```

As an alternative, we can also use a `lambda leg: leg[2]` to select a specific value from a tuple. Providing a name, `dist`, makes it a little more clear which item is being selected from the tuple.

Overview of writing higher-order functions

We'll look at designing our own higher-order functions. We'll summarize some of the process before diving into some more complex kinds of design patterns. We'll start by looking at common data transformations, such as the following:

- Wrap objects to create more complex objects
- Unwrap complex objects into their components
- Flatten a structure
- Structure a flat sequence

These patterns can help to visualize ways higher-order functions can be designed in Python.

It can also help to recall that a `Callable` class definition is a function that returns a callable object. We'll look at this as a way to write flexible functions into which configuration parameters can be injected.

We'll defer deeper consideration of decorators until *Chapter 12*, *Decorator Design Techniques*. A decorator is also a higher-order function, but it consumes one function and returns another, making it more complex than the examples in this chapter. We'll start with developing highly-customized versions of `map()` and `filter()`.

Writing higher-order mappings and filters

Python's two built-in higher-order functions, `map()` and `filter()`, generally handle almost everything we might want to throw at them. It's difficult to optimize them in a general way to achieve higher performance. We'll look at similar functions such as `imap()` in *Chapter 14*, *The Multiprocessing, Threading, and Concurrent.Futures Modules*.

We have three largely equivalent ways to express a mapping. Assume that we have some function, `f(x)`, and some collection of objects, `C`. The ways we can compute a mapping from the domain value in C to a range value are as follows:

- The `map()` function:

```
map(f, C)
```

- A generator expression:

```
(f(x) for x in C)
```

- A generator function with a `yield` statement:

```
from collections.abc import Callable, Iterable, Iterator
from typing import Any

def mymap(f: Callable[[Any], Any], C: Iterable[Any]) ->
Iterator[Any]:
    for x in C:
        yield f(x)
```

  This `mymap()` function can be used as an expression with the function to apply and the iterable source of data:

```
mymap(f, C)
```

Similarly, we have three ways to apply a filter function to a collection, all of which are equivalent:

- The `filter()` function:

```
filter(f, C)
```

- A generator expression:

```
(x for x in C if f(x))
```

- A generator function with a `yield` statement:

```
from collections.abc import Callable, Iterable, Iterator
from typing import Any
```

```
def myfilter(f: Callable[[Any], bool], C: Iterable[Any]) ->
Iterator[Any]:
    for x in C:
        if f(x):
            yield x
```

This `myfilter()` function can be used as an expression with the function to apply and the iterable source of data:

```
myfilter(f, C)
```

There are some minor performance differences; often the `map()` and `filter()` functions are fastest. More importantly, there are different kinds of extensions that fit these mapping and filtering designs, which are as follows:

- If we need to modify the processing, we can create a more sophisticated function, `g(x)`, that is applied to each element. This is the most general approach and applies to all three designs. This is where the bulk of our functional design energy is invested. We may define our new function around the existing `f(x)`, or we may find that we need to refactor the original function. In all cases, this design effort seems to yield the most benefits.
- We can tweak the `for` loop inside the generator expression or generator function. One obvious tweak is to combine mapping and filtering into a single operation by extending the generator expression with an `if` clause. We can also merge the `mymap()` and `myfilter()` functions to combine mapping and filtering. This requires some care to be sure the resulting function is not a clutter of features.

Profound changes that alter the structure of the data handled by the loop often happen as software evolves and matures. We have a number of design patterns, including wrapping, unwrapping (or extracting), flattening, and structuring. We've looked at a few of these techniques in previous chapters.

In the following sections, we'll look at ways to design our own higher-order functions.

We'll start with unwrapping complex data while also applying a mapping function. For each example, it's important to look at where the complexity arises, and decide if the resulting code really is succinct and expressive.

Unwrapping data while mapping

When we use a construct such as `(f(x) for x, y in C)`, we use the multiple assignment feature of the `for` statement to unwrap a multi-valued tuple and then apply a function. The whole expression is a mapping. This is a common Python optimization to change the structure and apply a function.

We'll use our trip data from *Chapter 4, Working with Collections*. The following is a concrete example of unwrapping while mapping:

```
from collections.abc import Callable, Iterable, Iterator
from typing import Any, TypeAlias

Conv_F: TypeAlias = Callable[[float], float]
Leg: TypeAlias = tuple[Any, Any, float]

def convert(
        conversion: Conv_F,
        trip: Iterable[Leg]) -> Iterator[float]:
    return (
        conversion(distance)
        for start, end, distance in trip
    )
```

This higher-order function would be supported by conversion functions that we can apply to our raw data, as follows:

```
from collections.abc import Callable
from typing import TypeAlias

Conversion: TypeAlias = Callable[[float], float]
```

```
to_miles: Conversion = lambda nm: nm * 6076.12 / 5280

to_km: Conversion = lambda nm: nm * 1.852

to_nm: Conversion = lambda nm: nm
```

These have been defined as lambdas and assigned to variables. Some static analysis tools will object to this because PEP-8 frowns on it.

The following shows how we can extract distance and apply a conversion function:

```
>>> convert(to_miles, trip)
<generator object ...>
>>> miles = list(convert(to_miles, trip))
>>> trip[0]
((37.54901619777347, -76.33029518659048), (37.840832, -76.273834),
17.7246)
>>> miles[0]
20.397120559090908
>>> trip[-1]
((38.330166, -76.458504), (38.976334, -76.473503), 38.8019)
>>> miles[-1]
44.652462240151515
```

As we're unwrapping, the result will be a sequence of floating-point values. The results are as follows:

```
[20.397120559090908, 35.37291511060606, ..., 44.652462240151515]
```

This `convert()` function is highly specific to our start-end-distance trip data structure, as the `for` statement decomposes a specific three-tuple.

We can build a more general solution for this kind of **unwrapping-while-mapping** design pattern. It suffers from being a bit more complex. First, we need general-purpose decomposition functions, as in the following code snippet:

```
from collections.abc import Callable
from operator import itemgetter
from typing import TypeAlias

Selector: TypeAlias = Callable[[tuple[Any, ...]], Any]

fst: Selector = itemgetter(0)

snd: Selector = itemgetter(1)

sel2: Selector = itemgetter(2)
```

We'd like to be able to express `f(sel2(s_e_d)) for s_e_d in trip`. This involves functional composition; we're combining a function, such as `to_miles()`, and a selector, such as `sel2()`.

More descriptive names are often more useful than generic names. We'll leave the renaming as an exercise for the reader. We can express functional composition in Python using yet another lambda, as follows:

```
from collections.abc import Callable

to_miles_sel2: Callable[[tuple[Any, Any, float]], float] = (
    lambda s_e_d: to_miles(sel2(s_e_d))
)
```

This gives us a longer but more specialized version of unwrapping and mapping, as follows:

```
>>> miles2 = list(
...     to_miles_sel2(s_e_d) for s_e_d in trip
... )
```

We can compare the higher-order `convert()` function against this generator expression. Both apply a number of transformations. The `convert()` function "conceals" a processing detail—the composition of a tuple as start, end, and distance—with a `for` statement that

decomposes the tuple. This expression exposes this decomposition by including the `sel2()` function as part of the definition of a composite function.

Neither is "better" by any measure. They represent two approaches to exposing or concealing details. In a specific application development context, the exposure (or concealment) might be more desirable.

The same design principle works to create hybrid filters as well as mappings. We'd apply the filter in an `if` clause of the generator expression that was returned.

We can combine mapping and filtering to create yet more complex functions. While it is appealing to create more complex functions, it isn't always valuable. A complex function might not beat the performance of a nested use of the `map()` and `filter()` functions. Generally, we only want to create a more complex function if it encapsulates a concept and makes the software easier to understand.

Wrapping additional data while mapping

When we use a construct such as `((f(x), x) for x in C)`, we've used wrapping to create a multi-valued tuple while also applying a transformational mapping. This is a common technique to save derived results by creating larger constructs. This has the benefit of avoiding recalculation without the liability of complex objects with an internal state change. In this case, the state change is structural and very visible.

This is part of the example shown in *Chapter 4, Working with Collections*, to create the trip data from the path of points. The code looks like this:

```
>>> from Chapter04.ch04_ex1 import (
...     floats_from_pair, float_lat_lon, row_iter_kml, haversine, legs
... )
>>> import urllib.request
>>> data = "file:./Winter%202012-2013.kml"

>>> with urllib.request.urlopen(data) as source:
...     path = floats_from_pair(float_lat_lon(row_iter_kml(source)))
```

```
...     trip = tuple(
...         (start, end, round(haversine(start, end), 4))
...         for start, end in legs(path)
...     )
```

We can revise this slightly to create a higher-order function that separates the wrapping from the other functions. We can refactor this design to create a function that constructs a new tuple including the original tuple and the distance. This function can be defined as follows:

```
from collections.abc import Callable, Iterable, Iterator
from typing import TypeAlias

Point: TypeAlias = tuple[float, float]
Leg_Raw: TypeAlias = tuple[Point, Point]
Point_Func: TypeAlias = Callable[[Point, Point], float]
Leg_D: TypeAlias = tuple[Point, Point, float]

def cons_distance(
        distance: Point_Func,
        legs_iter: Iterable[Leg_Raw]) -> Iterator[Leg_D]:
    return (
        (start, end, round(distance(start,end), 4))
        for start, end in legs_iter
    )
```

This function will decompose each leg into two variables, `start` and `end`. These variables will be `Point` instances, defined as tuples of two float values. These will be used with the given `distance()` function to compute the distance between the points. The function is a callable that accepts two `Point` objects and returns a float result. The result will build a three-tuple that includes the original two `Point` objects and also the calculated float result.

We can then rewrite our trip assignment to apply the `haversine()` function to compute distances, as follows:

```
>>> source_url = "file:./Winter%202012-2013.kml"
>>> with urllib.request.urlopen(source_url) as source:
...     path = floats_from_pair(
...         float_lat_lon(row_iter_kml(source))
...     )
...     trip2 = tuple(
...         cons_distance(haversine, legs(iter(path)))
...     )
```

We've replaced a generator expression with a higher-order function, `cons_distance()`. The function not only accepts a function as an argument, but it also returns a generator expression. In some applications, this larger and more complex processing step is a helpful way to elide unecessary details.

In *Chapter 10, The Functools Module*, we'll show how to use the `partial()` function to set a value for the `R` parameter of the `haversine()` function, which changes the units in which the distance is calculated.

Flattening data while mapping

In *Chapter 4, Working with Collections*, we looked at algorithms that flattened a nested tuple-of-tuples structure into a single iterable. Our goal at the time was simply to restructure some data without doing any real processing. We can create hybrid solutions that combine a function with a flattening operation.

Let's assume that we have a block of text that we want to convert to a flat sequence of numbers. The text looks as follows:

```
>>> text = """2 3 5 7 11 13 17 19 23 29
... 31 37 41 43 47 53 59 61 67 71
... 73 79 83 89 97 101 103 107 109 113
... 127 131 137 139 149 151 157 163 167 173
... 179 181 191 193 197 199 211 223 227 229
... """
```

Each line is a block of 10 numbers. We need to unblock the rows to create a flat sequence of numbers.

This is done with a two-part generator function, as follows:

```
>>> data = list(
...     v
...     for line in text.splitlines()
...         for v in line.split()
... )
```

This will split the text into lines and iterate through each line. It will split each line into words and iterate through each word. The output from this is a list of strings, as follows:

```
['2', '3', '5', '7', '11', '13', '17', '19', '23', '29', '31', '37',
'41', '43', '47', '53', '59', '61', '67', '71', '73', '79', '83',
'89', '97', '101', '103', '107', '109', '113', '127', '131', '137',
'139', '149', '151', '157', '163', '167', '173', '179', '181', '191',
'193', '197', '199', '211', '223', '227', '229']
```

There's an optimization to this, which applies to this specific text. We'll leave that as an exercise for the reader.

To convert the strings to numbers, we must apply a conversion function as well as unwind the blocked structure from its original format, using the following code snippet:

```
from collections.abc import Callable, Iterator
from typing import TypeAlias

Num_Conv: TypeAlias = Callable[[str], float]

def numbers_from_rows(
        conversion: Num_Conv,
        text: str) -> Iterator[float]:
    return (
        conversion(value)
```

```
        for line in text.splitlines()
        for value in line.split()
    )
```

This function has a `conversion` argument, which is a function that is applied to each value that will be emitted. The values are created by flattening using the algorithm shown previously.

We can use this `numbers_from_rows()` function in the following kind of expression:

```
>>> list(numbers_from_rows(float, text))
```

Here we've used the built-in `float()` to create a list of floating-point values from the block of text.

We have many alternatives using mixtures of higher-order functions and generator expressions. For example, we might express this as follows:

```
>>> text = (value
...     for line in text.splitlines()
...         for value in line.split()
... )
>>> numbers = map(float, text)
>>> list(numbers)
```

This might help us understand the overall structure of the algorithm. The principle is called **chunking**: we summarize the details of a function with a meaningful name. With this summary, the details are abstracted and we can work with the function as a small concept in a larger context. While we often use higher-order functions, there are times when a generator expression can be clearer.

Structuring data while filtering

The previous three examples combined additional processing with mapping. Combining processing with filtering doesn't seem to be quite as expressive as combining it with mapping. We'll look at an example in detail to show that, although it is useful, it doesn't seem to have as compelling a use case as combining mapping and processing.

In *Chapter 4, Working with Collections*, we looked at structuring algorithms. We can easily combine a filter with the structuring algorithm into a single, complex function. The following is a version of our preferred function to group the output from an iterable:

```
from collections.abc import Iterator
from typing import TypeVar

ItemT = TypeVar("ItemT")

def group_by_iter(
        n: int,
        iterable: Iterator[ItemT]
) -> Iterator[tuple[ItemT, ...]]:
    def group(n: int, iterable: Iterator[ItemT]) -> Iterator[ItemT]:
        for i in range(n):
            try:
                yield next(iterable)
            except StopIteration:
                return

    while row := tuple(group(n, iterable)):
        yield row
```

This will try to assemble a tuple of `n` items taken from an iterable object. If there are any items in the tuple, they are yielded as part of the resulting iterable. In principle, the function then operates recursively on the remaining items from the original iterable. As the recursion has limitations in Python, we've optimized the tail call structure into an explicit `while` statement.

The results of the `group_by_iter()` function is a sequence of *n*-tuples. In the following example, we'll create a sequence of numbers using a filter function, and then group them into 7-tuples:

```
>>> from pprint import pprint
>>> data = list(
...     filter(lambda x: x % 3 == 0 or x % 5 == 0, range(1, 50))
... )
>>> data
[3, 5, 6, 9, 10, ..., 48]
>>> grouped = list(group_by_iter(7, iter(data)))
>>> pprint(grouped)
[(3, 5, 6, 9, 10, 12, 15),
 (18, 20, 21, 24, 25, 27, 30),
 (33, 35, 36, 39, 40, 42, 45),
 (48,)]
```

We can merge grouping and filtering into a single function that does both operations in a single function body. The modification to `group_by_iter()` looks as follows:

```
from collections.abc import Callable, Iterator, Iterable
from typing import Any, TypeAlias

ItemFilterPredicate: TypeAlias = Callable[[Any], bool]

def group_filter_iter(
        n: int,
        predicate: ItemFilterPredicate,
        items: Iterator[ItemT]
) -> Iterator[tuple[ItemT, ...]]:
    def group(n: int, iterable: Iterator[ItemT]) -> Iterator[ItemT]:
        for i in range(n):
            try:
                yield next(iterable)
            except StopIteration:
                return
```

```
    subset = filter(predicate, items)
    # ^-- Added this to apply the filter
    while row := tuple(group(n, subset)):
                               # ^-- Changed to use the filter
        yield row
```

We've added a single line to the `group_by_iter()` function. This application of `filter()` creates a subset. We've changed the `while row := tuple(group(n, subset)):` line to use the subset instead of the original collection of items.

This `group_filter_iter()` function applies the filter predicate function to the source iterable provided as the `items` parameter. As the filter output is itself a non-strict iterable, the subset value isn't computed in advance; the values are created as needed. The bulk of this function is identical to the version shown previously.

We can slightly simplify the context in which we use this function. We can compare the explicit use of `filter()` and this combined function where the `filter()` is implicit. The comparison is in the following example:

```
>>> rule: ItemFilterPredicate = lambda x: x % 3 == 0 or x % 5 == 0
>>> groups_explicit = list(
...     group_by_iter(7, filter(rule, range(1, 50)))
... )
>>> groups = list(
...     group_filter_iter(7, rule, iter(range(1, 50)))
... )
```

Here, we've applied the filter predicate and grouped the results in a single function invocation. In the case of the `filter()` function, it's rarely a clear advantage to apply the filter in conjunction with other processing. It seems as if a separate, visible `filter()` function is more helpful than a combined function.

Building higher-order functions with callables

We can define higher-order functions as callable classes. This builds on the idea of writing generator functions; we'll write callables because we need stateful features of Python, like instance variables. In addition to using statements, we can also apply a static configuration when creating the higher-order functions. The **Strategy** design pattern, in particular, works very nicely to alter the features of callable objects.

What's important about a callable class definition is that the class object, created by the `class` statement, defines a function that emits a function. Commonly, we'll use a callable object to create a composite function that combines functions into something relatively complex.

To emphasize this, consider the following class:

```
from collections.abc import Callable
from typing import Any

class NullAware:
    def __init__(self, some_func: Callable[[Any], Any]) -> None:
        self.some_func = some_func

    def __call__(self, arg: Any) -> Any:
        return None if arg is None else self.some_func(arg)
```

This class is used to create a new function that is null aware. When an instance of this class is created, a function, `some_func`, is provided. The only restriction stated is that `some_func` be `Callable[[Any], Any]`. This means the argument takes a single argument and results in a single result. The resulting object is callable. A single, optional argument is expected. The implementation of the `__call__()` method handles the use of `None` objects as an argument. This method has the effect of making the resulting object `Callable[[Optional[Any]], Optional[Any]]`.

For example, evaluating the `NullAware(math.log)` expression will create a new function that can be applied to argument values. The `__init__()` method will save the given

function in the resulting object. This object is a function that can then be used to process data.

The common approach is to create the new function and save it for future use by assigning it a name, as follows:

```
import math

null_log_scale = NullAware(math.log)

null_round_4 = NullAware(lambda x: round(x, 4))
```

The first example creates a new function, and assigns the name `null_log_scale()`. The second example creates a null-aware function, `null_round_4`, that uses a lambda object as the internal value of the function to apply if the parameter is not `None`. We can then use the function in another context. Take a look at the following example:

```
>>> some_data = [10, 100, None, 50, 60]
>>> scaled = map(null_log_scale, some_data)
>>> [null_round_4(v) for v in scaled]
[2.3026, 4.6052, None, 3.912, 4.0943]
```

This example's `__call__()` method relies entirely on expression evaluation. It's an elegant and tidy way to define composite functions built up from lower-level component functions.

Assuring good functional design

The idea of stateless functional programming requires some care when using Python objects. Objects are typically stateful. Indeed, one can argue that the entire purpose of object-oriented programming is to encapsulate state change into class definitions. Because of this, we find ourselves pulled in opposing directions between functional programming and imperative programming when using Python class definitions to process collections.

The benefit of using a callable object to create a composite function gives us slightly simpler syntax when the resulting composite function is used. When we start working with iterable

mappings or reductions, we have to be aware of how and why we introduce stateful objects.

We'll turn to a fairly complex function that contains the following features:

- It applies a filter to a source of items.
- It applies a mapping to the items which pass the filter.
- It computes a sum of the mapped values.

We can try to define it as a simple higher-order function, but with three separate parameter values, it would be cumbersome to use. Instead, we'll create a callable object that is configured by the filter and mapping functions.

Using objects to configure an object is the **Strategy** design pattern, used in object-oriented programming. Here's a class definition that requires the filter and mapping function in order to create a callable:

```
from collections.abc import Callable, Iterable

class Sum_Filter:
    __slots__ = ["filter", "function"]

    def __init__(self,
            filter: Callable[[float], bool],
            func: Callable[[float], float]) -> None:
        self.filter = filter
        self.function = func

    def __call__(self, iterable: Iterable[float]) -> float:
        return sum(
            self.function(x)
            for x in iterable
            if self.filter(x)
        )
```

This class has two slots in each object; this puts a few constraints on our ability to use the function as a stateful object. It doesn't prevent all modifications to the resulting object, but it limits us to just two attributes. Attempting to add an attribute will result in an exception.

The initialization method, `__init__()`, stows the two function objects, `filter` and `func`, in the object's instance variables. The `__call__()` method returns a value based on a generator expression that uses the two internal function definitions. The `self.filter()` function is used to pass or reject items. The `self.function()` function is used to transform objects that are passed by the `filter()` function.

An instance of this class is a function that has two Strategy functions built into it. We create an instance as follows:

```
count_not_none = Sum_Filter(
    lambda x: x is not None,
    lambda x: 1
)
```

We've built a function named `count_not_none()` that counts the non-`None` values in a sequence. It does this by using a lambda to pass non-`None` values and a function that uses a constant, 1, instead of the actual values present.

Generally, this `count_not_none()` object will behave like any other Python function. We can use the `count_not_None()` function as follows:

```
>>> some_data = [10, 100, None, 50, 60]
>>> count_not_none(some_data)
4
```

This shows a technique for using some of Python's object-oriented programming features to create callable objects that are used in a functional approach to designing and building software. We can delegate some complexity to creating a sophisticated function. Having a single function with multiple features can simplify understanding of the context in which the function is used.

Review of some design patterns

The `max()`, `min()`, and `sorted()` functions have a default behavior without a `key=` function. They can be customized by providing a function that defines how to compute a key from the available data. For many of our examples, the `key=` function has been a simple extraction of available data. This isn't a requirement; the `key=` function can do anything.

Imagine the following method: `max(trip, key=random.randint())`. Generally, we try not to have `key=` functions that do something obscure like this.

The use of a `key=` function is a common design pattern. Functions we design can easily follow this pattern.

We've also looked at the way lambda forms can simplify the application of higher-order functions. One significant advantage of using lambda forms is that it follows the functional paradigm very closely. When writing more conventional functions, we can create imperative programs that might clutter an otherwise succinct and expressive functional design.

We've looked at several kinds of higher-order functions that work with a collection of values. Throughout the previous chapters, we've hinted at several different design patterns for higher-order functions that apply to collection objects and scalar objects. The following is a broad classification:

- **Return a generator**: A higher-order function can return a generator expression. We consider the function higher-order because it didn't return scalar values or collections of values. Some of these higher-order functions also accept functions as arguments.
- **Act as a generator**: Some function examples use the `yield` statement to make them first-class generator functions. The value of a generator function is an iterable collection of values that are evaluated lazily. We suggest that a generator function is essentially indistinguishable from a function that returns a generator expression. Both are non-strict. Both can yield a sequence of values. For this reason, we'll also consider generator functions as higher order. Built-in functions, such as `map()` and `filter()`, fall into this category.

- **Materialize a collection**: Some functions must return a materialized collection object: list, tuple, set, or mapping. These kinds of functions can be of a higher order if they have a function as part of the arguments. Otherwise, they're ordinary functions that happen to work with collections.
- **Reduce a collection**: Some functions work with an iterable to create a scalar result. The `len()` and `sum()` functions are examples of this. We can create higher-order reductions when we accept a function as an argument. We'll return to this in the next chapter.
- **Scalar**: Some functions act on individual data items. These can be higher-order functions if they accept another function as an argument.

As we design our own software, we can pick and choose among these established design patterns.

Summary

In this chapter, we have seen two reductions that are higher-order functions: `max()` and `min()`. We looked at the two central higher-order functions, `map()` and `filter()`. We also looked at `sorted()`.

In addition, we looked at how to use a higher-order function to transform the structure of data. We can perform several common transformations, including wrapping, unwrapping, flattening, and structuring sequences of different kinds.

We looked at two ways to define our own higher-order functions, which are as follows:

- The `def` statement. Similar to a lambda form that we assign to a variable.
- Defining a callable class as a kind of function that emits composite functions.

We can also use decorators to emit composite functions. We'll return to this in *Chapter 12, Decorator Design Techniques*.

In the next chapter, we'll look at the idea of purely functional iteration via recursion. We'll use Pythonic structures to make several common improvements over purely functional techniques. We'll also look at the associated problem of performing reductions from

collections to individual values.

Exercises

This chapter's exercises are based on code available from Packt Publishing on GitHub. See `https://github.com/PacktPublishing/Functional-Python-Programming-3rd-Editi` on.

In some cases, the reader will notice that the code provided on GitHub includes partial solutions to some of the exercises. These serve as hints, allowing the reader to explore alternative solutions.

In many cases, exercises will need unit test cases to confirm they actually solve the problem. These are often identical to the unit test cases provided in the GitHub repository. The reader should replace the book's example function name with their own solution to confirm that it works.

Classification of state

A web application might have a number of servers of various kinds, a database, and installed software components. Someone responsible for website reliability will want to know when things are running reasonably well. When things are broken, they'll want details.

As part of monitoring, a health application can gather status from the various components and summarize the status into an overall "health" value. The idea is to perform a kind of "reduce" on the status information.

Each individual service has a status URL that can be pinged for status information. The results can take one of these four values:

- No response at all. The service is not working. This is bad.
- A response that is outside a healthy time window. Even if the response is `"working"`, the service is degraded.
- A response of `"not working"`. A response of "not working" is almost as bad as no response at all. It indicates a severe problem, but it also means monitoring software

is working.
- A response of `"working"`. This is the ideal response.

The status forms a collection of 3-tuples: `("service", "status", response time)`. The service is a name, like `"primary database"` or `"router"` or any of numerous other services that can be part of a distributed web application. The status value is a string value of either `"working"` or `"not working"`. The response time is the number of milliseconds it took to respond. Typical numbers are 10-50.

The summary is one of the following values:

- `Stopped`: There is one service that is not responding.
- `Degraded`: There is one service that has responded with a time that is out of the healthy time window of 50 milliseconds or less. Or, there is one service that has responded with `"not working"`.
- `Running`: All services are working and responding within the 50 millisecond window.

Two of the possible implementations are as follows:

- Write four filter functions. Apply the filters to the sequence of status values and count how many match each filter. Based on the number of matches, decide which of the three responses to provide for the overall health of the system.
- Write a mapping to apply a severity number: 2 for an indication of `Stopped`, 1 for either of the indications of `Degraded`, or 0 for all other service status messages. The maximum value of this vector is the overall health of the system.

Implement all variations. Compare the resulting code for clarity and expressiveness.

Classification of state, Part II

In the previous exercise, services were described as reporting a status value as a string value of either `"working"` or `"not working"`.

Before proceeding, either complete the previous exercise, or develop a workable design to solve the previous exercise.

Due to technology upgrades, the status values for some services include a third value: `"degraded"`. This has the same implication as a slow response from a service. This may change the design. It will certainly change the implementation.

Provide an implementation that gracefully handles the idea of additional or distinct status messages. The idea is to isolate the status-message checking to a function that can be replaced easily. For example, we might start with three functions to evaluate status values: `is_stopped()`, `is_degraded()`, and `is_working()`. When a change is required, we can write a new version, `is_degraded_2()`, that can be used in place of the old `is_degraded()` function.

The objective is to create an application that does not require a change to the implementation of any particular function. Instead, new functions are added; these new functions will reuse existing functions plus new functions to complete the expanded objectives.

Optimizing a file parser

In *Flattening data while mapping*, we used the following expression to extract a sequence of numbers from text with space separators:

```
(
    v
    for line in text.splitlines()
        for v in line.split()
)
```

The definition of the `split()` method includes the `\n` character, which is also used by the `splitlines()` method. It seems like this can be optimized to use only the `split()` method.

After getting this to work, change the source text in the example to:

```
>>> text = """2,3,5,7,11,13,17,19,23,29
... 31,37,41,43,47,53,59,61,67,71
... 73,79,83,89,97,101,103,107,109,113
... 127,131,137,139,149,151,157,163,167,173
```

```
... 179,181,191,193,197,199,211,223,227,229
... """
```

We can parse this with a single use of the `split()` method. This requires restructuring a single, long sequence of values into various rows and columns.

Is this faster than the use of the `splitlines()` and `split()` methods?

6

Recursions and Reductions

Many functional programming language compilers will optimize a recursive function to transform a recursive call in the tail of the function to an iteration. This tail-call optimization will dramatically improve performance. Python doesn't do this automatic tail-call optimization. One consequence is pure recursion suffers from limitations. Lacking an automated optimization, we need to do the tail-call optimization manually. This means rewriting recursion to use an explicit iteration. There are two common ways to do this, and we'll consider them both in this chapter.

In previous chapters, we've looked at several related kinds of processing design patterns; some of them are as follows:

- Mapping and filtering, which create collections from collections
- Reductions that create a scalar value from a collection

The distinction is exemplified by functions such as `map()` and `filter()` that accomplish the first kind of collection processing. There are some more specialized reduction functions, which include `min()`, `max()`, `len()`, and `sum()`. There's a general-purpose reduction

function as well, `functools.reduce()`.

We'll also consider creating a `collections.Counter()` object as a kind of reduction operator. It doesn't produce a single scalar value *per se*, but it does create a new organization of the data that eliminates some of the original structure. At heart, it's a kind of count-group-by operation that has more in common with a counting reduction than with a mapping.

In this chapter, we'll look at reduction functions in more detail. From a purely functional perspective, a reduction can be defined recursively. The tail-call optimization techniques available in Python apply elegantly to reductions.

We'll review a number of built-in reduction algorithms including `sum()`, `count()`, `max()`, and `min()`. We'll look at the `collections.Counter()` creation and related `itertools.groupby()` reductions. We'll also look at how parsing (and lexical scanning) are proper reductions since they transform sequences of tokens (or sequences of characters) into higher-order collections with more complex properties.

Simple numerical recursions

We can consider all numeric operations to be defined by recursions. For more details, read about the **Peano axioms** that define the essential features of numbers at `https://www.britannica.com/science/Peano-axioms`.

From these axioms, we can see that addition is defined recursively using more primitive notions of the next number, or the successor of a number n, $S(n)$.

To simplify the presentation, we'll assume that we can define a predecessor function, $P(n)$, such that $n = S(P(n)) = P(S(n))$, as long as $n \neq 0$. This formalizes the idea that a number is the successor of the number's predecessor.

Addition between two natural numbers could be defined recursively as follows:

$$\text{add}(a, b) = \begin{cases} b & \textbf{if } a = 0 \\ \text{add}(P(a), S(b)) & \textbf{if } a \neq 0 \end{cases}$$

If we use the more typical notations of $n + 1$ and $n - 1$ instead of $S(n)$ and $P(n)$, we can more easily see how the rule $\text{add}(a, b) = \text{add}(a - 1, b + 1)$ when $a \neq 0$ works.

This translates neatly into Python, as shown in the following function definition:

```
def add(a: int, b: int) -> int:
    if a == 0:
        return b
    else:
        return add(a - 1, b + 1)
```

We've rearranged the abstract mathematical notation into concrete Python.

There's no good reason to provide our own functions in Python to do simple addition. We rely on Python's underlying implementation to properly handle arithmetic of various kinds. Our point here is that fundamental scalar arithmetic can be defined recursively, and the definition translates to Python.

This suggests that more complicated operations, defined recursively, can also be translated to Python. The translation can be manually optimized to create working code that matches the abstract definitions, reducing questions about possible bugs in the implementation.

A recursive definition must include at least two cases: a non-recursive (or *base*) case where the value of the function is defined directly, and the recursive case where the value of the function is computed from a recursive evaluation of the function with different argument values.

In order to be sure the recursion will terminate, it's important to see how the recursive case computes values that approach the defined non-recursive base case. Pragmatically, there are often constraints on the argument values that we've omitted from the functions here. For example, the `add()` function in the preceding command snippet could be expanded to include `assert a>=0 and b>=0` to establish two necessary constraints on the input values.

Without these constraints, starting with `a` equal to -1 won't approach the non-recursive case of `a == 0` as we keep subtracting 1 from `a`.

Implementing manual tail-call optimization

For some functions, the recursive definition is the most succinct and expressive. A common example is the `factorial()` function.

We can see how this is rewritten as a simple recursive function in Python from the following formula:

$$n! = \begin{cases} 1 & \textbf{if } n = 0 \\ n \times (n-1)! & \textbf{if } n \neq 0 \end{cases}$$

The preceding formula can be implemented in Python by using the following function definition:

```
def fact(n: int) -> int:
    if n == 0:
        return 1
    else:
        return n*fact(n-1)
```

This implementation has the advantage of simplicity. The recursion limits in Python artificially constrain us; we can't do anything larger than about `fact(997)`. The value of 1000! has 2,568 digits and generally exceeds our floating-point capacity; on some systems the floating-point limit is near 10^{300}. Pragmatically, it's common to switch to a log gamma function instead of working with immense numbers.

See `https://functions.wolfram.com/GammaBetaErf/LogGamma/introductions/Gammas/ShowAll.html` for more on log gamma functions.

We can expand Python's call stack limit to stretch this to the limits of memory. It's better, however, to manually optimize these kinds of functions to eliminate the recursion.

This function demonstrates a typical tail recursion. The last expression in the function is a call to the function with a new argument value. An optimizing compiler can replace the function call stack management with a loop that executes very quickly.

In this example, the function involves an incremental change from n to $n - 1$. This means that we're generating a sequence of numbers and then doing a reduction to compute their product.

Stepping outside purely functional processing, we can define an imperative `facti()` calculation as follows:

```
def facti(n: int) -> int:
    if n == 0:
        return 1
    f = 1
    for i in range(2, n+1):
        f = f * i
    return f
```

This version of the factorial function will compute values beyond 1000! (2000!, for example, has 5,736 digits). This example isn't purely functional. We've optimized the tail recursion into a stateful `for` statement depending on the `i` variable to maintain the state of the computation.

In general, we're obliged to do this in Python because Python can't automatically do the tail-call optimization. There are situations, however, where this kind of optimization isn't actually helpful. We'll look at a few of them.

Leaving recursion in place

In some cases, the recursive definition is actually optimal. Some recursions involve a *divide and conquer* strategy that minimizes the work. One example of this is the algorithm for doing exponentiation by squaring. This works for computing values that have a positive integer exponent, like 2^{64}. We can state it formally as follows:

$$a^n = \begin{cases} 1 & \textbf{if } n = 0 \\ a \times a^{(n-1)} & \textbf{if } a \textbf{ is odd} \\ (a^{\frac{n}{2}})^2 & \textbf{if } a \textbf{ is even} \end{cases}$$

We've broken the process into three cases, easily written in Python as a recursion. Look at the following function definition:

```
def fastexp(a: float, n: int) -> float:
    if n == 0:
        return 1
    elif n % 2 == 1:
        return a * fastexp(a, n - 1)
    else:
        t = fastexp(a, n // 2)
        return t * t
```

For odd numbers, the `fastexp()` method is defined recursively. The exponent `n` is reduced by 1. A simple tail-recursion optimization would work for this case. It would not work for the even case, however.

For even numbers, the `fastexp()` recursion uses `n // 2`, chopping the problem into half of its original size. Since the problem size is reduced by a factor of 2, this case results in a significant speed-up of the processing.

We can't trivially reframe this kind of function into a tail-call optimization loop. Since it's already optimal, we don't really need to optimize it further. The recursion limit in Python would impose the constraint of $n \le 2^{1000}$, a generous upper bound.

Handling difficult tail-call optimization

We can look at the definition of **Fibonacci** numbers recursively. The following is one widely used definition for the n^{th} Fibonacci number, F_n:

$$F_n = \begin{cases} 0 & \textbf{if } n = 0 \\ 1 & \textbf{if } n = 1 \\ F_{n-1} + F_{n-2} & \textbf{if } n \ge 2 \end{cases}$$

A given Fibonacci number, F_n, is defined as the sum of the previous two numbers, $F_{n-1} + F_{n-2}$. This is an example of multiple recursion: it can't be trivially optimized as a simple tail recursion. However, if we don't optimize it to a tail recursion, we'll find it to be too slow to be useful.

The following is a naïve implementation:

```
def fib(n: int) -> int:
    if n == 0: return 0
    if n == 1: return 1
    return fib(n-1) + fib(n-2)
```

This suffers from a terrible multiple recursion problem. When computing the `fib(n)` value, we must compute the `fib(n-1)` and `fib(n-2)` values. The computation of the `fib(n-1)` value involves a duplicate calculation of the `fib(n-2)` value. The two recursive uses of the `fib()` function will more than duplicate the amount of computation being done.

Because of the left-to-right Python evaluation rules, we can evaluate values up to about `fib(1000)`. However, we have to be patient. Very patient. (Trying to find the actual upper bound with the default stack size means waiting a long time before the `RecursionError` is raised.)

The following is one alternative, which restates the entire algorithm to use stateful variables instead of a simple recursion:

```
def fibi(n: int) -> int:
    if n == 0: return 0
    if n == 1: return 1
    f_n2, f_n1 = 1, 1
    for _ in range(2, n):
        f_n2, f_n1 = f_n1, f_n2 + f_n1
    return f_n1
```

Our stateful version of this function counts up from 0, unlike the recursion, which counts down from the initial value of n. This version is considerably faster than the recursive version.

What's important here is that we couldn't trivially optimize the `fib()` function recursion with an obvious rewrite. In order to replace the recursion with an imperative version, we had to look closely at the algorithm to determine how many stateful intermediate variables were required.

As an exercise for the reader, try using the `@cache` decorator from the `functools` module. What impact does this have?

Processing collections through recursion

When working with a collection, we can also define the processing recursively. We can, for example, define the `map()` function recursively. The formalism could be stated as follows:

$$\text{map}(f, C) = \begin{cases} [] & \textbf{if } \text{len}(C) = 0 \\ \text{map}(f, C_{[:-1]}) + [f(C_{-1})] & \textbf{if } \text{len}(C) > 0 \end{cases}$$

We've defined the mapping of a function, f, to an empty collection as an empty sequence, `[]`. We've also specified that applying a function to a collection can be defined recursively with a three-step expression. First, recursively perform the mapping of the function to all of the collection except the last element, creating a sequence object. Then apply the function to the last element. Finally, append the last calculation to the previously built sequence.

Following is a purely recursive function version of this `map()` function:

```
from collections.abc import Callable, Sequence
from typing import Any, TypeVar

MapD = TypeVar("MapD")
```

```
MapR = TypeVar("MapR")

def mapr(
        f: Callable[[MapD], MapR],
        collection: Sequence[MapD]
) -> list[MapR]:
    if len(collection) == 0: return []
    return mapr(f, collection[:-1]) + [f(collection[-1])]
```

The value of the `mapr(f,[])` method is defined to be an empty list object. The value of the `mapr()` function with a non-empty list will apply the function to the last element in the list and append this to the list built recursively from the `mapr()` function applied to the head of the list.

We have to emphasize that this `mapr()` function actually creates a list object. The built-in `map()` function is an iterator; it doesn't create a list object. It yields the result values as they are computed. Also, the work is done in right-to-left order, which is not the way Python normally works. This is only observable when using a function that has side effects, something we'd like to avoid doing.

While this is an elegant formalism, it still lacks the tail-call optimization required. An optimization will allow us to exceed the default recursion limit of 1,000 and also performs much more quickly than this naïve recursion.

The use of `Callable[[Any], Any]` is a weak type hint. To be more clear, it can help to define a domain type variable and a range type variable. We'll include this detail in the optimized example.

Tail-call optimization for collections

We have two general ways to handle collections: we can use a higher-order function that returns a generator expression, or we can create a function that uses a `for` statement to process each item in a collection. These two patterns are very similar.

Following is a higher-order function that behaves like the built-in `map()` function:

```
from collections.abc import Callable, Iterable, Iterator
from typing import Any, TypeVar

DomT = TypeVar("DomT")
RngT = TypeVar("RngT")

def mapf(
        f: Callable[[DomT], RngT],
        C: Iterable[DomT]
) -> Iterator[RngT]:
    return (f(x) for x in C)
```

We've returned a generator expression that produces the required mapping. This uses the explicit `for` in the generator expression as a kind of tail-call optimization.

The source of data, `C`, has a type hint of `Iterable[DomT]` to emphasize that some type, `DomT`, will form the domain for the mapping. The transformation function has a hint of `Callable[[DomT], RngT]` to make it clear that it transforms from some domain type to a range type. The function `float()`, for example, can transform values from the string domain to the float range. The result has the hint of `Iterator[RngT]` to show that it iterates over the range type, `RngT`; the result type of the callable function.

The following is a generator function with the same signature and result:

```
def mapg(
        f: Callable[[DomT], RngT],
        C: Iterable[DomT]
) -> Iterator[RngT]:
    for x in C:
        yield f(x)
```

This uses a complete `for` statement for the tail-call optimization. The results are identical. This version is slightly slower because it involves multiple statements.

In both cases, the result is an iterator over the results. We must do something else to

materialize a sequence object from an iterable source. For example, here is the `list()` function being used to create a sequence from the iterator:

```
>>> list(mapg(lambda x: 2 ** x, [0, 1, 2, 3, 4]))
[1, 2, 4, 8, 16]
```

For performance and scalability, this kind of tail-call optimization is required in Python programs. It makes the code less than purely functional. However, the benefit far outweighs the lack of purity. In order to reap the benefits of succinct and expressive functional design, it is helpful to treat these less-than-pure functions as if they were proper recursions.

What this means, pragmatically, is that we must avoid cluttering up a collection processing function with additional stateful processing. The central tenets of functional programming are still valid even if some elements of our programs are less than purely functional.

Using the assignment (sometimes called the "walrus") operator in recursions

In some cases, recursions involve conditional processing that can be optimized using the "walrus" or assignment operator, `:=`. The use of assignment means that we're introducing stateful variables. If we're careful of the scope of those variables, the possibility of terribly complex algorithms is reduced.

We reviewed the `fast_exp()` function shown below in the *Leaving recursion in place* section. This function used three separate cases to implement a *divide and conquer* strategy. In the case of raising a number, `a`, to an even power, we can use $t = a^{\frac{n}{2}}$ to compute $t \times t = a^n$:

```
def fastexp_w(a: float, n: int) -> float:
    if n == 0:
        return 1
    else:
        q, r = divmod(n, 2)
        if r == 1:
            return a * fastexp_w(a, n - 1)
```

```
        else:
            return (t := fastexp_w(a, q)) * t
```

This uses the `:=` walrus operator to compute a partial answer, `fastexp_w(a, q)`, and save it into a temporary variable, `t`. This is used later in the same statement to compute `t * t`.

For the most part, when we perform tail-call optimization on a recursion, the body of the `for` statement will have ordinary assignment statements. It isn't often necessary to exploit the walrus operator.

The assignment operator is often used in situations like regular expression matching, where we want to save the match object as well as make a decision. It's very common to see `if (match := pattern.match(text)):` as a way to both attempt a regular expression match, save the resulting match object, and confirm it's not a `None` object.

Reductions and folding a collection from many items to one

We can consider the `sum()` function to have the following kind of definition. We could say that the sum of a collection is 0 for an empty collection. For a non-empty collection, the sum is the first element plus the sum of the remaining elements:

$$\text{sum}([c_0, c_1, c_2, ..., c_n]) = \begin{cases} 0 & \textbf{if } n = 0 \\ c_0 + \text{sum}([c_1, c_2, ..., c_n]) & \textbf{if } n > 0 \end{cases}$$

We can use a slightly simplified notation called the Bird-Meertens Formalism. This uses $\oplus/[c_0, c_1, ...c_n]$ to show how some arbitrary binary operator, $\oplus$, can be applied to a sequence of values. It's used as follows to summarize a recursive definition into something a little easier to work with:

$$\text{sum}([c_0, c_1, c_2, ..., c_n]) = +/[c_0, c_1, c_2, ..., c_n] = 0 + c_0 + c_1 + ... + c_n$$

We've effectively folded the + operator between each item of the sequence. Implicitly, the processing will be done left to right. This could be called a "fold left" way of reducing a collection to a single value. We could also imagine grouping the operators from right to left, calling this a "fold right." While some compiled languages will perform this optimization, Python works strictly from left to right when given a sequence of similar precedence operators.

In Python, a product function can be defined recursively as follows:

```
from collections.abc import Sequence

def prodrc(collection: Sequence[float]) -> float:
    if len(collection) == 0: return 1
    return collection[0] * prodrc(collection[1:])
```

This is a tiny rewrite from a mathematical notation to Python. However, it is less than optimal because all of the slices will create a large number of intermediate list objects. It's also limited to only working with explicit collections; it can't work easily with iterable objects.

We can revise this slightly to work with an iterable, which avoids creating any intermediate collection objects. The following is a properly recursive product function that works with any iterator as a source of data:

```
from collections.abc import Iterator

def prodri(items: Iterator[float]) -> float:
    try:
        head = next(items)
    except StopIteration:
        return 1
    return head * prodri(items)
```

This doesn't work with iterable collections. We can't interrogate an iterator with the `len()`

function to see how many elements it has. All we can do is attempt to extract the head of the iterator. If there are no items in the iterator, then any attempt to get the head will raise the `StopIteration` exception. If there is an item, then we can multiply this item by the product of the remaining items in the sequence.

Note that we must explicitly create an iterator from a materialized sequence object, using the `iter()` function. In other contexts, we might have an iterable result that we can use. Following is an example:

```
>>> prodri(iter([1,2,3,4,5,6,7]))
5040
```

This recursive definition does not rely on explicit state or other imperative features of Python. While it's more purely functional, it is still limited to working with collections of under 1,000 items. (While we can extend the stack size, it's far better to optimize this properly.) Pragmatically, we can use the following kind of imperative structure for reduction functions:

```
from collections.abc import Iterable

def prodi(items: Iterable[float]) -> float:
    p: float = 1
    for n in items:
        p *= n
    return p
```

This avoids any recursion limits. It includes the required tail-call optimization. Furthermore, this will work equally well with any iterable. This means a `Sequence` object, or an iterator.

Tail-call optimization using deques

The heart of recursion is a stack of function calls. Evaluating `fact(5)`, for example, is `5*fact(4)`. The value of `fact(4)` is `5*fact(3)`. There is a stack of pending computations until `fact(0)` has a value of 1. Then the stack of computations is completed, revealing the

final result.

Python manages the stack of calls for us. It imposes an arbitrary default limit of 1,000 calls on the stack, to prevent a program with a bug in the recursion from running forever.

We can manage the stack manually, also. This gives us another way to optimize recursions. We can—explicitly—create a stack of pending work. We can then do a final summarization of the pending work, emptying the items from the stack.

For something as simple as computing a factorial value, the stacking and unstacking can seem like needless overhead. For more complex applications, like examining the hierarchical file system, it seems more appropriate to mix processing files with putting directories onto a stack for later consideration.

We need a function to traverse a directory hierarchy without an explicit recursion. The core concept is that a directory is a collection of entries, and each entry is either a file, a sub-directory, or some other filesystem object we don't want to touch (e.g., a mount point, symbolic link, etc.).

We can say a node in the directory tree is a collection of entries: $N = e_0, e_1, e_2, ..., e_n$. Each entry is either another directory, $e \in \mathbb{D}$, or a file, $e \in \mathbb{F}$.

We can perform mappings on each file in the tree to process each file's content. We might perform a filter operation to create an iterator over files with a specific property. We can also perform a reduction to count the number of files with a property. In this example, we'll count the occurrences of a specific substring throughout the contents of files in a directory tree.

Formally, we want a function $p(f)$ that will provide the count of `"print"` in a node of the directory tree. It could be defined like this:

$$p(N) = \begin{cases} |\text{“print”} \in N| & \textbf{if } N \in \mathbb{F} \\ \sum_{e \in N} p(e) & \textbf{if } N \in \mathbb{D} \end{cases}$$

This shows how to apply the $p(N)$ function to each element of a directory tree. When the element is a file, $e \in \mathbb{F}$, we can count instances of "print". When the element is a directory, $e \in \mathbb{D}$, we need to apply the $p(N)$ function recursively to each entry, e_x, in the directory. While directory trees can't be deep enough to break Python's stack size limit, this kind of algorithm reveals an alternative tail-call optimization. It is an opportunity to use an explicit stack.

The `collections.deque` class is a marvelous way to build stacks and queues. The name comes from "double-ended queue," sometimes spelled dequeue. The data structure can be used as either a **last-in-first-out** (**LIFO**) stack or a **first-in-first-out** (**FIFO**). In this example, we use the `append()` and `pop()` methods, which enforce LIFO stack behavior. While this is much like a list, there are some optimizations in the `deque` implementation that can make it slightly faster than the generic list.

Using a stack data structure lets us work with a hierarchy of indefinite size without running into Python's internal stack depth limitation and raising `RecursionError` exceptions. The following function will traverse a file hierarchy looking at Python source files (with a suffix of `.py`):

```
from collections import deque
from pathlib import Path

def all_print(start: Path) -> int:
    count = 0
    pending: deque[Path] = deque([start])
    while pending:
        dir_path = pending.pop()
        for path in dir_path.iterdir():
            if path.is_file():
                if path.suffix == '.py':
                    count += path.read_text().count("print")
            elif path.is_dir():
                if not path.stem.startswith('.'):
                    pending.append(path)
```

```
            else:  # Ignore other filesystem objects
                pass
    return count
```

We seeded the stack of pending tasks with the initial directory. The essential algorithm is to unstack a directory and visit each entry in the directory. For entries that are files with the proper suffix, the processing is performed: counting the occurrences of "print". For entries that are directories, the directory is put into the stack as a pending task. Note that directories with a leading dot in their name need to be ignored. For the code in this book, those directories include caches used by tools like **mypy**, **pytest**, and **tox**. We want to skip over those cache directories.

The processing performed on each file is part of the `all_print()` function. This can be refactored as a separate function that's applied to each node as part of a reduction. Rewriting the `all_print()` function to be a proper higher-order function is left as an exercise for the reader.

The idea here is we have two strategies for transforming a formal recursion into a usefully optimized function. We can reframe the recursion into an iteration, or we can introduce an explicit stack.

In the next section, we will apply the idea of a reduction (and the associated tail-call optimizations) to creating groups of items and computing a reduction for the groups.

Group-by reduction from many items to fewer

The idea of a reduction can apply in many ways. We've looked at the essential recursive definition of a reduction that produces a single value from a collection of values. This leads us to optimizing the recursion so we have the ability to compute summaries without the overheads of a naive Pythonic implementation.

Creating subgroups in Python isn't difficult, but it can help to understand the formalisms that support it. This understanding can help to avoid implementations that perform

extremely poorly.

A very common operation is a reduction that groups values by some key or indicator. The raw data is grouped by some column's value, and reductions (sometimes called *aggregate functions*) are applied to other columns.

In SQL, this is often called the `GROUP BY` clause of the `SELECT` statement. The SQL aggregate functions include `SUM`, `COUNT`, `MAX`, and `MIN`, and often many more.

Python offers us several ways to group data before computing a reduction of the grouped values. We'll start by looking at two ways to get simple counts of grouped data. Then we'll look at ways to compute different summaries of grouped data.

We'll use the trip data that we computed in *Chapter 4, Working with Collections*. This data started as a sequence of latitude-longitude waypoints. We restructured it to create legs represented by three-tuples of start, end, and distance for each leg. The data looks as follows:

```
(((37.5490162, -76.330295), (37.840832, -76.273834), 17.7246),
((37.840832, -76.273834), (38.331501, -76.459503), 30.7382),
((38.331501, -76.459503), (38.845501, -76.537331), 31.0756),
...
((38.330166, -76.458504), (38.976334, -76.473503), 38.8019))
```

We'd like to know the most common distance. Since the data is real-valued, and continuous, each distance is a unique value. We need to constrain these values from the continuous domain to a discrete set of distances. For example, quantizing each leg to the nearest multiple of five nautical miles. This creates bands of 0 to 5 miles, over 5 to 10 miles, etc. Once we've created discrete integer values, we can count the number of legs in each of these bands.

These quantized distances can be produced with a generator expression:

```
quantized = (5 * (dist // 5) for start, stop, dist in trip)
```

This will divide each distance by 5—discarding any fractions—then multiply the truncated result by 5 to compute a number that represents the distance rounded down to the nearest 5 nautical miles.

We don't use the values assigned to the `start` and `stop` variables. It's common practice to assign these values to the _ variable. This can lead to some confusion because this can obscure the structure of the triple. It would look like this:

```
quantized = (5 * (dist // 5) for _, _, dist in trip)
```

This approach can be helpful for removing some visual clutter.

Building a mapping with Counter

A mapping like the `collections.Counter` class is a great optimization for doing reductions that create counts (or totals) grouped by some value in the collection. The following expression creates a mapping from distance to frequency:

```
# See Chapter 4 for ways to parse "file:./Winter%202012-2013.kml"
# We want to build a trip variable with the sequence of tuples

>>> from collections import Counter

>>> quantized = (5 * (dist // 5) for start, stop, dist in trip)
>>> summary = Counter(quantized)
```

The resulting `summary` object is stateful; it can be updated. The expression to create the groups, `Counter()`, looks like a function, making it a good fit for a design based on functional programming ideas.

If we print the `summary.most_common()` value, we'll see the following results:

```
>>> summary.most_common()
[(30.0, 15), (15.0, 9), ...]
```

The most common distance was about 30 nautical miles. We can also apply functions like `min()` and `max()` to find the shortest recorded and longest legs as well.

Note that your output may vary slightly from what's shown. The results of the `most_common()` function are in order of frequency; equal-frequency bins may be in any order. These five lengths may not always be in the order shown:

```
(35.0, 5), (5.0, 5), (10.0, 5), (20.0, 5), (25.0, 5)
```

This slight variability makes testing with the **doctest** tool a little bit more complex. One helpful trick for testing with counters is to use a dictionary to validate the results in general; the comparison between actual and expected no longer relies on the vagaries of internal hash computations.

Building a mapping by sorting

An alternative to `Counter` is to sort the original collection, and then use a recursive loop to identify when each group begins. This involves materializing the raw data, performing a sort that could—at worst—do $\mathbf{O}(n \log n)$ operations, and then doing a reduction to get the sums or counts for each key.

In order to work in a general way with Python objects that can be sorted, we need to define the protocol required for sorting. We'll call the protocol `SupportsRichComparisonT` because we can sort any kinds of objects that implement the rich comparison operators, `<` and `>`. This isn't a particular class of objects; it's a protocol that any number of classes might implement. We formalize the idea of a protocol that classes must support using the `typing.Protocol` type definition. It could be also be called an interface that a class must implement. Python's flexibility stems from having a fairly large number of protocols that many different classes support.

The following is a common algorithm for creating groups from sorted data:

```
from collections.abc import Iterable
from typing import Any, TypeVar, Protocol, TypeAlias

class Comparable(Protocol):
    def __lt__(self, __other: Any) -> bool: ...
    def __gt__(self, __other: Any) -> bool: ...
SupportsRichComparisonT = TypeVar("SupportsRichComparisonT",
bound=Comparable)

Leg: TypeAlias = tuple[Any, Any, float]

def group_sort(trip: Iterable[Leg]) -> dict[int, int]:

    def group(
            data: Iterable[SupportsRichComparisonT]
    ) -> Iterable[tuple[SupportsRichComparisonT, int]]:
        sorted_data = iter(sorted(data))
        previous, count = next(sorted_data), 1
        for d in sorted_data:
            if d == previous:
                count += 1
            else:
                yield previous, count
                previous, count = d, 1
        yield previous, count

    quantized = (int(5 * (dist // 5)) for beg, end, dist in trip)
    return dict(group(quantized))
```

The internal `group()` function steps through the sorted sequence of legs. If a given item key has already been seen—it matches the value in `previous`—then the `counter` variable is incremented. If a given item does not match the previous value, then there's been a change in value: emit the previous value and the count, and begin a new accumulation of counts for the new value.

The definition of `group()` provides two important type hints. The source data is an iterable over some type, shown with the type variable `SupportsRichComparisonT`. In this specific

case, it's pretty clear that the values in use will be of type `int`; however, the algorithm will work for any Python type. The resulting iterable from the `group()` function will preserve the type of the source data, and this is made explicit by using the same type variable, `SupportsRichComparisonT`.

The final line of the `group_sort()` function creates a dictionary from the grouped items. This dictionary will be similar to a `Counter` dictionary. The primary difference is that a `Counter()` function will have a `most_common()` method function, which a default dictionary lacks.

We can also do this with `itertools.groupby()`. We'll look at this function closely in *Chapter 8, The Itertools Module*.

Grouping or partitioning data by key values

There are no limits to the kinds of reductions we might want to apply to grouped data. We might have data with a number of independent and dependent variables. We can consider partitioning the data by an independent variable and computing summaries such as the maximum, minimum, average, and standard deviation of the values in each partition.

The essential trick to doing more sophisticated reductions is to collect all of the data values into each group. The `Counter()` function merely collects counts of identical items. For deeper analysis, we want to create sequences of the original members of the group.

Looking back at our trip data, each five-mile bin could contain the entire collection of legs of that distance, not merely a count of the legs. We can consider the partitioning as a recursion or as a stateful application of `defaultdict(list)` objects. We'll look at the recursive definition of a `groupby()` function, since it's easy to design.

Clearly, the `groupby(C, key)` computation for an empty collection, `[]`, is the empty dictionary, `dict()`. Or, more usefully, the empty `defaultdict(list)` object.

For a non-empty collection, we need to work with item `C[0]`, the head, and recursively process sequence `C[1:]`, the tail. We can use slice expressions, or we can use the `head, *tail = C` statement to do this parsing of the collection, as follows:

```
>>> C = [1,2,3,4,5]
>>> head, *tail = C
>>> head
1
>>> tail
[2, 3, 4, 5]
```

If we have a `defaultdict` object named `groups`, we need to use the expression `groups[key(head)].append(head)` to include the head element in the `groups` dictionary. After this, we need to evaluate the `groupby(tail, key)` expression to process the remaining elements.

We can create a function as follows:

```
from collections import defaultdict
from collections.abc import Callable, Sequence, Hashable
from typing import TypeVar

SeqItemT = TypeVar("SeqItemT")
ItemKeyT = TypeVar("ItemKeyT", bound=Hashable)

def group_by(
        key: Callable[[SeqItemT], ItemKeyT],
        data: Sequence[SeqItemT]
) -> dict[ItemKeyT, list[SeqItemT]]:

    def group_into(
            key: Callable[[SeqItemT], ItemKeyT],
            collection: Sequence[SeqItemT],
            group_dict: dict[ItemKeyT, list[SeqItemT]]
    ) -> dict[ItemKeyT, list[SeqItemT]]:
        if len(collection) == 0:
            return group_dict
        head, *tail = collection
        group_dict[key(head)].append(head)
        return group_into(key, tail, group_dict)
```

```
    return group_into(key, data, defaultdict(list))
```

The interior function `group_into()` handles the essential recursive definition. An empty value for `collection` returns the provided dictionary, `group_dict`. A non-empty collection is partitioned into a head and tail. The head is used to update the `group_dict` dictionary. The tail is then used, recursively, to update the dictionary with all remaining elements.

The type hints make an explicit distinction between the type of the source objects `SeqItemT` and the type of the key `ItemKeyT`. The function provided as the `key` parameter must be a callable that returns a value of the key type `ItemKeyT`, given an object of the source type `SeqItemT`. In many of the examples, a function to extract the distance from a `Leg` object will be be shown. This is a `Callable[[SeqItemT], ItemKeyT]` where the source type `SeqItemT` is the `Leg` object and the key type `ItemKeyT` is the float value.

`bound=Hashable` is an additional constraint. This defines an "upper bound" on the possible types, alerting **mypy** that any type that could be assigned to this type variable must implement the protocol for `Hashable`. The essential, immutable Python types of numbers, strings, and tuples all meet this bound. A mutable object like a dictionary, set, or list, will not meet the upper bound, leading to warnings from **mypy**.

We can't easily use Python's default values to collapse this into a single function. We explicitly cannot use the following incorrect command snippet:

```
# Bad use of a mutable default value

def group_by(key, data, dictionary=defaultdict(list)):
```

If we try this, all uses of the `group_by()` function share one common `defaultdict(list)` object. This does not work because Python builds the default value just once. Mutable objects as default values rarely do what we want. The common practice is to provide a `None` value, and use an explicit `if` statement to create each unique, empty instance of

`defaultdict(list)` as needed. We've shown how to use a wrapper function definition to avoid the `if` statement.

We can group the data by distance as follows:

```
>>> binned_distance = lambda leg: 5 * (leg[2] // 5)
>>> by_distance = group_by(binned_distance, trip)
```

We've defined a reusable lambda that puts our distances into bins, each of which is 5 nautical miles in size. We then grouped the data using the provided lambda.

We can examine the binned data as follows:

```
>>> import pprint
>>> for distance in sorted(by_distance):
...     print(distance)
...     pprint.pprint(by_distance[distance])
```

The following is what the output looks like:

```
0.0
[((35.505665, -76.653664), (35.508335, -76.654999), 0.1731),
 ((35.028175, -76.682495), (35.031334, -76.682663), 0.1898),
 ((25.4095, -77.910164), (25.425833, -77.832664), 4.3155),
 ((25.0765, -77.308167), (25.080334, -77.334), 1.4235)]
5.0
[((38.845501, -76.537331), (38.992832, -76.451332), 9.7151),
 ((34.972332, -76.585167), (35.028175, -76.682495), 5.8441),
 ((30.717167, -81.552498), (30.766333, -81.471832), 5.103),
 ((25.471333, -78.408165), (25.504833, -78.232834), 9.7128),
 ((23.9555, -76.31633), (24.099667, -76.401833), 9.844)]
...
125.0
[((27.154167, -80.195663), (29.195168, -81.002998), 129.7748)]
```

Having looked at a recursive definition, we can turn to looking at making a tail-call

optimization to build a group-by algorithm using iteration. This will work with larger collections of data, because it can exceed the internal stack size limitation.

We'll start with doing tail-call optimization on the `group_into()` function. We'll rename this to `partition()` because partitioning is another way of looking at grouping.

The `partition()` function can be written as an iteration as follows:

```
from collections import defaultdict
from collections.abc import Callable, Hashable, Iterable
from typing import TypeVar

SeqT = TypeVar("SeqT")
KeyT = TypeVar("KeyT", bound=Hashable)

def partition(
        key: Callable[[SeqT], KeyT],
        data: Iterable[SeqT]
) -> dict[KeyT, list[SeqT]]:
    group_dict = defaultdict(list)
    for head in data:
        group_dict[key(head)].append(head)
        #---------------------------------
    return group_dict
```

When doing the tail-call optimization, the essential line of the code in the imperative version will match the recursive definition. We've put a comment under the changed line to emphasize the rewrite is intended to have the same outcome. The rest of the structure represents the tail-call optimization we've adopted as a common way to work around the Python limitations.

The type hints emphasize the distinction between the source type `SeqT` and the key type `KeyT`. The source data can be anything, but the keys are limited to types that have proper hash values.

Writing more general group-by reductions

Once we have partitioned the raw data, we can compute various kinds of reductions on the data elements in each partition. We might, for example, want the northernmost point for the start of each leg in the distance bins.

We'll introduce some helper functions to decompose the tuple as follows:

```
# Legs are (start, end, distance) tuples

start = lambda s, e, d: s

end = lambda s, e, d: e

dist = lambda s, e, d: d

# start and end of a Leg are (lat, lon) tuples

latitude = lambda lat, lon: lat

longitude = lambda lat, lon: lon
```

Each of these helper functions expects a tuple object to be provided using the `*` operator to map each element of the tuple to a separate parameter of the lambda. Once the tuple is expanded into the `s`, `e`, and `p` parameters, it's reasonably obvious to return the proper parameter by name. It's much clearer than trying to interpret the `tuple_arg[2]` value.

The following is how we use these helper functions:

```
>>> point = ((35.505665, -76.653664), (35.508335, -76.654999),
0.1731)
>>> start(*point)
(35.505665, -76.653664)

>>> end(*point)
(35.508335, -76.654999)
```

```
>>> dist(*point)
0.1731

>>> latitude(*start(*point))
35.505665
```

Our initial `point` object is a nested three tuple with (0)—a starting position, (1)—the ending position, and (2)—the distance. We extracted various fields using our helper functions.

Given these helpers, we can locate the northernmost starting position for the legs in each bin:

```
>>> binned_distance = lambda leg: 5 * (leg[2] // 5)
>>> by_distance = partition(binned_distance, trip)
>>> for distance in sorted(by_distance):
...     print(
...         distance,
...         max(by_distance[distance],
...         key=lambda pt: latitude(*start(*pt)))
...     )
```

The data that we grouped by distance included each leg of the given distance. We supplied all of the legs in each bin to the `max()` function. The `key` function we provided to the `max()` function extracted just the latitude of the starting point of the leg.

This gives us a short list of the northernmost legs of each distance, as follows:

```
0.0 ((35.505665, -76.653664), (35.508335, -76.654999), 0.1731)
5.0 ((38.845501, -76.537331), (38.992832, -76.451332), 9.7151)
10.0 ((36.444168, -76.3265), (36.297501, -76.217834), 10.2537)
...
125.0 ((27.154167, -80.195663), (29.195168, -81.002998), 129.7748)
```

Writing higher-order reductions

We'll look at an example of a higher-order reduction algorithm here. This will introduce a rather complex topic. The simplest kind of reduction develops a single value from a collection of values. Python has a number of built-in reductions, including `any()`, `all()`, `max()`, `min()`, `sum()`, and `len()`.

As we noted in *Chapter 4, Working with Collections*, we can do a great deal of statistical calculation if we start with a few reductions such as the following:

```
from collections.abc import Sequence

def sum_x0(data: Sequence[float]) -> float:
    return sum(1 for x in data)  # or len(data)

def sum_x1(data: Sequence[float]) -> float:
    return sum(x for x in data)  # or sum(data)

def sum_x2(data: Sequence[float]) -> float:
    return sum(x*x for x in data)
```

This allows us to define mean, standard deviation, normalized values, correction, and even least-squares linear regression, building on these base reduction functions.

The last of our reductions, `sum_x2()`, shows how we can apply existing reductions to create higher-order functions. We might change our approach to be more like the following:

```
from collections.abc import Callable, Iterable
from typing import Any

def sum_f(
        function: Callable[[Any], float],
        data: Iterable[float]
) -> float:
    return sum(function(x) for x in data)
```

We've added a function, `function()`, as a parameter; the function can transform the data.

This overall function, `sum_f()`, computes the sum of the transformed values.

Now we can apply this function in three different ways to compute the three essential sums as follows:

```
>>> data = [7.46, 6.77, 12.74, 7.11, 7.81,
...     8.84, 6.08, 5.39, 8.15, 6.42, 5.73]

>>> N = sum_f(lambda x: 1, data)  # x**0
>>> N
11
>>> S = sum_f(lambda x: x, data)  # x**1
>>> round(S, 2)
82.5
>>> S2 = sum_f(lambda x: x*x, data)  # x**2
>>> round(S2, 4)
659.9762
```

We've plugged in a small lambda to compute $\sum_{x \in X} x^0 = \sum_{x \in X} 1$, which is the count, $\sum_{x \in X} x^1 = \sum_{x \in X} x$, the sum, and $\sum_{x \in X} x^2$, the sum of the squares, which we can use to compute standard deviation.

A common extension to this includes a filter to reject raw data that is unknown or unsuitable in some way. We might use the following function to reject bad data:

```
from collections.abc import Callable, Iterable

def sum_filter_f(
        filter_f: Callable[[float], bool],
        function: Callable[[float], float],
        data: Iterable[float]
) -> float:
    return sum(function(x) for x in data if filter_f(x))
```

The following function definition for computing a mean will reject `None` values in a simple way:

```
valid = lambda x: x is not None

def mean_f(predicate: Callable[[Any], bool], data: Sequence[float])
-> float:
    count_ = lambda x: 1
    sum_ = lambda x: x
    N = sum_filter_f(valid, count_, data)
    S = sum_filter_f(valid, sum_, data)
    return S / N
```

This shows how we can provide two distinct combinations of lambdas to our `sum_filter_f()` function. The filter argument is a lambda that rejects `None` values; we've called it `valid` to emphasize its meaning. The function argument is a lambda that implements a count or a sum operation. We can easily add a lambda to compute a sum of squares.

The reuse of a common `valid` rule assures that the various computations are all identical in applying any filters to the source data. This can be combined with a user-selected filter criteria to provide a tidy plug-in to compute a number of statistics related to a user's requested subset of the data.

Writing file parsers

We can often consider a file parser to be a kind of reduction. Many languages have two levels of definition: the lower-level tokens in the language and the higher-level structures built from those tokens. When looking at an XML file, the tags, tag names, and attribute names form this lower-level syntax; the structures which are described by XML form a higher-level syntax.

The lower-level lexical scanning is a kind of reduction that takes individual characters and groups them into tokens. This fits well with Python's generator function design pattern. We can often write functions that look as follows:

```
from collections.abc import Iterator
from enum import Enum
import re

class Token(Enum):
    SPACE = 1
    PARA = 2
    EOF = 3

def lexical_scan(some_source: str) -> Iterator[tuple[Token, str]]:
    previous_end = 0
    separator_pat = re.compile(r"\n\s*\n", re.M|re.S)
    for sep in separator_pat.finditer(some_source):
        start, end = sep.span()
        yield Token.PARA, some_source[previous_end: start]
        yield Token.SPACE, some_source[start: end]
        previous_end = end
    yield Token.PARA, some_source[previous_end:]
    yield Token.EOF, ""
```

For well-known file formats, we'll use existing file parsers. For data in CSV, JSON, XML, or TOML format, we don't need to write file parsers. Most of these modules have a `load()` method that produces useful Python objects.

In some cases, we'll need to combine the results of this parsing into higher-level objects, useful for our specific application. While the CSV parser provides individual rows, these might need to be used to create `NamedTuple` instances, or perhaps some other class of immutable Python objects. Our examples of trip data, starting in *Chapter 4, Working with Collections*, are combined into higher-level objects, legs of a journey, by an algorithm that combines waypoints into pairs. When we introduce more complex decision-making, we make a transition from restructuring into parsing.

In order to provide useful waypoints in the first place, we needed to parse a source file. In these examples, the input was a KML file; KML is an XML representation of geographic information. The essential features of the parser look similar to the following definition:

```
from collections.abc import Iterator
from typing import TextIO, cast

def comma_split(text: str) -> list[str]:
    return text.split(",")

def row_iter_kml(file_obj: TextIO) -> Iterator[list[str]]:
    ns_map = {
        "ns0": "http://www.opengis.net/kml/2.2",
        "ns1": "http://www.google.com/kml/ext/2.2"}
    xpath = (
        "./ns0:Document/ns0:Folder/"
        "ns0:Placemark/ns0:Point/ns0:coordinates")
    doc = XML.parse(file_obj)
    return (
        comma_split(cast(str, coordinates.text))
        for coordinates in doc.findall(xpath, ns_map)
    )
```

The bulk of the `row_iter_kml()` function is the XML parsing that allows us to use the `doc.findall()` function to iterate through the `<ns0:coordinates>` tags in the document. We've used a function named `comma_split()` to parse the text of this tag into a three-tuple of values.

The `cast()` function is only present to provide evidence to **mypy** that the value of `coordinates.text` is a `str` object. The default definition of the text attribute is `Union[str, bytes]`; in this application, the data will be `str` exclusively. The `cast()` function doesn't do any runtime processing.

This function focused on working with the normalized XML structure. The document is close to the database designer's definitions of **first normal form**: each attribute is atomic (a single value), and each row in the XML data has the same columns with data of a consistent type. The data values aren't fully atomic, however: we have to split the points on the , to separate longitude, latitude, and altitude into atomic string values. However, the text value for these XML tags is internally consistent, making it a close fit with first

normal form.

A large volume of data—XML tags, attributes, and other punctuation—is reduced to a somewhat smaller volume, including just floating-point latitude and longitude values. For this reason, we can think of parsers as a kind of reduction.

We'll need a higher-level set of conversions to map the tuples of text into floating-point numbers. Also, we'd like to discard altitude, and reorder longitude and latitude. This will produce the application-specific tuple we need. We can use functions as follows for this conversion:

```
from collections.abc import Iterator

def pick_lat_lon(
        lon: str, lat: str, alt: str
) -> tuple[str, str]:
    return lat, lon

def float_lat_lon(
        row_iter: Iterator[list[str]]
) -> Iterator[tuple[float, float]]:
    lat_lon_iter = (
        pick_lat_lon(*row)
        for row in row_iter
    )
    return (
        (float(lat), float(lon))
        for lat, lon in lat_lon_iter
    )
```

The essential tool is the `float_lat_lon()` function. This is a higher-order function that returns a generator expression. The generator uses the `map()` function to apply the `float()` function conversion to the results of the `pick_lat_lon()` function, and the `*row` argument to assign each member of the row tuple to a different parameter of the `pick_lat_lon()` function. This only works when each row is a three-tuple. The `pick_lat_lon()` function then returns a two-tuple of the selected items in the required order.

The source includes XML that looks like this:

```
<Placemark><Point>
<coordinates>-76.33029518659048, 37.54901619777347,0</coordinates>
</Point></Placemark>
```

We can use this parser as follows:

```
>>> import urllib.request
>>> source_url = "file:./Winter%202012-2013.kml"
>>> with urllib.request.urlopen(source_url) as source:
...     flat = list(float_lat_lon(row_iter_kml(source)))
```

This will build a tuple-of-tuples representation of each waypoint along the path in the original KML file. The result will be a flat sequence of pairs that looks like this:

```
>>> from pprint import pprint
>>> pprint(flat)  # doctest: +ELLIPSIS
[(37.54901619777347, -76.33029518659048),
 ...
 (38.976334, -76.473503)]
```

The `float_lat_lon()` function uses a low-level XML parser to extract rows of text data from the original representation. It uses a higher-level parser to transform the text items into more useful tuples of floating-point values suitable for the target application.

Parsing CSV files

In *Chapter 3, Functions, Iterators, and Generators*, we saw another example where we parsed a CSV file that was not in a normalized form: we had to discard header rows to make it useful. To do this, we used a function that extracted the header and returned an iterator over the remaining rows.

The data looks as follows:

```
Anscombe's quartet
I II III IV
x y x y x y x y
10.0 8.04 10.0 9.14 10.0 7.46 8.0 6.58
8.0 6.95 8.0 8.14 8.0 6.77 8.0 5.76
...
5.0 5.68 5.0 4.74 5.0 5.73 8.0 6.89
```

The columns are separated by tab characters. Plus, there are three rows of headers that we can discard.

Here's another version of that CSV-based parser. We've broken it into three functions. The first, `row_iter_csv()` function, returns the iterator over the rows in a tab-delimited file. The function looks as follows:

```
from collections.abc import Iterator
import csv
from typing import TextIO

def row_iter_csv(source: TextIO) -> Iterator[list[str]]:
    rdr = csv.reader(source, delimiter="\t")
    return rdr
```

This is a small wrapper around the CSV parsing process. When we look back at the previous parsers for XML and plain text, this was the kind of thing that was missing from those parsers. Producing an iterable over row tuples can be a common feature of parsers for normalized data.

Once we have a row of tuples, we can pass rows that contain usable data and reject rows that contain other metadata, such as titles and column names. We'll introduce a helper function that we can use to do some of the parsing, plus a `filter()` function to validate a row of data.

Following is the conversion:

```
from typing import cast

def float_none(data: str) -> float | None:
    try:
        data_f = float(data)
        return data_f
    except ValueError:
        return None
```

This function handles the conversion of a single string to float values, converting bad data to a `None` value. The type hint of `float | None` expresses the idea of having a value of the given type or having a value of the same type as `None`. This can also be stated as `Union[float, None]` to show how the result is a union of different alternative types.

We can embed the `float_none()` function in a mapping so that we convert all columns of a row to a float or `None` value. A lambda for this looks as follows:

```
from collections.abc import Callable
from typing import TypeAlias

R_Float: TypeAlias = list[float | None]

float_row: Callable[[list[str]], R_Float] = \
    lambda row: list(map(float_none, row))
```

Two type hints are used to make the definition of the `float_row()` function explicit. The `R_Float` hint defines the floating-point version of a row of data that may include `None` values.

Following is a row-level validator based on the use of the `all()` function to ensure that all values are `float` (or none of the values are `None`):

```
all_numeric: Callable[[R_Float], bool] = \
    lambda row: all(row) and len(row) == 8
```

This lambda is a kind of reduction, transforming a row of floating-point values to a Boolean value if all values are not "falsy" (that is, neither `None` nor zero) and there are exactly eight values.

The simplistic `all_numeric()` function conflates zero and `None`. A more sophisticated test would rely on something such as `not any(item is None for item in row)`. The rewrite is left as an exercise for the reader.

The essential design is to create row-based elements that can be combined to create more complete algorithms for parsing an input file. The foundational functions iterate over tuples of text. These are combined to convert and validate the converted data. For the cases where files are either in first normal form (all rows are the same) or a simple validator can reject the extraneous rows, this design pattern works out nicely.

All parsing problems aren't quite this simple, however. Some files have important data in header or trailer rows that must be preserved, even though it doesn't match the format of the rest of the file. These non-normalized files will require a more sophisticated parser design.

Parsing plain text files with headers

In *Chapter 3, Functions, Iterators, and Generators*, the `Crayola.GPL` file was presented without showing the parser. This file looks as follows:

```
GIMP Palette
Name: Crayola
Columns: 16
#
239 222 205 Almond
205 149 117 Antique Brass
```

We can parse a text file using regular expressions. We need to use a filter to read (and parse) header rows. We also want to return an iterable sequence of data rows. This rather complex two-part parsing is based entirely on the two-part—head and tail—file structure.

Following is a low-level parser that handles both the four lines of the header and the long tail:

```
from collections.abc import Iterator
from typing import TextIO, TypeAlias

Head_Body: TypeAlias = tuple[tuple[str, str], Iterator[list[str]]]

def row_iter_gpl(file_obj: TextIO) -> Head_Body:
    header_pat = re.compile(
        r"GIMP Palette\nName:\s*(.*?)\nColumns:\s*(.*?)\n#\n",
        re.M)

    def read_head(file_obj: TextIO) -> tuple[tuple[str, str],
    TextIO]:
        if match := header_pat.match(
            "".join(file_obj.readline() for _ in range(4))
        ):
            return (match.group(1), match.group(2)), file_obj
        else:
            raise ValueError("invalid header")

    def read_tail(
            headers: tuple[str, str],
            file_obj: TextIO) -> Head_Body:
        return (
            headers,
            (next_line.split() for next_line in file_obj)
        )

    return read_tail(*read_head(file_obj))
```

The `Head_Body` type definition summarizes the overall goal of the row iterator. The result is a two-tuple. The first item is a two-tuple with details from the file header. The second item is an iterator that provides the text items for a color definition. This `Head_Body` type hint is used in two places in this function definition.

The `header_pat` regular expression parses all four lines of the header. There are instances

of `()` in the expression to extract the name and column information from the header.

There are two internal functions for parsing different parts of the file. The `read_head()` function parses the header lines and returns interesting text and a `TextIO` object that can be used for the rest of the parsing. It does this by reading four lines and merging them into a single long string. This is then parsed with the `header_pat` regular expression.

The idea of returning the iterator from one function to be used in another function is a pattern for passing an explicitly stateful object from one function to another. It seems helpful to make sure all of the arguments for the `read_tail()` function are the results from the `read_head()` function.

The `read_tail()` function parses the iterator over the remaining lines. These lines are merely split on spaces, since that fits the description of the GPL file format.

For more information, visit the following link: `https://code.google.com/p/grafx2/issues/detail?id=518`.

Once we've transformed each line of the file into a canonical tuple-of-strings format, we can apply the higher level of parsing to this data. This involves conversion and (if necessary) validation.

The following is a higher-level parser command snippet:

```
from collections.abc import Iterator
from typing import NamedTuple

class Color(NamedTuple):
    red: int
    blue: int
    green: int
    name: str

def color_palette(
        headers: tuple[str, str],
```

```
        row_iter: Iterator[list[str]]
) -> tuple[str, str, tuple[Color, ...]]:
    name, columns = headers
    colors = tuple(
        Color(int(r), int(g), int(b), " ".join(name))
        for r, g, b, *name in row_iter
    )
    return name, columns, colors
```

This function will work with the output of the lower-level `row_iter_gpl()` parser: it requires the headers and the iterator over individual rows. This function will use the multiple assignment feature of the `for` clause in the generator to separate the color numbers and the remaining words into four variables, `r`, `g`, `b`, and `name`. The use of the `*name` parameter ensures that all remaining values will be assigned to the `name` variable as a tuple. The `" ".join(name)` expression then concatenates the words into a single space-separated string.

The following is how we can use this two-tier parser:

```
>>> from pathlib import Path
>>> source_path = Path("crayola.gpl")
>>> with source_path.open() as source:
...     name, cols, colors = color_palette(
...         *row_iter_gpl(source)
...     )
>>> name
'Crayola'
>>> cols
'16'
>>> len(colors)
133
```

We've applied the higher-level parser to the results of the lower-level parser. This will return the headers and a tuple built from the sequence of `Color` objects.

Summary

In this chapter, we've looked at two significant functional programming topics. We've looked at recursions in some detail. Many functional programming language compilers will optimize a recursive function to transform a call in the tail of the function to a loop. This is sometimes called tail recursion elimination. More commonly, it's known as tail-call optimization. In Python, we must do the tail-call optimization manually by using an explicit `for` statement, replacing a purely functional recursion.

We've also looked at reduction algorithms, including `sum()`, `count()`, `max()`, and `min()` functions. We looked at the `collections.Counter()` function and related `groupby()` reductions.

We've also looked at how parsing (and lexical scanning) are similar to reductions since they transform sequences of tokens (or sequences of characters) into higher-order collections with more complex properties. We've examined a design pattern that decomposes parsing into a lower level and tries to produce tuples of raw strings, and a higher level that creates more useful application objects.

In the next chapter, we'll look at some techniques appropriate to working with named tuples and other immutable data structures. We'll look at techniques that make stateful objects unnecessary. While stateful objects aren't purely functional, the idea of a class hierarchy can be used to package related method definitions.

Exercises

This chapter's exercises are based on code available from Packt Publishing on GitHub. See `https://github.com/PacktPublishing/Functional-Python-Programming-3rd-Edition`.

In some cases, the reader will notice that the code provided on GitHub includes partial solutions to some of the exercises. These serve as hints, allowing the reader to explore alternative solutions.

In many cases, exercises will need unit test cases to confirm they actually solve the problem.

These are often identical to the unit test cases already provided in the GitHub repository. The reader should replace the book's example function name with their own solution to confirm that it works.

Multiple recursion and caching

In *Handling difficult tail-call optimization*, we looked at a naive definition of a function to compute Fibonacci numbers, the `fib()` function. The `functools.cache` decorator can have a profound impact on the performance of this algorithm.

Implement both versions and describe the impact of caching on the time required to compute large Fibonacci numbers.

Refactor the all_print() function

In *Tail-call optimization using deques*, we showed a function that used a `collections.deque` to visit all nodes in a directory tree, summing the value for each node that is a proper file. This can be done with a list as well as a `deque`, with some minor code changes.

This function embedded a specific computation. This computation (finding all occurrences of "print") really should have been a separate function. The body of the `all_print()` function should be refactored into two functions:

- A generic directory traverse that applies a function to each text file with the expected suffix and sums the results.
- A function that counts instances of "print" in a given Python file.

Parsing CSV files

See the *Parsing CSV files* section, earlier in this chapter. In that example, the simplistic `all_numeric()` function conflates zero and `None`.

Create a test case for this function that will show that it does not handle zero correctly, treating it as `None`. Once the test case is defined, rewrite the `all_numeric()` function to distinguish between zero and `None`.

Note that it's common practice in Python to use the `is` operator when comparing with

`None`. This specifically avoids some subtle problems that can arise when a class has an implementation of `__eq__()` that doesn't handle `None` as a properly distinct object.

Classification of state, Part III

See *Chapter 5*, *Higher-Order Functions*, the *Classification of state* exercise.

There's a third way to consume status details and summarize them.

Write a reduce computation. This starts with an initial state of **Running**. As each service's three-tuple is folded into the result, there is a comparison between the state and the three-tuple. If the three-tuple has a non-responsive service, the state advances to **Stopped**. If the three-tuple has a slow or not working service, the state advances to **Degraded**. If no problems are found, the initial value becomes the final health of the overall system.

The idea is to provide a `status_add(previous, this_service)` function. This can be used in the context of `status = reduce(status_add, service_status_sequence, "Running")` to compute the current status of the sequence of services.

Diesel engine data

A diesel engine has had some repairs that raised doubts about the accuracy of the tachometer. After some heroic effort, the following table of data was collected showing the observed reading on the engine's tachometer, and the actual RPMs measured with an optical device on the engine.

Sample	Tach	Engine
1	1000	883
2	1500	1242
3	1500	1217
4	1600	1306
5	1750	1534
6	2000	1805
7	2000	1720

If needed, create a CSV file with the data. If you have access to the GitHub repository for this book, this is available in the `engine.csv` file.

Create a `NamedTuple` for each sample and write some functions to acquire this data in a useful form. Once the data is available, see the *Using sums and counts for statistics* section of *Chapter 4*, *Working with Collections*, for a definition of a correlation function.

The objective is to apply this correlation function to the engine and tach values to see if the values correlate. If they do, it suggests that the engine's instruments can be recalibrated. If they don't correlate, something else is wrong with the engine.

Note that the *Chapter 4*, *Working with Collections*, correlation example may have assumptions about data types that don't necessarily apply to the `NamedTuple` defined earlier. If necessary, rewrite the type hints or your `NamedTuple` definition. Note that it can be difficult to write perfectly generic type hints, and it often takes a bit of work to resolve the differences.

7

Complex Stateless Objects

Many of the examples we've looked at have either been functions using atomic (or scalar) objects, or relatively simple structures built from small tuples. We can often exploit Python's immutable `typing.NamedTuple` as a way to build complex data structures. The class-like syntax seems much easier to read than the older `collections.namedtuple` syntax.

One of the beneficial features of object-oriented programming is the ability to create complex data structures incrementally. In some respects, an object can be viewed as a cache for results of functions; this will often fit well with functional design patterns. In other cases, the object paradigm provides for property methods that include sophisticated calculations to derive data from an object's properties. Using properties of an otherwise immutable class is also a good fit for functional design ideas.

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

- How we create and use `NamedTuple` and frozen `@dataclass` definitions.
- Ways that immutable `NamedTuple` or frozen `@dataclass` objects can be used instead of stateful object classes.

- How to use the popular third-party `pyrsistent` package instead of stateful object classes. This is not part of the standard library, and requires a separate install.
- Some techniques to write generic functions outside any polymorphic class definition. While we can rely on callable classes to create a polymorphic class hierarchy, in some cases, this might be a needless overhead in a functional design. This will touch on using the `match` statement for identifying types or structures.

While frozen dataclasses and `NamedTuple` subclasses are nearly equivalent, a frozen dataclass omits the sequence features that a `NamedTuple` includes. Iterating over the members of a `NamedTuple` object is a confusing feature; a dataclass doesn't suffer from this potential problem.

We'll start our journey by looking at using `NamedTuple` subclasses.

Using tuples to collect data

In *Chapter 3*, *Functions, Iterators, and Generators*, we showed two common techniques to work with tuples. We've also hinted at a third way to handle complex structures. We can go with any of the following techniques, depending on the circumstances:

- Use lambdas (or functions created with the `def` statement) to select a named item based on the index
- Use lambdas (or `def` functions) with multiple positional parameters coupled with `*args` to assign a tuple of items to parameter names
- Use a `NamedTuple` class to select an item by attribute name or index

Our trip data, introduced in *Chapter 4*, *Working with Collections*, has a rather complex structure. The data started as an ordinary time series of position reports. To compute the distances covered, we transposed the data into a sequence of legs with a start position, end position, and distance as a nested three-tuple.

Each item in the sequence of legs looks as follows as a three-tuple:

```
>>> some_leg = (
...     (37.549016, -76.330295),
...     (37.840832, -76.273834),
...     17.7246
... )
```

The first two items are the starting and ending points. The third item is the distance between the points. This is a short trip between two points on the Chesapeake Bay.

A nested tuple of tuples can be rather difficult to read; for example, expressions such as `some_leg[0][0]` aren't very informative.

Let's look at the three alternatives for selecting values out of a tuple. The first technique involves defining some simple selection functions that can pick items from a tuple by index position:

```
>>> start = lambda leg: leg[0]
>>> end = lambda leg: leg[1]
>>> distance = lambda leg: leg[2]

>>> latitude = lambda pt: pt[0]
>>> longitude = lambda pt: pt[1]
```

With these definitions, we can use `latitude(start(some_leg))` to refer to a specific piece of data. It looks like this code example:

```
>>> latitude(start(some_leg))
37.549016
```

It's awkward to provide type hints for lambdas. The following shows how this becomes complex-looking:

```
from collections.abc import Callable
from typing import TypeAlias

Point: TypeAlias = tuple[float, float]
Leg: TypeAlias = tuple[Point, Point, float]

start: Callable[[Leg], Point] = lambda leg: leg[0]
```

The type hint must be provided as part of the assignment statement. This tells **mypy** that the object named `start` is a callable function that accepts a single parameter of a type named `Leg` and returns a result of the `Point` type. A function created with the `def` statement will usually have an easier-to-read type hint.

A variation on this first technique for collecting complex data uses the `*parameter` notation to conceal some details of the index positions. The following are some selection functions that are evaluated with the `*` notation:

```
>>> start_s = lambda start, end, distance: start
>>> end_s = lambda start, end, distance: end
>>> distance_s = lambda start, end, distance: distance

>>> latitude_s = lambda lat, lon: lat
>>> longitude_s = lambda lat, lon: lon
```

With these definitions, we can extract specific pieces of data from a tuple. We've used the `_s` suffix to emphasize the need to use star, `*`, when evaluating these lambdas. It looks like this code example:

```
>>> longitude_s(*start_s(*some_leg))
-76.330295
```

This has the advantage of a little more clarity in the function definitions. The association between position and name is given by the list of parameter names. It can look a little odd to see the `*` operator in front of the tuple arguments to these selection functions. This

operator is useful because it maps each item in a tuple to a parameter of the function.

While these are very functional, the syntax for selecting individual attributes can be confusing. Python offers two object-oriented alternatives, `NamedTuple` and the frozen `@dataclass`.

Using NamedTuple to collect data

The second technique for collecting data into a complex structure is `typing.NamedTuple`. The idea is to create a class that is an immutable tuple with named attributes. There are two variations available:

- The `namedtuple` function in the `collections` module.
- The `NamedTuple` base class in the `typing` module. We'll use this almost exclusively because it allows explicit type hinting.

In the following examples, we'll use nested `NamedTuple` classes such as the following:

```
from typing import NamedTuple

class PointNT(NamedTuple):
    latitude: float
    longitude: float

class LegNT(NamedTuple):
    start: PointNT
    end: PointNT
    distance: float
```

This changes the data structure from simple anonymous tuples to named tuples with type hints provided for each attribute. Here's an example:

```
>>> first_leg = LegNT(
...     PointNT(29.050501, -80.651169),
...     PointNT(27.186001, -80.139503),
...     115.1751)
```

```
>>> first_leg.start.latitude
29.050501
```

The `first_leg` object was built as the `LegNT` subclass of the `NamedTuple` class. This object contains two other named tuple objects and a float value. Using `first_leg.start.latitude` will fetch a particular piece of data from inside the tuple structure. The change from prefix function names to postfix attribute names can be seen as a helpful emphasis. It can also be seen as a confusing shift in the syntax.

The `NT` suffix in the names is **not** a recommended practice.

We've included the suffix in the book to emphatically distinguish among similar-looking solutions to the problem of defining a useful class.

In actual applications, we'd choose one definition, and use the simplest, clearest names possible, avoiding needless suffixes that clutter up textbooks like this one.

Replacing simple `tuple()` functions with appropriate `LegNT()` or `PointNT()` function calls is important. This changes the processing that builds the data structure. It provides an explicitly named structure with type hints that can be checked by the **mypy** tool.

For example, take a look at the following code snippet to create point pairs from source data:

```
from collections.abc import Iterable, Iterator
from Chapter04.ch04_ex1 import pick_lat_lon

def float_lat_lon_tuple(
        row_iter: Iterable[list[str]]
) -> Iterator[tuple[float, float]]:
    lat_lon_iter = (pick_lat_lon(*row) for row in row_iter)
    return (
        (float(lat), float(lon))
```

```
        for lat, lon in lat_lon_iter
    )
```

This requires an iterable object whose individual items are a list of strings. A CSV reader, or KML reader, can do this. The `pick_lat_lon()` function picks two values from the row. The generator expression applies the `pick_lat_lon()` function to the data source. The final generator expression creates a somewhat more useful two-tuple from the two string values.

The preceding code would be changed to the following code snippet to create `Point` objects:

```
from Chapter04.ch04_ex1 import pick_lat_lon
from typing import Iterable, Iterator

def float_lat_lon(
    row_iter: Iterable[list[str]]
) -> Iterator[PointNT]:
                #------
    lat_lon_iter = (pick_lat_lon(*row) for row in row_iter)
    return (
        PointNT(float(lat), float(lon))
        #------
        for lat, lon in lat_lon_iter
    )
```

The `PointNT()` constructor has been injected into the code. The data type that is returned is revised to be `Iterator[PointNT]`. It's clear that this function builds `Point` objects instead of anonymous two-tuples of floating-point coordinates.

Similarly, we can introduce the following to build the complete trip of `LegNT` objects:

```
from collections.abc import Iterable, Iterator
from typing import cast, TextIO
import urllib.request
from Chapter04.ch04_ex1 import legs, haversine, row_iter_kml
```

```
source_url = "file:./Winter%202012-2013.kml"
def get_trip(url: str=source_url) -> list[LegNT]:
    with urllib.request.urlopen(url) as source:
        path_iter = float_lat_lon(row_iter_kml(source))
        pair_iter = legs(path_iter)
        trip_iter = (
            LegNT(start, end, round(haversine(start, end), 4))
            for start, end in pair_iter
        )
        trip = list(trip_iter)
    return trip
```

The processing is defined as a sequence of generator expressions, each one of which is lazy and operates on a single object. The `path_iter` object uses two generator functions, `row_iter_kml()` and `float_lat_lon()`, to read the rows from a KML file, pick fields, and convert them to `Point` objects. The `pair_iter()` object uses the `legs()` generator function to yield overlapping pairs of `Point` objects showing the start and end of each leg.

The `trip_iter` generator expression creates the final `LegNT` objects from pairs of `Point` objects. These generated objects are consumed by the `list()` function to create a single list of legs. The `haversine()` function from *Chapter 4, Working with Collections*, is used to compute the distance.

The rounding is applied in this function for two reasons. First, as a practical matter, 0.0001 nautical miles is about 20 cm (7 inches). Pragmatically, rounding to 0.001 nautical miles involves fewer digits that offer a false sense of precision. Second–and more important–it makes the unit testing more reliable across platforms if we avoid looking at all the digits of a floating-point number.

The final `trip` object is a sequence of `LegNT` instances. It will look as follows when we try to print it:

```
>>> source_url = "file:./Winter%202012-2013.kml"
>>> trip = get_trip(source_url)

>>> trip[0].start
PointNT(latitude=37.54901619777347, longitude=-76.33029518659048)
>>> trip[0].end
PointNT(latitude=37.840832, longitude=-76.273834)
>>> trip[0].distance
17.7246
```

It's important to note that the `haversine()` function was written to use simple tuples. We've reused this function with a `NamedTuple` class instance. As we carefully preserved the order of the arguments, this small change in representation from anonymous tuple to named tuple was handled gracefully by Python.

Since this is a class definition, we can easily add methods and properties. This ability to add features to a `NamedTuple` makes them particularly useful for computing derived values. We can, for example, more directly implement the distance computation as part of the `Point` class, as shown in the following code:

```
import math

class PointE(NamedTuple):
    latitude: float
    longitude: float

    def distance(self, other: "PointE", R: float = 360*60/math.tau)
    -> float:
        """Equirectangular, 'flat-earth' distance."""
        Δϕ = (
            math.radians(self.latitude) -
            math.radians(other.latitude)
        )
```

```
        Δλ = (
            math.radians(self.longitude) -
            math.radians(other.longitude)
        )
        mid_ϕ = (
            (math.radians(self.latitude) -
            math.radians(other.latitude))
            / 2
        )
        x = R * Δλ * math.cos(mid_ϕ)
        y = R * Δϕ
        return math.hypot(x, y)
```

Given this definition of the `PointE` class, we have encapsulated the functions for working with points and distances. This can be helpful because it gives the reader a single place to look for the relevant attributes and methods.

Within the body of the `PointE` class, we can't easily refer to the class. The class name doesn't exist within the body of the `class` statement. The **mypy** tool lets us use a string instead of a class name to resolve these rare cases when a class needs to refer to itself.

We can use this class as shown in the following example:

```
>>> start = PointE(latitude=38.330166, longitude=-76.458504)
>>> end = PointE(latitude=38.976334, longitude=-76.473503)

# Apply the distance() method of the start object...
>>> leg = LegNT(start, end, round(start.distance(end), 4))
>>> leg.start == start
True
>>> leg.end == end
True
>>> leg.distance
38.7805
```

In most cases, the `NamedTuple` class definition adds clarity. The use of `NamedTuple` will lead to a change from function-like prefix syntax to object-like suffix syntax.

Using frozen dataclasses to collect data

The third technique for collecting data into a complex structure is the frozen `@dataclass`. The idea is to create a class that is an immutable collection of named attributes.

Following the example from the previous section, we can have nested dataclasses such as the following:

```
from dataclasses import dataclass

@dataclass(frozen=True)
class PointDC:
    latitude: float
    longitude: float

@dataclass(frozen=True)
class LegDC:
    start: PointDC
    end: PointDC
    distance: float
```

We've used a decorator, `@dataclass(frozen=True)`, in front of the class definition to create an immutable (known as "frozen") dataclass. The decorator will add a number of functions for us, building a fairly sophisticated class definition without our having to provide anything other than the attributes. For more information on decorators, see *Chapter 12*, *Decorator Design Techniques*.

This also changes the data structure from simple anonymous tuples to a class definition with type hints provided for each attribute. Here's an example:

```
>>> first_leg = LegDC(
...     PointDC(29.050501, -80.651169),
...     PointDC(27.186001, -80.139503),
...     115.1751)
>>> first_leg.start.latitude
29.050501
```

The `first_leg` object was built as the `LegDC` instance. This object contains two other `PointDC` objects and a float value. Using `first_leg.start.latitude` will fetch a particular attribute of the object.

The `DC` suffix in the names is **not** a recommended practice.

We've included the suffix in the book to emphatically distinguish among similar-looking solutions to the problem of defining a useful class.

In actual applications, we'd choose one definition, and use the simplest, clearest names possible, avoiding needless suffixes that clutter up textbooks like this one.

Replacing a `()` tuple construction with appropriate `LegDC()` or `PointDC()` constructors builds a more sophisticated data structure than anonymous tuples. It provides an explicitly named structure with type hints that can be checked by the **mypy** tool.

Comparing frozen dataclasses with `NamedTuple` instances can lead to a "Which is better?" discussion. There are a few tradeoffs here. Most notably, a `NamedTuple` object is extremely simple: it ties up relatively little memory and offers few methods. A dataclass, on the other hand, can have a great deal of built-in functionality, and can tie up more memory. We can manage this using the `slots=True` argument with the `@dataclass` decorator, something we'll address later in this section.

Additionally, a `NamedTuple` object is a sequence of values. We can use an iterator over the tuple's attributes, a processing option that seems to create nothing but confusion. Iterating

over the values without using the names subverts the essential design concept of naming the members of the tuple.

A simple procedure for evaluating memory use is to create millions of instances of a class and see how much memory is allocated for the Python runtime. This works out best because Python object size involves a recursive walk through all of the associated objects, each of which has its own complex sizing computation. Generally, we only care about aggregate memory use for a large collection of objects, so it's more effective to measure that directly.

Here is one class definition to support a script designed to evaluate the size of 1,000,000 `NamedTuple` objects:

```
from typing import NamedTuple

class LargeNT(NamedTuple):
  a: str
  b: int
  c: float
  d: complex
```

We can then define a function to create a million objects, assigning them to a variable, `big_sequence`. The function can then report the amount of memory allocated by the Python runtime. This function will involve some odd-looking overheads. The documentation for the `getallocatedblocks()` function advises us to clear the type cache with the `sys._clear_type_cache()` function and force garbage collection via the `gc.collect()` function to clean up the objects that are no longer referenced. These two steps should compact memory to the smallest size, and provide more repeatable reports on the use of storage by this sequence of objects.

The following function creates a million instances of a given type and displays the allocated memory:

```
from typing import Type, Any

def sizing(obj_type: Type[Any]) -> None:
    big_sequence = [
        obj_type(f"Hello, {i}", 42*i, 3.1415926*i, i+2j)
        for i in range(1_000_000)
    ]
    sys._clear_type_cache()
    gc.collect()
    print(f"{obj_type.__name__} {sys.getallocatedblocks()}")
    del big_sequence
```

Evaluating this function with different class definitions will reveal how much storage is occupied by 1,000,000 objects of that class. We can use `sizing(LargeNT)` to see the space taken up by a `NamedTuple` class.

We'll need to define alternatives, of course. We can define a frozen dataclass. Additionally, we can use `@dataclass(frozen=True, slots=True)` to see what impact the use of `__slots__` has on the object sizing. The bodies of the classes must all have the same attributes in the same order to simplify construction of the objects by the `sizing()` function.

The actual results are highly implementation-specific, but the author's results on macOS Python 3.10.0 show the following:

Class Kind	Blocks Allocated
LargeNT	5,035,408
LargeDC	7,035,404
LargeDC_Slots	5,035,569
Baseline	35,425

This suggests that a `@dataclass` will use about 40% more memory than a `NamedTuple` or a `@dataclass` with `slots=True`.

This also suggests that a radically different design—one that uses iterators to avoid creating large in-memory collections—can use substantially less memory. What's important is to

have a correct solution in hand, and then explore alternative implementations to see which makes most effective use of the machine resources.

How to implement a complicated initialization is the most telling distinction among the various class definition approaches. We'll look at that next.

Complicated object initialization and property computations

When working with data in unhelpful formats, it often becomes necessary to build Python objects from source data that has a different structure or different underlying object types. There are two overall ways to treat object creation:

- It's part of the application as a whole. Data should be decomposed by a parser and recomposed into useful Python objects. This is the approach we've taken in previous examples.
- It's part of the object's class definition. Source data should be provided more or less in its raw form, and the class definition will perform the necessary conversions.

This distinction is never simple, nor crisp. Pragmatic considerations will identify the best approach for each unique case of building a Pythonic object from source data. The two examples that point to the distinct choices available are the following:

- The `Point` class: The syntax for geographic points is highly variable. A common approach is simple floating-point degree numbers. However, some sources provide degrees and minutes. Others might provide separate degrees, minutes, and seconds. Further, there are also Open Location Codes, which encode latitude and longitude. (See `https://maps.google.com/pluscodes/` for more information.) All of these various parsers should not be part of the class.
- The `LegNT` (or `LegDC`) class: The leg includes two points and a distance. The distance can be seeded as a simple value. It can also be computed as a property. A third choice is to use a sophisticated object builder. In effect, our `get_trip()` function (defined in *Using NamedTuple to collect data*) has implicitly included an object builder for `LegNT`

objects.

Using `LegNT(start, end, round(haversine(start, end), 4))` to create a `LegNT` instance isn't wrong, but it makes a number of assumptions that need to be challenged. Here are some of the assumptions:

- The application should always use `haversine()` to compute distances.
- The application should always pre-compute the distance. This is often an optimization question. Computing a distance once and saving it is helpful if every leg's distance will be examined. If distances are not always needed, it can be less expensive to compute the distance only when required.
- We always want to create `LegNT` instances. We've already seen cases where we might want a `@dataclass` implementation. In the next section, we'll look at a `pyrsistent.PRecord` implementation.

One general way to encapsulate the construction of `LegNT` instances is to use a `@classmethod` to handle complex initialization. Additionally, a `@dataclass` provides some additional initialization techniques.

A slightly better way to define a `NamedTuple` initialization is shown in the following example:

```
from typing import NamedTuple

class EagerLeg(NamedTuple):
    start: Point
    end: Point
    distance: float

    @classmethod
    def create(cls, start: Point, end: Point) -> "EagerLeg":
        return cls(
            start=start,
            end=end,
            distance=round(haversine(start, end), 4)
```

```
        )
```

Compare the above definition, which eagerly computes the distance, with the following, which lazily computes the distance:

```
from typing import NamedTuple

class LazyLeg(NamedTuple):
    start: Point
    end: Point

    @property
    def distance(self) -> float:
        return round(haversine(self.start, self.end), 4)

    @classmethod
    def create(cls, start: Point, end: Point) -> "LazyLeg":
        return cls(
            start=start,
            end=end
        )
```

Both of these class definitions have an identical `create()` method. We can use `EagerLeg.create(start, end)` or `LazyLeg.create(start, end)` without breaking anything else in the application.

What's most important is that the decision to compute values eagerly or lazily becomes a decision that we can alter at any time. We can replace these two definitions to see which has higher performance for our specific application's needs. The distance computation, similarly, is now part of this class, making it easier to define a subclass to make a change to the application.

A dataclass offers a somewhat more complex and flexible interface for object construction: a `__post_init__()` method. This method is evaluated after the object's fields have their values assigned, permitting eager calculation of derived values. This, however, can't work

for frozen dataclasses. The `__post_init__()` method can only be used for non-frozen dataclasses to eagerly compute additional values from the provided initialization values.

For dataclasses, as well as `NamedTuple` classes, a `@classmethod` creator is a good design pattern for doing initialization that involves eagerly computing attribute values.

As a final note on initialization, there are three different syntax forms for creating named tuple objects. Here are the three choices:

- We can provide the values positionally. This works well when the order of the parameters is obvious. It looks like this:

```
LegNT(start, end, round(haversine(start, end), 4))
```

- We can unpack a sequence using the `*` operator. This, too, requires the ordering of parameters be obvious. For example:

```
PointNT(*map(float, pick_lat_lon(*row)))
```

- We can use explicit keyword assignment. This has the advantage of making the parameter names clear and avoids hidden assumptions about ordering. Here's an example:

```
PointNT(longitude=float(row[0]), latitude=float(row[1]))
```

These examples show one way to package the initialization of complex objects. What's important is to avoid state change in these objects. A complex initialization is done exactly once, providing a single, focused place to understand how the object's state was established. For this reason, it's imperative for the initialization to be expressive of the object's purpose as well as flexible to permit change.

Using pyrsistent to collect data

In addition to Python's `NamedTuple` and `@dataclass` definitions, we can also use the `pyrsistent` module to create more complex object instances. The huge advantage of-

fered by the `pyrsistent` module is that the collections are immutable. Instead of updating in place, a change to a collection works through a general-purpose "evolution" object that creates a new immutable object with the changed value. In effect, what appears to be a state-changing method is actually an operator creating a new object.

The following example shows how to import the `pyrsistent` module and create a mapping structure with names and values:

```
>>> import pyrsistent
>>> v = pyrsistent.pmap({"hello": 42, "world": 3.14159})
>>> v  # doctest: +SKIP
pmap({'hello': 42, 'world': 3.14159})
>>> v['hello']
42
>>> v['world']
3.14159
```

We can't change the value of this object, but we can evolve an object `v` to a new object, `v2`. This object has the same starting values, but also includes a changed attribute value. It looks like this:

```
>>> v2 = v.set("another", 2.71828)
>>> v2    # doctest: +SKIP
pmap({'hello': 42, 'world': 3.14159, 'another': 2.71828})
```

The original object, `v`, is immutable, and its value hasn't changed:

```
>>> v    # doctest: +SKIP
pmap({'hello': 42, 'world': 3.14159})
```

It can help to think of this operation as having two parts. First, the original object is cloned. After the cloning, the changes are applied. In the above example, the `set()` method was used to provide a new key and value. We can create the evolution separately and apply it to an object to create a clone with changes applied. This seems ideal for an application

where an audit history of changes is required.

Note that we had to prevent using two examples as unit test cases. This is because the order of the keys isn't fixed. It's easy to check that the keys and values match our expectations, but a simplistic comparison with a dictionary literal doesn't *always* work.

The `PRecord` class is appropriate for defining complex objects. These objects are similar in some ways to a `NamedTuple`. We'll revisit our waypoint and leg data model using `PRecord` instances. The definitions are given in the following example:

```
from pyrsistent import PRecord, field

class PointPR(PRecord):  # type: ignore [type-arg]
    latitude = field(type=float)
    longitude = field(type=float)

class LegPR(PRecord):  # type: ignore [type-arg]
    start = field(type=PointPR)
    end = field(type=PointPR)
    distance = field(type=float)
```

Each field definition uses the sophisticated `field()` function to build the definition of the attribute. In addition to a sequence of types, this function can specify an invariant condition that must be true for the values, an initial value, whether or not the field is mandatory, a factory function that builds appropriate values, and a function to serialize the value into a string.

The PR suffix in the names is **not** a recommended practice.

We've included the suffix in the book to emphatically distinguish among similar-looking solutions to the problem of defining a useful class.

In actual applications, we'd choose one definition, and use the simplest, clearest names possible, avoiding needless suffixes that clutter up textbooks like this one.

As an extension to these definitions, we could convert the point value into a more useful format using a serializer function. This requires some formatting details because there's a slight difference in the way latitudes and longitudes are displayed. Latitudes include "N" or "S" and longitudes include "E" or "W":

```
from math import isclose, modf

def to_dm(format: dict[str, str], point: float) -> str:
    """Use {"+": "N", "-": "S"} for latitude; {"+": "E", "-": "W"}
    for longitude."""
    sign = "-" if point < 0 else "+"
    ms, d = modf(abs(point))
    ms = 60 * ms
    # Handle the 59.999 case:
    if isclose(ms, 60, rel_tol=1e-5):
        ms = 0.0
        d += 1
    return f"{d:3.0f}°{ms:.3f}'{format.get(sign, sign)}"
```

This function can be included as part of the field definition for the `PointPR` class; we must provide the function as the `serializer=` parameter of the `field()` factory:

```
from pyrsistent import PRecord, field

class PointPR_S(PRecord):  # type: ignore[type-arg]
    latitude = field(
        type=float,
        serializer=(
            lambda format, value:
                to_dm((format or {}) | {"+": "N", "-": "S"}, value)
        )
    )
    longitude = field(
        type=float,
        serializer=(
            lambda format, value:
```

```
                to_dm((format or {}) | {"+": "E", "-": "W"}, value)
            )
        )
```

This lets us print a point in a nicely formatted style:

```
>>> p = PointPR_S(latitude=32.842833333, longitude=-79.929166666)
>>> p.serialize()  # doctest: +SKIP
{'latitude': " 32°50.570'N", 'longitude': " 79°55.750'W"}
```

These definitions will provide nearly identical processing capabilities to the `NamedTuple` and `@dataclass` examples shown earlier. We can, however, leverage some additional features of the `pyrsistent` package to create `PVector` objects, which will be immutable sequences of waypoints in a trip. This requires a few small changes to previous applications.

The definition of the `get_trip()` function using `pyrsistent` can look like this:

```
from collections.abc import Iterable, Iterator
from typing import TextIO
import urllib.request
from Chapter04.ch04_ex1 import legs, haversine, row_iter_kml
from pyrsistent import pvector
from pyrsistent.typing import PVector

source_url = "file:./Winter%202012-2013.kml"
def get_trip_p(url: str=source_url) -> PVector[LegPR]:
    with urllib.request.urlopen(url) as source:
        path_iter = float_lat_lon(row_iter_kml(source))
        pair_iter = legs(path_iter)
        trip_iter = (
            LegPR(
                start=PointPR.create(start._asdict()),
                end=PointPR.create(end._asdict()),
                distance=round(haversine(start, end), 4))
            #-----------------------------------------
```

```
            for start, end in pair_iter
        )
        trip = pvector(trip_iter)
               #------
    return trip
```

The first change is relatively large. Instead of rewriting the `float_lat_lon()` function to return a `PointPR` object, we left this function alone. We used the `PRecord.create()` method to convert a dictionary into a `PointPR` instance. Given the two `PointPR` objects and the distance, we can create a `LegPR` object.

Earlier in this chapter, we showed a version of the `legs()` function that returned `typing.NamedTuple` instances with the raw data for each point along a leg. The `_asdict()` method of a `NamedTuple` will translate the tuple into a dictionary. The tuple's attribute names will be keys in the dictionary. The transformation can be seen in the following example:

```
>>> p = PointNT(2, 3)
>>> p._asdict()
{'latitude': 2, 'longitude': 3}
```

This can then be provided to the `PointPR.create()` method to create a proper `PointPR` instance that will be used in the rest of the application. The initial `PointNT` object can be discarded, having served as a bridge between input parsing and building more useful Python objects. In the long run, it's a good idea to revisit the underlying `legs()` function to rewrite it to work with `pyrsistent` record definitions.

Finally, instead of assembling a `list` from the iterator, we assembled a `PVector` instance, using the `pvector()` function. This has many of the same properties as the built-in list class, but is immutable. Any changes will create a clone of the object.

These high-performance, immutable collections are helpful ways to be sure an application behaves in a functional manner. The handy serialization into JSON-friendly notation

makes these class definitions ideal for applications that make use of JSON. Web servers, in particular, can benefit from using these class definitions.

Avoiding stateful classes by using families of tuples

In several previous examples, we've shown the idea of **wrap-unwrap** design patterns that allow us to work with anonymous and named tuples. The point of this kind of design is to use immutable objects that wrap other immutable objects instead of mutable instance variables.

A common statistical measure of correlation between two sets of data is the **Spearman's rank correlation**. This compares the rankings of two variables. Rather than trying to compare values, which might have different units of measure, we'll compare the relative orders. For more information, visit: `https://www.itl.nist.gov/div898/software/dataplot/refman2/auxillar/partraco.htm`.

Computing the Spearman's rank correlation requires assigning a rank value to each observation. It seems like we should be able to use `enumerate(sorted())` to do this. Given two sets of possibly correlated data, we can transform each set into a sequence of rank values and compute a measure of correlation.

We'll apply the wrap-unwrap design pattern to do this. We'll wrap data items with their rank for the purposes of computing the correlation coefficient.

In *Chapter 3, Functions, Iterators, and Generators*, we showed how to parse a simple dataset. We'll extract the four samples from that dataset as follows:

```
>>> from Chapter03.ch03_ex4 import (
... series, head_map_filter, row_iter)
>>> from pathlib import Path

>>> source_path = Path("Anscombe.txt")
>>> with source_path.open() as source:
```

```
...     data = list(head_map_filter(row_iter(source)))
```

The resulting collection of data has four different series of data combined in each row. A `series()` function was defined in *Generators for lists, dicts, and sets*, back in *Chapter 3, Functions, Iterators, and Generators*, to extract the pairs for a given series from the overall row.

The definition looked like this:

```
def series(
      n: int,
      row_iter: Iterable[list[SrcT]]
) -> Iterator[tuple[SrcT, SrcT]]:
```

The argument to this function is an iterable of some source type (usually a string). The result of this function is an iterable series of two-tuples from the source type. When working with CSV files, strings are the expectation. It's much nicer for the result to be a named tuple.

Here's a named tuple for each pair:

```
from typing import NamedTuple

class Pair(NamedTuple):
    x: float
    y: float
```

We'll introduce a transformation to convert anonymous tuples into named tuples or dataclasses:

```
from collections.abc import Callable, Iterable
from typing import TypeAlias
```

```
RawPairIter: TypeAlias = Iterable[tuple[float, float]]

pairs: Callable[[RawPairIter], list[Pair]] \
    = lambda source: list(Pair(*row) for row in source)
```

The `RawPairIter` type definition describes the intermediate output from the `series()` function. This function emits an iterable sequence of two-tuples. The `pairs` lambda object is a callable that expects an iterable and will produce a list of `Pair` named tuples or dataclass instances.

The following shows how the `pairs()` function and the `series()` function are used to create pairs from the original data:

```
>>> series_I = pairs(series(0, data))
>>> series_II = pairs(series(1, data))
>>> series_III = pairs(series(2, data))
>>> series_IV = pairs(series(3, data))
```

Each of these series is a list of `Pair` objects. Each `Pair` object has `x` and `y` attributes. The data looks as follows:

```
>>> from pprint import pprint

>>> pprint(series_I)
[Pair(x=10.0, y=8.04),
 Pair(x=8.0, y=6.95),
 ...
 Pair(x=5.0, y=5.68)]
```

We'll break the rank ordering problem into two parts. First, we'll look at a generic, higher-order function that we can use to assign ranks to any attribute. For example, it can rank a sample according to either the the `x` or `y` attribute value of a `Pair` object. Then, we'll define a wrapper around the `Pair` object that includes the various rank order values.

So far, this seems like a place where we can wrap each pair, sort them into order, then use a function like `enumerate()` to assign ranks. It turns out that this approach isn't really the proper algorithm for rank ordering.

While the essence of rank ordering is being able to sort the samples, there's another important part of this. When two observations have the same value, they should get the same rank. The general rule is to average the positions of equal observations. The sequence `[0.8, 1.2, 1.2, 2.3, 18]` should have rank values of 1, 2.5, 2.5, 4, 5. The two ties with ranks of 2 and 3 have the midpoint value of 2.5 as their common rank.

A consequence of this is that we don't really need to sort all of the data. We can, instead, create a dictionary with a given attribute value and all of the samples that share the attribute value. All these items have a common rank. Given this dictionary, the keys need to be processed in ascending order. For some collections of data, there may be significantly fewer keys than original sample objects being ranked.

The rank ordering function works in two passes:

1. First, it builds a dictionary listing samples with duplicate values. We can call this the `build_duplicates()` phase.
2. Second, it emits a sequence of values, in ranked order, with a mean rank order for the items with the same value. We can call this the `rank_output()` phase.

The following function implements the two-phase ordering via two embedded functions:

```
from collections import defaultdict
from collections.abc import Callable, Iterator, Iterable, Hashable
from typing import NamedTuple, TypeVar, Any, Protocol, cast

BaseT = TypeVar("BaseT", int, str, float)
DataT = TypeVar("DataT")

def rank(
    data: Iterable[DataT],
    key: Callable[[DataT], BaseT]
```

```
) -> Iterator[tuple[float, DataT]]:

    def build_duplicates(
            duplicates: dict[BaseT, list[DataT]],
            data_iter: Iterator[DataT],
            key: Callable[[DataT], BaseT]
    ) -> dict[BaseT, list[DataT]]:
        for item in data_iter:
            duplicates[key(item)].append(item)
        return duplicates

    def rank_output(
            duplicates: dict[BaseT, list[DataT]],
            key_iter: Iterator[BaseT],
            base: int=0
    ) -> Iterator[tuple[float, DataT]]:
        for k in key_iter:
            dups = len(duplicates[k])
            for value in duplicates[k]:
                yield (base+1+base+dups)/2, value
            base += dups

    duplicates = build_duplicates(
        defaultdict(list), iter(data), key
    )
    return rank_output(
        duplicates,
        iter(sorted(duplicates.keys())),
        0
    )
```

As we can see, this rank ordering function has two internal functions to transform a list of samples to a list of two-tuples, each pair having the assigned rank and the original sample object.

To keep the data structure type hints simple, the base type of the sample tuple is defined as `BaseT`, which can be any of the string, integer, or float types. The essential ingredient here

is a simple, hashable, and comparable object.

Similarly, the `DataT` type is any type for the raw samples; the claim is that it will be used consistently throughout the function, and it's two internal functions. This is an intentionally vague claim, because any kind of `NamedTuple`, dataclass, or `PRecord` will work.

The `build_duplicates()` function works with a stateful object to build the dictionary that maps keys to values. This implementation relies on the tail-call optimization of a recursive algorithm. The arguments to `build_duplicates()` expose the internal state as argument values. A base case for a recursive definition is when `data_iter` is empty.

Similarly, the `rank_output()` function could be defined recursively to emit the original collection of values as two-tuples with the assigned rank values. What's shown is an optimized version with two nested `for` statements. To make the rank value computation explicit, it includes the low end of the range (`base+1`), the high end of the range (`base+dups`), and computes the midpoint of these two values. If there is only a single duplicate, the rank value is `(2*base+2)/2`, which has the advantage of being a general solution, resulting in `base+1`, in spite of extra computations.

The dictionary of duplicates has the type hint of `dict[BaseT, list[tuple[BaseT, ...]]]`, because it maps a sample attribute value, `BaseT`, to lists of the original data item type, `tuple[BaseT, ...]`.

The following is how we can test this to be sure it works. The first example ranks individual values. The second example ranks a list of pairs, using a lambda to pick the key value from each pair:

```
>>> from pprint import pprint

>>> data_1 = [(0.8,), (1.2,), (1.2,), (2.3,), (18.,)]
>>> ranked_1 = list(rank(data_1, lambda row: row[0]))
>>> pprint(ranked_1)
```

```
[(1.0, (0.8,)), (2.5, (1.2,)), (2.5, (1.2,)), (4.0, (2.3,)), (5.0,
(18.0,))]

>>> from random import shuffle
>>> shuffle(data_1)
>>> ranked_1s = list(rank(data_1, lambda row: row[0]))
>>> ranked_1s == ranked_1
True

>>> data_2 = [(2., 0.8), (3., 1.2), (5., 1.2), (7., 2.3), (11., 18.)]
>>> ranked_2 = list(rank(data_2, key=lambda x: x[1],))
>>> pprint(ranked_2)
[(1.0, (2.0, 0.8)),
 (2.5, (3.0, 1.2)),
 (2.5, (5.0, 1.2)),
 (4.0, (7.0, 2.3)),
 (5.0, (11.0, 18.0))]
```

The sample data included two identical values. The resulting ranks split positions 2 and 3 to assign position 2.5 to both values. This confirms that the function implements the common statistical practice for computing the Spearman's rank-order correlation between two sets of values.

The `rank()` function involves rearranging the input data as part of discovering duplicated values. If we want to rank on both the x and y values in each pair, we need to reorder the data twice.

Computing Spearman's rank-order correlation

The Spearman rank-order correlation is a comparison between the rankings of two variables. It neatly bypasses the magnitude of the values, and it can often find a correlation even when the relationship is not linear. The formula is as follows:

$$\rho = 1 - \frac{6\sum(r_x - r_y)^2}{n(n^2 - 1)}$$

This formula shows us that we'll be summing the differences in rank r_x, and r_y, for all of the pairs of observed values. This requires computing ranks on both x and y variables. This means merging the two rank values into a single, composite object with ranking combined with the original raw sample.

The target class could look like the following:

```
class Ranked_XY(NamedTuple):
    r_x: float
    r_y: float
    raw: Pair
```

This can be built by first ranking on one variable, then computing a second ranking of the raw data. For a simple dataset with a few variables, this isn't terrible. For more than a few variables, this becomes needlessly complicated. The function definitions, in particular for ranking, would all be nearly identical, suggesting a need to factor out the common code.

It works out much better to depend on using the `pyrsistent` module to create, and evolve, the values in a dictionary that accumulate ranking values. We can use a `PRecord` pair that has a dictionary of rankings and the original data. The dictionary of rankings is an immutable `PMap`. This means that any attempt to make a change will lead to evolving a new instance.

The instance, after being evolved, is immutable. We can clearly separate the accumulation of state from processing objects that do not have any further state changes.

Here's our `PRecord` subclass that contains the ranking mapping and the original, raw data:

```
from pyrsistent import PRecord, field, PMap, pmap

class Ranked_XY(PRecord):  # type: ignore [type-arg]
    rank = field(type=PMap)
    raw = field(type=Pair)
```

Within each `Ranked_XY`, the `PMap` dictionary provides a mapping from the variable name

to the ranking value. The raw data is the original sample. We want to be able to use `sample.rank[attribute_name]` to extract the ranking for a specific attribute.

We can reuse our generic `rank()` function to build the essential information that contains a ranking and the raw data. We can then merge each new ranking into a `Ranked_XY` instance. The following function definition will compute rankings for two attributes:

```
def rank_xy(pairs: Sequence[Pair]) -> Iterator[Ranked_XY]:
    data = list(Ranked_XY(rank=pmap(), raw=p) for p in pairs)

    for attribute_name in ('x', 'y'):
        ranked = rank(
            data,
            lambda rxy: cast(float, getattr(rxy.raw, attribute_name))
        )
        data = list(
            original.set(
                rank=original.rank.set(attribute_name, r)  # type:
                ignore [arg-type]
            )
            for r, original in ranked
        )

    yield from iter(data)
```

We've built an initial list of `Ranked_XY` objects with empty ranking dictionaries. For each attribute of interest, we'll use the previously defined `rank()` function to create a sequence of rank values and raw objects. The `for` clause of the generator decomposes the ranking two-tuple into `r`, the ranking, and `original`, the raw source data.

From each pair of values from the underlying `rank()` function, we've made two changes to the `pyrsistent` module's data structures. We've used the following expression to create a new dictionary of the previous rankings merged with this new ranking:

```
original.rank.set(attribute_name, r)
```

The result of this becomes part of the `original.set(rank=...)` expression to create a new `Rank_XY` object using the newly evolved rank, `PMap` instance.

The `.set()` method is an "evolver": it creates a new object by applying a new state to an existing object. These changes by evolution are important because they result in new, immutable objects.

The `# type: ignore [arg-type]` comment is required to silence a **mypy** warning. The type information used internally by the `pyrsistent` module isn't visible to **mypy**.

A Python version of a rank correlation function depends on the `sum()` and `len()` functions, as follows:

```
from collections.abc import Sequence

def rank_corr(pairs: Sequence[Pair]) -> float:
    ranked = rank_xy(pairs)
    sum_d_2 = sum(
        (r.rank['x'] - r.rank['y']) ** 2  # type: ignore[operator,
        index]
        for r in ranked
    )
    n = len(pairs)
    return 1 - 6 * sum_d_2/(n * (n ** 2 - 1))
```

We've created `Rank_XY` objects for each `Pair` object. Given this, we can then subtract the `r_x` and `r_y` values from those pairs to compare their difference. We can then square and sum the differences.

See *Avoiding stateful classes by using families of tuples* earlier in this chapter for the definition of the `Pair` class.

Again, we've had to suppress **mypy** warnings related to the lack of detailed internal type hints in the `pyrsistent` module. Because this works properly, we feel confident in silencing the warnings.

A good article on statistics will provide detailed guidance on what the coefficient means. A

value around 0 means that there is no correlation between the data ranks of the two series of data points. A scatter plot shows a random scattering of points. A value around +1 or -1 indicates a strong relationship between the two values. A graph of the pairs would show a clear line or simple curve.

The following is an example based on Anscombe's quartet series:

```
>>> data = [Pair(x=10.0, y=8.04),
... Pair(x=8.0, y=6.95),
... Pair(x=13.0, y=7.58), Pair(x=9.0, y=8.81),
... Pair(x=11.0, y=8.33), Pair(x=14.0, y=9.96),
... Pair(x=6.0, y=7.24), Pair(x=4.0, y=4.26),
... Pair(x=12.0, y=10.84), Pair(x=7.0, y=4.82),
... Pair(x=5.0, y=5.68)]
>>> round(pearson_corr(data), 3)
0.816
```

For this particular dataset, the correlation is strong.

In *Chapter 4, Working with Collections*, we showed how to compute the Pearson correlation coefficient. The function we showed, `corr()`, worked with two separate sequences of values. We can use it with our sequence of `Pair` objects as follows:

```
from collections.abc import Sequence
from Chapter04.ch04_ex4 import corr

def pearson_corr(pairs: Sequence[Pair]) -> float:
    X = tuple(p.x for p in pairs)
    Y = tuple(p.y for p in pairs)
    return corr(X, Y)
```

We've unwrapped the `Pair` objects to get the raw values that we can use with the existing `corr()` function. This provides a different correlation coefficient. The Pearson value is based on how well the standardized values compare between two sequences. For many datasets, the difference between the Pearson and Spearman correlations is relatively small.

For some datasets, however, the differences can be quite large.

To see the importance of having multiple statistical tools for exploratory data analysis, compare the Spearman and Pearson correlations for the four sets of data in Anscombe's quartet.

Polymorphism and type pattern matching

Some functional programming languages offer some clever approaches to the problem of working with statically typed function definitions. The problem is that many functions we'd like to write are entirely generic with respect to data type. For example, most of our statistical functions are identical for `int` or `float` numbers, as long as the division returns a value that is a subclass of `numbers.Real`. The types `Decimal`, `Fraction`, and `float` should all work almost identically. In many functional languages, sophisticated type or type-pattern matching rules are used by the compiler to allow a single generic definition to work for multiple data types.

Instead of the (possibly) complex features of statically typed functional languages, Python changes the approach dramatically. Python uses dynamic selection of the final implementation of an operator based on the data types being used. In Python, we always write generic definitions. The code isn't bound to any specific data type. The Python runtime will locate the appropriate operations based on the types of the actual objects in use. The *6.1. Arithmetic conversions* and *3.3.8. Emulating numeric types* sections of the language reference manual and the `numbers` module in the standard library provide details on how this mapping from operation to special method name works.

In Python, there's no compiler to certify that our functions are expecting and producing the proper data types. We generally rely on unit testing and the **mypy** tool for this kind of type checking.

In rare cases, we might need to have different behavior based on the types of data elements. We have two ways to tackle this:

- We can use the `match` statement to distinguish the different cases. This replaces

sequences of `isinstance()` functions to compare argument values against types.

- We can create class hierarchies that provide alternative implementations for methods.

In some cases, we'll actually need to do both so that we can include appropriate data type conversions for an operation. Each class is responsible for the coercion of argument values to a type it can use. The alternative is to return the special `NotImplemented` object, which forces the Python runtime to continue to search for a class that implements the operation and handles the required data types.

The ranking example in the previous section is tightly bound to the idea of applying rank-ordering to simple pairs. It's bound to the `Pair` class definition. While this is the way the Spearman correlation is defined, a multivariate dataset has a need to do rank-order correlation among all the variables.

The first thing we'll need to do is generalize our idea of rank-order information. The following is a `NamedTuple` value that handles a tuple of ranks and a raw data object:

```
from typing import NamedTuple, Any

class RankData(NamedTuple):
    rank_seq: tuple[float, ...]
    raw: Any
```

We can provide a sequence of rankings, each computed with respect to a different variable within the raw data. We might have a data point that has a rank of 2 for the `'key1'` attribute value and a rank of 7 for the `'key2'` attribute value. A typical use of this kind of class definition is shown in this example:

```
>>> raw_data = {'key1': 1, 'key2': 2}
>>> r = RankData((2, 7), raw_data)
>>> r.rank_seq[0]
2
>>> r.raw
{'key1': 1, 'key2': 2}
```

The row of raw data in this example is a dictionary with two keys for the two attribute names. There are two rankings for this particular item in the overall list. An application can get the sequence of rankings as well as the original raw data item.

We'll add some syntactic sugar to our ranking function. In many previous examples, we've required either an iterable or a concrete collection. The `for` statement is graceful about working with either one. However, we don't always use the `for` statement, and for some functions, we've had to explicitly use `iter()` to make an iterator from an iterable collection. (We've also been forced sometimes to use `list()` to materialize an iterable into a concrete collection object.)

Looking back at the `legs()` function shown in *Chapter 4, Working with Collections*, we saw this definition:

```
from collections.abc import Iterator, Iterable
from typing import Any, TypeVar

LL_Type = TypeVar('LL_Type')

def legs(
        lat_lon_iter: Iterator[LL_Type]
) -> Iterator[tuple[LL_Type, LL_Type]]:
    begin = next(lat_lon_iter)
    for end in lat_lon_iter:
        yield begin, end
        begin = end
```

This only works for an `Iterator` object. If we want to use a sequence, we're forced to insert `iter(some_sequence)` to create an iterator from the sequence. This is annoying and error-prone.

The traditional way to handle this situation is with an `isinstance()` check, as shown in the following code snippet:

```
from collections.abc import Iterator, Iterable, Sequence
from typing import Any, TypeVar

# Defined earlier
# LL_Type = TypeVar('LL_Type')

def legs_g(
        lat_lon_src: Iterator[LL_Type] | Sequence[LL_Type]
) -> Iterator[tuple[LL_Type, LL_Type]]:
    if isinstance(lat_lon_src, Sequence):
        return legs_g(iter(lat_lon_src))
    elif isinstance(lat_lon_src, Iterator):
        begin = next(lat_lon_src)
        for end in lat_lon_src:
            yield begin, end
            begin = end
    else:
        raise TypeError("not an Iterator or Sequence")
```

This example includes a type check to handle the small difference between a `Sequence` object and an `Iterator`. Specifically, when the argument value is a sequence, the `legs()` function uses `iter()` to create an `Iterator` from the `Sequence`, and calls itself recursively with the derived value.

This can be done in a slightly nicer and more general manner with type matching. The idea is to handle the variable argument types with a `match` statement that applies any needed conversions to a uniform type that can be processed:

```
from collections.abc import Sequence, Iterator, Iterable
from typing import Any, TypeVar

# Defined earlier
# LL_Type = TypeVar('LL_Type')

def legs_m(
        lat_lon_src: Iterator[LL_Type] | Sequence[LL_Type]
```

```
) -> Iterator[tuple[LL_Type, LL_Type]]:

    match lat_lon_src:
        case Sequence():
            lat_lon_iter = iter(lat_lon_src)
        case Iterator() as lat_lon_iter:
            pass
        case _:
            raise TypeError("not an Iterator or Sequence")

    begin = next(lat_lon_iter)
    for end in lat_lon_iter:
        yield begin, end
        begin = end
```

This example has shown how we can match types to make it possible to work with either sequences or iterators. A great many other type matching capabilities can be implemented in a similar fashion. It may be helpful, for example, to work with string or float values, coercing the string values to float.

It turns out that a type check isn't the only solution to this specific problem. The `iter()` function can be applied to iterators as well as concrete collections. When the `iter()` function is applied to an iterator, it does nothing and returns the iterator. When applied to a collection, it creates an iterator from the collection.

The objective of the `match` statement is to avoid the need to use the built-in `isinstance()` function. The `match` statement provides more matching alternatives with an easier-to-read syntax.

Summary

In this chapter, we looked at different ways to use `NamedTuple` subclasses to implement more complex data structures. The essential features of a `NamedTuple` are a good fit with functional design. They can be created with a creation function and accessed by position as well as name.

Similarly, we looked at frozen dataclasses as an alternative to `NamedTuple` objects. The use of a dataclass seems slightly superior to a `NamedTuple` subclass because a dataclass doesn't also behave like a sequence of attribute values.

We looked at how immutable objects can be used instead of stateful object definitions. The core technique for replacing state changes is to wrap objects in larger objects that contain derived values.

We also looked at ways to handle multiple data types in Python. For most arithmetic operations, Python's internal method dispatch locates proper implementations. To work with collections, however, we might want to handle iterators and sequences slightly differently using the `match` statement.

In the next two chapters, we'll look at the `itertools` module. This standard library module provides a number of functions that help us work with iterators in sophisticated ways. Many of these tools are examples of higher-order functions. They can help a functional design stay succinct and expressive.

Exercises

This chapter's exercises are based on code available from Packt Publishing on GitHub. See `https://github.com/PacktPublishing/Functional-Python-Programming-3rd-Edition`.

In some cases, the reader will notice that the code provided on GitHub includes partial solutions to some of the exercises. These serve as hints, allowing the reader to explore alternative solutions.

In many cases, exercises will need unit test cases to confirm they actually solve the problem. These are often identical to the unit test cases already provided in the GitHub repository. The reader should replace the book's example function name with their own solution to confirm that it works.

Frozen dictionaries

A dictionary with optional key values can be a source of confusing state change management. Python's implementation of objects generally relies on an internal dictionary, named `__dict__`, to keep an object's attribute values. This is easy to mirror in application code, and it can create problems.

While dictionary updates can be confusing, the previously described use case seems rare. A much more common use of dictionaries is to load a mapping from a source, and then use the mapping during later processing. One example is a dictionary that contains translations from source encodings to more useful numeric values. It might use a mapping value like this: `{"y": 1, "Y": 1, "n": 0, "N": 0}`. In this case, the dictionary is created once, and does not change state after that. It's effectively frozen.

Python doesn't have a built-in frozen dictionary class. One approach to defining this class is to extend the built-in `dict` class, adding a mode change. There would be two modes: "load" and "query." A dictionary in "load" mode can have keys and values created. A dictionary in "query" mode, however, does not permit changes. This includes refusing to go back to "load" mode. This is an extra layer of stateful behavior that permits or denies the underlying mapping behavior.

The dictionary class has a long list of special methods like `__setitem__()` and `update()` that change the internal state. The Python Language Reference, section 3.3.7 *Emulating Container Types*, provides a detailed list of methods that change the state of a mapping. Additionally, the Library Reference, in a section named *Mapping Types – dict*, provides a list of methods for the built-in `dict` class. Finally, the `collections.abc` module also defines some of the methods that mappings must implement.

Work out the list of methods that must be implemented with code like the following example:

```
    def some_method(self, *args: Any, **kwargs: Any) -> None:
        if self.frozen:
            raise RuntimeError("mapping is frozen")
        else:
            super.some_method(*args, **kwargs)
```

Given the list of methods that need this kind of wrapper, comment on the value of having a frozen mapping. Contrast the work required to implement this class with the possible confusion from a stateful dictionary. Provide a cost-benefit justification for either writing this class or setting the idea aside and looking for a better solution. Recall that dictionary key look-ups are very fast, relying on a hash computation instead of a lengthy search.

Dictionary-like sequences

Python doesn't have a built-in frozen dictionary class. One approach to defining this class is to leverage the `bisect` module to build a list. The list is maintained in sorted order, and the `bisect` module can do relatively rapid searches of a sorted list.

For an unsorted list, the complexity of finding a specific item in a sequence of n items is $\mathbf{O}(n)$. For a sorted list, the `bisect` module can reduce this to $\mathbf{O}(\log_2 n)$, a significant reduction in time for a large list. (And, of course, a dictionary's hashed lookup is generally $\mathbf{O}(1)$, which is better still.)

The dictionary class has a long list of special methods like `__setitem__()` and `update()` that change the internal state. The previous exercise provides some pointers for locating all of the special methods that are relevant to building a dictionary.

A function to build a sorted list can wrap `bisect.insort_left()`. A function to query the sorted list can leverage `bisect.bisect_left()` to locate and then return the value associated with a key that's in the list, or raise a `KeyError` exception for an item that's not in the list.

Build a small demo application that creates a dictionary from a source file, then does several thousand randomized retrievals from that dictionary. Compare the time required to run

the demo using the built-in `dict` against the `bisect`-based dictionary-like list.

Using the built-in `sys.getallocatedblocks()`, compare the memory used by a list of values and the memory used by a dictionary of values. For this to be meaningful, the dictionary will need several thousand keys and values. A pool of random numbers and randomly generated strings can be useful for this comparison.

Revise the rank_xy() function to use native types

In the *Computing Spearman's rank-order correlation* section, we presented a `rank_xy()` function that created a `pyrsistent.PMap` object with various ranking positions. This was contained within a `PRecord` subclass.

First, rewrite the function (and the type hints) to use either a named tuple or a dataclass instead of a `PRecord` subclass. This replaces one immutable object with another.

Next, consider replacing the `PMap` object with a native Python dictionary. Since dictionaries are mutable, what additional processing is needed to create a copy of a dictionary before adding a new ranking value?

After revising the `PMap` object to a dictionary, compare the performance of the `pyrsistent` objects with native objects. What conclusions can you draw?

Revise the rank_corr() function

In the *Polymorphism and type pattern matching* section, we presented a way to create `RankedSample` objects that contain rankings and underlying `Rank_Data` objects with the raw sample value.

Rewrite the `rank_corr()` function to compute the rank correlations of any of the available values in the `rank_seq` attribute of the `RankedSample` objects.

Revise the legs() function to use pyrsistent

In the *Using pyrsistent to collect data* section, a number of functions were reused from earlier examples. The `legs()` function was called out specifically. The entire parsing pipeline,

however, can be rewritten to use the `pyrsistent` variations on the fundamental, immutable object classes.

After doing the revision, explain any improvements in the code from using one module consistently for the data collections. Create an application that loads and computes distances for a trip several thousand times. Use each of the various representations and accumulate timing data to see which, if any, is faster.

8

The Itertools Module

Functional programming emphasizes stateless objects. In Python, this leads us to work with generator expressions, generator functions, and iterables, instead of large, mutable collection objects. In this chapter, we'll look at elements of the `itertools` library. This library has numerous functions to help us work with iterable sequences of objects, as well as collection objects.

We introduced iterator functions in *Chapter 3*, *Functions, Iterators, and Generators*. In this chapter, we'll expand on that superficial introduction. We used some related functions in *Chapter 5*, *Higher-Order Functions*.

There are a large number of iterator functions in the `itertools` module. We'll examine the combinatoric functions in the next chapter. In this chapter, we'll look at the following three broad groupings of the remaining iterator functions:

- Functions that work with potentially infinite iterators. These can be applied to any iterable or an iterator over any collection. For example, the `enumerate()` function doesn't require an upper bound on the number of items in the iterable.

- Functions that work with finite iterators. Often, these are used to create a reduction of the source. For example, grouping the items produced by an iterator reduces the source to groups of items with a common key.
- The `tee()` iterator function clones an iterator into several copies that can each be used independently. This provides a way to overcome the primary limitation of Python iterators: they can be used only once. This is memory-intensive, however, and redesign is often required.

We need to emphasize the important limitation of iterables that we've touched upon in other places: they can only be used once.

Iterables can be used only once.

This can be astonishing because there's no error exception raised by attempting to reuse an iterator that's been consumed fully. Once exhausted, they appear to have no elements and will only raise the `StopIteration` exception every time they're used.

There are some other features of iterators that don't involve such profound limitations. Note that many Python functions, as well as the `for` statement, will use the built-in `iter()` function to create as many iterators as required from a collection object.

Other features of iterators include:

- There's no `len()` function for an iterator.
- Iterators, a subclass of iterables, can do `next()` operations, unlike a container. We'll often use the built-in `iter()` to create an iterator that has a `next()` operation.
- The `for` statement makes the distinction between containers and other iterables invisible by evaluating the built-in `iter()` function. A container object, for example, a list, responds to this function by producing an iterator over the items. An iterable object that's not a collection, for example, a generator function, returns itself, since it is designed to follow the `Iterator` protocol.

These points will provide some necessary background for this chapter. The idea of the `itertools` module is to leverage what iterables can do to create succinct, expressive applications without the complicated-looking overheads associated with the details of managing the iterables.

Working with the infinite iterators

The `itertools` module provides a number of functions that we can use to enhance or enrich an iterable source of data. We'll look at the following three functions:

- `count()`: This is an unlimited version of the `range()` function. An upper bound must be imposed by the consumer of this sequence.
- `cycle()`: This will reiterate a cycle of values. The consumer must decide when enough values have been produced.
- `repeat()`: This can repeat a single value an indefinite number of times. The consumer must end the repetition.

Our goal is to understand how these various iterator functions can be used in generator expressions and with generator functions.

Counting with count()

The built-in `range()` function is defined by an upper limit: the lower limit and step values are optional. The `count()` function, on the other hand, has a start and optional step, but no upper limit.

This function can be thought of as the primitive basis for a function such as the built-in `enumerate()` function. We can define the `enumerate()` function in terms of `zip()` and `count()` functions, as follows:

```
>>> from itertools import count
>>> enumerate = lambda x, start=0: zip(count(start), x)
```

The `enumerate()` function behaves as if it's a `zip()` function that uses the `count()` function

to generate the values associated with some iterable source of objects.

Consequently, the following two expressions are equivalent to each other:

```
>>> list(zip(count(), iter('word')))
[(0, 'w'), (1, 'o'), (2, 'r'), (3, 'd')]
>>> list(enumerate(iter('word')))
[(0, 'w'), (1, 'o'), (2, 'r'), (3, 'd')]
```

Both will emit a sequence of numbers of two-tuples. The first item in each tuple is an integer counter. The second item comes from the iterator. In this example, the iterator is built from a string of characters.

Here's something we can do with the `count()` function that's difficult to do with the `enumerate()` function:

```
>>> list(zip(count(1, 3), iter('word')))
[(1, 'w'), (4, 'o'), (7, 'r'), (10, 'd')]
```

The value of `count(b, s)` is the sequence of values $\{b, b + s, b + 2s, b + 3s, ...\}$. In this example, it will provide values of 1, 4, 7, 10, and so on as the identifiers for each value from the enumerator. The `enumerate()` function doesn't provide a way to change the step.

We can, of course, combine generator functions to achieve this result. Here's how changing the step can be done with the `enumerate()` function:

```
>>> source = iter('word')
>>> gen3 = ((1+3*e, x) for e, x in enumerate(source))
>>> list(gen3)
[(1, 'w'), (4, 'o'), (7, 'r'), (10, 'd')]
```

This shows how a new value, $1 + 3e$, is computed from the source enumeration value of e. This behaves like the sequence started at 1 and is incremented by 3.

Counting with float arguments

The `count()` function permits non-integer values. We can use something such as the `count(0.5, 0.1)` expression to provide floating-point values. This will accumulate an error if the increment value doesn't have an exact representation. It's generally better to use integer `count()` arguments such as `(0.5+x*.1 for x in count())` to ensure that representation errors don't accumulate.

Here's a way to examine the accumulating error. This exploration of the float approximation shows some interesting functional programming techniques.

We'll define a function that will evaluate items from an iterator until some condition is met. This is a way to find the first item that meets some criteria defined by a function. Here's how we can define a `find_first()` function:

```
from collections.abc import Callable, Iterator
from typing import TypeVar
T = TypeVar("T")

def find_first(
    terminate: Callable[[T], bool],
    iterator: Iterator[T]
) -> T:
    i = next(iterator)
    if terminate(i):
        return i
    return find_first(terminate, iterator)
```

This function starts by getting the next value from the iterator object. No specific type is provided; the type variable `T` tells **mypy** that the source iterator and the target result will be the same type. If the chosen item passes the test, that is, this is the desired value, iteration stops and the return value will be of the given type associated with the type variable, `T`. Otherwise, we'll evaluate this function recursively to search for a subsequent value that passes the test.

Because the tail-call recursion is not replaced with an optimized `for` statement, this is

limited to iterables with about 1,000 items.

If we have some series of values computed by a generator, this will consume items from the iterator. Here's a silly example. Let's say we have an approximation that is a sum of a series of values. One example is this:

$$\pi = 4 \arctan(1) = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} \cdots\right)$$

The terms of this series can be created by a generator function like this:

```
>>> def term_iter():
...     d = 1
...     sgn = 1
...     while True:
...         yield Fraction(sgn, d)
...         d += 2
...         sgn = -1 if sgn == 1 else 1
```

This will yield values like `Fraction(1, 1)`, `Fraction(-1, 3)`, `Fraction(1, 5)`, and `Fraction(-1, 7)`. It will yield an infinite number of them. We want values up until the first value that meets some criteria. For example, we may want to know the first value that will be less than $\frac{1}{100}$ (this is pretty easy to work out with pencil and paper to check the results):

```
>>> find_first(lambda v: abs(v) < 1E-2, term_iter())
Fraction(1, 101)
```

Our goal is to compare counting with float values against counting with integer values and then applying a scaling factor. We want to define a source that has both sequences as pairs. As an introduction to the concept, we'll look at generating pairs from two parallel sources. Then we'll return to the computation shown above.

In the following example, the `source` object is a generator of the pairs of pure float and

int-to-float values:

```
from itertools import count
from collections.abc import Iterator
from typing import NamedTuple, TypeAlias

Pair = NamedTuple('Pair', [('flt_count', float), ('int_count',
float)])
Pair_Gen: TypeAlias = Iterator[Pair]

source: Pair_Gen = (
  Pair(fc, ic) for fc, ic in
  zip(count(0, 0.1), (.1*c for c in count()))
)

def not_equal(pair: Pair) -> bool:
  return abs(pair.flt_count - pair.int_count) > 1.0E-12
```

The `Pair` tuple will have two float values: one generated by summing float values, and the other generated by counting integers and multiplying by a floating-point scaling factor.

The generator, `source`, has provided a type hint on the assignment statement to show that it iterates over the pairs.

When we evaluate the `find_first(not_equal, source)` method, we'll repeatedly compare float approximations of decimal values until they differ. One is a sum of 0.1 values: $0.1 \times \sum_{x \in \mathbb{N}} 1$. The other is a sum of integer values, weighted by 0.1: $\sum_{x \in \mathbb{N}} 0.1$. Viewed as abstract mathematical definitions, there's no distinction.

We can formalize it as follows:

$$0.1 \times \sum_{x \in \mathbb{N}} 1 \equiv \sum_{x \in \mathbb{N}} 0.1$$

With concrete approximations of the abstract numbers, however, the two values will differ. The result is as follows:

```
>>> find_first(not_equal, source)
Pair(flt_count=92.799999999999, int_count=92.80000000000001)
```

After about 928 iterations, the sum of the error bits has accumulated to 10^{-12}. Neither value has an exact binary representation.

The `find_first()` function example is close to the Python recursion limit. We'd need to rewrite the function to use tail-call optimization to locate examples with a larger cumulative error value.

We've left this as change as an exercise for the reader.

The smallest detectable difference can be computed as follows:

```
>>> source: Pair_Gen = map(Pair, count(0, 0.1), (.1*c for c in
count()))

>>> find_first(lambda pair: pair.flt_count != pair.int_count, source)
Pair(flt_count=0.6, int_count=0.6000000000000001)
```

This uses a simple equality check instead of an error range. After six steps, the `count(0, 0.1)` method has accumulated a tiny, but measurable, error of 10^{-16}. While small, these error values can accumulate to become more significant and visible in a longer computation. When looking at how $\frac{1}{10}$ is represented as a binary value, an infinite binary expansion would be required. This is truncated to about $10^{-16} \approx 2^{-53}$ from the conceptual value. The magic number 53 is the number of bits available in IEEE standard for 64-bit floating-point values.

This is why we generally count things with ordinary integers and apply a weighting to compute a floating-point value.

Re-iterating a cycle with cycle()

The `cycle()` function repeats a sequence of values. This can be used when partitioning data into subsets by cycling among the dataset identifiers.

We can imagine using it to solve silly fizz-buzz problems. Visit `http://rosettacode.org/wiki/FizzBuzz` for a comprehensive set of solutions to a fairly trivial programming problem. Also see `https://projecteuler.net/problem=1` for an interesting variation on this theme.

We can use the `cycle()` function to emit sequences of `True` and `False` values as follows:

```
>>> from itertools import cycle

>>> m3 = (i == 0 for i in cycle(range(3)))
>>> m5 = (i == 0 for i in cycle(range(5)))
```

These two generator expressions can produce infinite sequences with a pattern of `[True, False, False, True, False, False, ...]` or `[True, False, False, False, False, True, False, False, False, False, ...]`. These are iterators and can only be consumed once. They will tend to maintain their internal state. If we don't consume precisely 15 values, the least common multiple of their cycles, the next time we consume values, they will be in an unexpected, in-between state.

If we zip together a finite collection of numbers and these two derived values, we'll get a set of three-tuples with a number, the multiple of three true-false condition, and the multiple of five true-false condition. It's important to introduce a finite iterable to create a proper upper bound on the volume of data being generated. Here's a sequence of values and their multiplier conditions:

```
>>> multipliers = zip(range(10), m3, m5)
```

This is a generator; we can use `list(multipliers)` to see the resulting object. It looks like this:

```
>>> list(multipliers)
[(0, True, True), (1, False, False), (2, False, False), ..., (9, True,
False)]
```

We can now decompose the triples and use a filter to pass numbers that are multiples and reject all others:

```
>>> multipliers = zip(range(10), m3, m5)
>>> total = sum(i
...     for i, *multipliers in multipliers
...     if any(multipliers)
... )
```

The `for` clause decomposes each triple into two parts: the value, `i`, and the flags, `multipliers`. If any of the multipliers are true, the value is passed; otherwise, it's rejected.

The `cycle()` function has another, more valuable, use for exploratory data analysis.

Using cycle() for data sampling

We often need to work with samples of large sets of data. The initial phases of cleansing and model creation are best developed with small sets of data and tested with larger and larger sets of data. We can use the `cycle()` function to fairly select rows from within a larger set. This is distinct from making random selections and trusting the fairness of the random number generator. Because this approach is repeatable and doesn't rely on a random number generator, it can be applied to very large datasets processed by multiple computers.

Given a population size, N_p, and the desired sample size, N_s, this is the required size of the cycle, c, that will produce appropriate subsets:

$$c = \frac{N_p}{N_s}$$

We'll assume that the data can be parsed with a common library like the `csv` module. This

leads to an elegant way to create subsets. Given a value for the `cycle_size` and two open files, `source_file` and `target_file`, we can create subsets using the following function definition:

```
from collections.abc import Iterable, Iterator
from itertools import cycle
from typing import TypeVar
DT = TypeVar("DT")

def subset_iter(
         source: Iterable[DT], cycle_size: int
) -> Iterator[DT]:
    chooser = (x == 0 for x in cycle(range(cycle_size)))
    yield from (
        row
        for keep, row in zip(chooser, source)
        if keep
    )
```

The `subset_iter()` function uses a `cycle()` function based on the selection factor, `cycle_size`. For example, we might have a population of ten million records; a 1,000-record subset would be built with `cycle_size` set to $c = \frac{10^7}{10^3} = 10{,}000$. We'd keep one record in ten thousand.

The `subset_iter()` function can be used by a function that reads from a source file and writes a subset to a destination file. This processing is part of the following function definition:

```
import csv
from pathlib import Path

def csv_subset(
        source: Path, target: Path, cycle_size: int = 3
) -> None:
    with (
            source.open() as source_file,
```

```
            target.open('w', newline='') as target_file
    ):
        rdr = csv.reader(source_file, delimiter='\t')
        wtr = csv.writer(target_file)
        wtr.writerows(subset_iter(rdr, cycle_size))
```

We can use this generator function to filter the data using the `cycle()` function and the source data that's available from the `csv` reader. Since the chooser expression and the expression used to write the rows are both non-strict, there's little memory overhead from this kind of processing.

We can also rewrite this method to use `compress()`, `filter()`, and `islice()` functions, as we'll see later in this chapter.

This design can also be used to reformat a file from any non-standard CSV-like format into a standardized CSV format. As long as we define a parser function that returns consistently defined tuples of strings and write consumer functions that write tuples to the target files, we can do a great deal of cleansing and filtering with relatively short, clear scripts.

Repeating a single value with repeat()

The `repeat()` function seems like an odd feature: it returns a single value over and over again. It can serve as an alternative for the `cycle()` function when a single value is needed.

The difference between selecting all of the data and selecting a subset of the data can be expressed with this. The expression `(x==0 for x in cycle(range(size)))` emits a `[True, False, False, ...]` pattern, suitable for picking a subset. The function `(x==0 for x in repeat(0))` emits a `[True, True, True, ...]` pattern, suitable for selecting all of the data.

We can think of the following kinds of commands:

```
from itertools import cycle, repeat

def subset_rule_iter(
        source: Iterable[DT], rule: Iterator[bool]
) -> Iterator[DT]:
    return (
        v
        for v, keep in zip(source, rule)
        if keep
    )

all_rows = lambda: repeat(True)
subset = lambda n: (i == 0 for i in cycle(range(n)))
```

This allows us to make a single parameter change, which will either pick all data or pick a subset of data. We can also use `cycle([True])` instead of `repeat(True)`; the results are identical.

This pattern can be extended to randomize the subset chosen. The following technique adds an additional kind of choice:

```
import random

def randomized(limit: int) -> Iterator[bool]:
    while True:
        yield random.randrange(limit) == 0
```

The `randomized()` function generates a potentially infinite sequence of random numbers over a given range. This fits the pattern of `cycle()` and `repeat()`.

This allows code such as the following:

```
>>> import random
>>> random.seed(42)
>>> data = [random.randint(1, 12) for _ in range(12)]
```

```
>>> data
[11, 2, 1, 12, 5, 4, 4, 3, 12, 2, 11, 12]

>>> list(subset_rule_iter(data, all_rows()))
[11, 2, 1, 12, 5, 4, 4, 3, 12, 2, 11, 12]
>>> list(subset_rule_iter(data, subset(3)))
[11, 12, 4, 2]

>>> random.seed(42)
>>> list(subset_rule_iter(data, randomized(3)))
[2, 1, 4, 4, 3, 2]
```

This provides us the ability to use a variety of techniques for selecting subsets. A small change among available functions `all()`, `subset()`, and `randomized()` lets us change our sampling approach in a way that seems succinct and expressive.

Using the finite iterators

The `itertools` module provides a number of functions that we can use to produce finite sequences of values. We'll look at 10 functions in this module, plus some related built-in functions:

- `enumerate()`: This function is actually part of the `__builtins__` package, but it works with an iterator and is very similar to functions in the `itertools` module.
- `accumulate()`: This function returns a sequence of reductions of the input iterable. It's a higher-order function and can do a variety of clever calculations.
- `chain()`: This function combines multiple iterables serially.
- `groupby()`: This function uses a function to decompose a single iterable into a sequence of iterables over subsets of the input data.
- `zip_longest()`: This function combines elements from multiple iterables. The built-in `zip()` function truncates the sequence at the length of the shortest iterable. The `zip_longest()` function pads the shorter iterables with the given fill value.
- `compress()`: This function filters one iterable based on a second, parallel iterable of

Boolean values.

- `islice()`: This function is the equivalent of a slice of a sequence when applied to an iterable.
- `dropwhile()` and `takewhile()`: Both of these functions use a Boolean function to filter items from an iterable. Unlike `filter()` or `filterfalse()`, these functions rely on a single `True` or `False` value to change their filter behavior for all subsequent values.
- `filterfalse()`: This function applies a filter function to an iterable. This complements the built-in `filter()` function.
- `starmap()`: This function maps a function to an iterable sequence of tuples using each iterable as an `*args` argument to the given function. The `map()` function does a similar thing using multiple parallel iterables.

We'll start with functions that could be seen as useful for grouping or arranging items of an `Iterator`. After that, we'll look at functions that are more appropriate for filtering and mapping the items.

Assigning numbers with enumerate()

In the *Using enumerate() to include a sequence number* section of *Chapter 4*, *Working with Collections*, we used the `enumerate()` function to make a naive assignment of rank numbers to sorted data. We can do things such as pairing up a value with its position in the original sequence, as follows:

```
>>> raw_values = [1.2, .8, 1.2, 2.3, 11, 18]

>>> tuple(enumerate(sorted(raw_values)))
((0, 0.8), (1, 1.2), (2, 1.2), (3, 2.3), (4, 11), (5, 18))
```

This will sort the items in `raw_values` in order, create two-tuples with an ascending sequence of numbers, and materialize an object we can use for further calculations.

In *Chapter 7*, *Complex Stateless Objects*, we implemented an alternative form of the `enumerate()`

function, the `rank()` function, which handles ties in a more statistically useful way.

Enumerating rows of data is a common feature that is added to a parser to record the source data row numbers. In many cases, we'll create some kind of `row_iter()` function to extract the string values from a source file. This may iterate over the string values in tags of an XML file or in columns of a CSV file. In some cases, we may even be parsing data presented in an HTML file parsed with Beautiful Soup.

In *Chapter 4, Working with Collections*, we parsed an XML file to create a simple sequence of position tuples. We then created legs with a start, end, and distance. We did not, however, assign an explicit leg number. If we ever sorted the trip collection, we'd be unable to determine the original ordering of the legs.

In *Chapter 7, Complex Stateless Objects*, we expanded on the basic parser to create named tuples for each leg of the trip. The output from this enhanced parser looks as follows:

```
>>> from textwrap import wrap
>>> from pprint import pprint

>>> trip[0]
LegNT(start=PointNT(latitude=37.54901619777347,
longitude=-76.33029518659048), ...

>>> pprint(wrap(str(trip[0])))
['LegNT(start=PointNT(latitude=37.54901619777347,',
 'longitude=-76.33029518659048), end=PointNT(latitude=37.840832,',
 'longitude=-76.273834), distance=17.7246)']
>>> pprint(wrap(str(trip[-1])))
['LegNT(start=PointNT(latitude=38.330166, longitude=-76.458504),',
 'end=PointNT(latitude=38.976334, longitude=-76.473503),',
 'distance=38.8019)']
```

The value of `trip[0]` is quite wide, too wide for the book. To keep the output in a form that fits in this book's pages, we've wrapped the string representation of the value, and used `pprint` to show the individual lines. The first `Leg` object is a short trip between two

points on the Chesapeake Bay.

We can add a function that will build a more complex tuple with the input order information as part of the tuple. First, we'll define a slightly more complex version of the `Leg` class:

```
from typing import NamedTuple

class Point(NamedTuple):
    latitude: float
    longitude: float

class Leg(NamedTuple):
    order: int
    start: Point
    end: Point
    distance: float
```

The `Leg` definition is similar to the variations shown in *Chapter 7, Complex Stateless Objects*, specifically the `LegNT` definition. We'll define a function that decomposes pairs and creates `Leg` instances as follows:

```
from typing import Iterator
from Chapter04.ch04_ex1 import haversine

def numbered_leg_iter(
    pair_iter: Iterator[tuple[Point, Point]]
) -> Iterator[Leg]:
    for order, pair in enumerate(pair_iter):
        start, end = pair
        yield Leg(
            order,
            start,
            end,
            round(haversine(start, end), 4)
        )
```

We can use this function to enumerate each pair of start and end points. We'll de-

compose the pair and then re-assemble the `order`, `start`, and `end` parameters and the `haversine(start,end)` parameter's value as a single `Leg` instance. This generator function will work with an iterable sequence of pairs.

In the context of the preceding explanation, it is used as follows:

```
>>> from Chapter06.ch06_ex3 import row_iter_kml
>>> from Chapter04.ch04_ex1 import legs, haversine
>>> import urllib.request

>>> source_url = "file:./Winter%202012-2013.kml"
>>> with urllib.request.urlopen(source_url) as source:
...     path_iter = float_lat_lon(row_iter_kml(source))
...     pair_iter = legs(path_iter)
...     trip_iter = numbered_leg_iter(pair_iter)
...     trip = list(trip_iter)
```

We've parsed the original file into the path points, created start-end pairs, and then created a trip that was built of individual `Leg` objects. The `enumerate()` function ensures that each item in the iterable sequence is given a unique number that increments from the default starting value of 0. A second argument value to the `enumerate()` function can be given to provide a different starting value.

Running totals with accumulate()

The `accumulate()` function folds a given function into an iterable, accumulating a series of reductions. This will iterate over the running totals from another iterator; the default function is `operator.add()`. We can provide alternative functions to change the essential behavior from sum to product. The Python library documentation shows a particularly clever use of the `max()` function to create a sequence of maximum values so far.

One application of running totals is quartiling data. The **quartile** is one of many measures of position. The general approach is to multiply a sample's value by a scaling factor to convert it to the quartile number. If values range from $0 \le v_i < N$, we can scale by $\lceil \frac{N}{4} \rceil$ to convert any value, v_i, to a value in the range 0 to 3, which map to the various quartiles. The

`math.ceil()` function is used to round the scaling fraction up to the next higher integer. This will ensure that no scaled value will produce a scaled result of 4, an impossible fifth quartile.

If the minimum value of v_i is not zero, we'll need to subtract this from each value before multiplying by the scaling factor.

In the *Assigning numbers with enumerate()* section, we introduced a sequence of latitude-longitude coordinates that describe a sequence of legs on a voyage. We can use the distances as a basis for quartiling the waypoints. This allows us to determine the midpoint in the trip.

See the previous section for the value of the `trip` variable. The value is a sequence of `Leg` instances. Each `Leg` object has a start point, an end point, and a distance. The calculation of quartiles looks like the following code:

```
>>> from itertools import accumulate
>>> import math

>>> distances = (leg.distance for leg in trip)
>>> distance_accum = list(accumulate(distances))
>>> scale = math.ceil(distance_accum[-1] / 4)

>>> quartiles = list(int(scale*d) for d in distance_accum)
```

We extracted the distance values and computed the accumulated distances for each leg. The last of the accumulated distances is the total. The value of the `quartiles` variable is as follows:

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
```

We can use the `zip()` function to merge this sequence of quartile numbers with the original

data points. We can also use functions such as `groupby()` to create distinct collections of the legs in each quartile.

Combining iterators with chain()

A collection of iterators can be unified into a single sequence of values via the `chain()` function. This can be helpful to combine data that was decomposed via the `groupby()` function. We can use this to process a number of collections as if they were a single collection.

Python's `contextlib` offers a clever class, `ExitStack()`, which can be used to perform a number of operations at the end of the context in a `with` statement. This permits an application to create any number of sub-contexts, all of which will have a proper `__enter__()` and `__exit__()` evaluated. This is particularly useful when we have an indefinite number of files to open.

In this example, we can combine the `itertools.chain()` function with a `contextlib.ExitStack` object to process—and properly close—a collection of files. Further, the data from all of these files will be processed as a single iterable sequence of values. Instead of wrapping each individual file operation in a `with` statement, we can wrap all of the operations in a single `with` context.

We can create a single context for multiple files like this:

```
import csv
from collections.abc import Iterator
from contextlib import ExitStack
from pathlib import Path
from typing import TextIO

def row_iter_csv_tab(*filepaths: Path) -> Iterator[list[str]]:
    with ExitStack() as stack:
        files: list[TextIO] = [
            stack.enter_context(path.open())
            for path in filepaths
```

```
        ]
        readers = map(
            lambda f: csv.reader(f, delimiter='\t'),
            files)
        yield from chain(*readers)
```

We've created an `ExitStack` object that can contain a number of individual contexts open. When the `with` statement finishes, all items in the `ExitStack` object will be closed properly. In the above function, a sequence of open file objects is assigned to the `files` variable. The `stack.enter_context()` method enters these objects into the `ExitStack` object to be properly closed.

Given the sequence of files in the `files` variable, we created a sequence of CSV readers in the `readers` variable. In this case, all of our files have a common tab-delimited format, which makes it very pleasant to open them with a simple, consistent application of a function to the sequence of files.

Finally, we chained all of the readers into a single iterator with `chain(*readers)`. This was used to yield the sequence of rows from all of the files.

It's important to note that we can't return the `chain(*readers)` object. If we do, this would exit the `with` statement context, closing all the source files. Instead, we must yield individual rows from the generator so that the `with` statement context is kept active until all the rows are consumed.

Partitioning an iterator with groupby()

We can use the `groupby()` function to partition an iterator into smaller iterators. This works by evaluating the given `key` function for each item in the given iterable. If the key value matches the previous item's key, the two items are part of the same partition. If the key does not match the previous item's key, the previous partition is ended and a new partition is started. Because the matching is done on adjacent items in the iterable, the values must be sorted by the key.

The output from the `groupby()` function is a sequence of two-tuples. Each tuple has the group's key value and an iterable over the items in the group, something like `[(key, iter(group)), (key, iter(group)), ...]`. Each group's iterator can then be processed to create a materialized collection, or perhaps reduce it to some summary value.

In the *Running totals with accumulate()* section, earlier in the chapter, we showed how to compute quartile values for an input sequence. We'll extend that to create groups based on the distance quartiles. Each group will be an iterator over legs that fit into the range of distances.

Given the `trip` variable with the raw data and the `quartile` variable with the quartile assignments, we can group the data using the following commands:

```
>>> from itertools import groupby
>>> from Chapter07.ch07_ex1 import get_trip

>>> source_url = "file:./Winter%202012-2013.kml"
>>> trip = get_trip(source_url)
>>> quartile = quartiles(trip)
>>> group_iter = groupby(zip(quartile, trip), key=lambda q_raw:
q_raw[0])
>>> for group_key, group_iter in group_iter:
...     print(f"Group {group_key+1}: {len(list(group_iter))} legs")
Group 1: 23 legs
Group 2: 14 legs
Group 3: 19 legs
Group 4: 17 legs
```

This will start by zipping the quartile numbers with the raw trip data, creating an iterator over two-tuples with quartile number and leg. The `groupby()` function will use the given lambda object to group by the quartile number, `q_raw[0]`, in each `q_raw` tuple. We used a `for` statement to examine the results of the `groupby()` function. This shows how we get a group key value and an iterator over members of each individual group.

The input to the `groupby()` function must be sorted by the key values. This will ensure

that all of the items in a group will be adjacent. For very large datasets, this may force us to use the operating system's sort in the rare cases of a file being too large to fit into memory.

Note that we can also create groups using a `defaultdict(list)` object. This avoids a sort step, but can build a large, in-memory dictionary of lists. The function can be defined as follows:

```
from collections import defaultdict
from collections.abc import Iterable, Callable, Hashable

DT = TypeVar("DT")
KT = TypeVar("KT", bound=Hashable)

def groupby_2(
    iterable: Iterable[DT],
    key: Callable[[DT], KT]
) -> Iterator[tuple[KT, Iterator[DT]]]:
    groups: dict[KT, list[DT]] = defaultdict(list)
    for item in iterable:
        groups[key(item)].append(item)
    for g in groups:
        yield g, iter(groups[g])
```

We created a `defaultdict` object that will use `list()` as the default value associated with each new key. The type hints clarify the relationship between the `key` function, which emits objects of some arbitrary type associated with the type variable `KT`, and the dictionary, which uses the same type, `KT`, for the keys.

Each item will have the given `key()` function applied to create a key value. The item is appended to the list in the `defaultdict` object with the given key.

Once all of the items are partitioned, we can then return each partition as an iterator over the items that share a common key. This will retain all of the original values in memory, and introduce a dictionary and a list for each unique key value. For very large datasets, this may require more memory than is available on the processor.

The type hints clarify that the source is some arbitrary type, associated with the variable `DT`. The result will be an iterator that includes iterators of the type `DT`. This makes a strong statement that no transformation is happening: the range type matches the input domain type.

Merging iterables with zip_longest() and zip()

We saw the `zip()` function in *Chapter 4, Working with Collections*. The `zip_longest()` function differs from the `zip()` function in an important way: whereas the `zip()` function stops at the end of the shortest iterable, the `zip_longest()` function pads short iterables with a given value, and stops at the end of the longest iterable.

The `fillvalue=` keyword parameter allows filling with a value other than the default value, `None`.

For most exploratory data analysis applications, padding with a default value is statistically difficult to justify. The **Python Standard Library** document includes the `grouper` recipe that can be done with the `zip_longest()` function. It's difficult to expand on this without drifting far from our focus on data analysis.

Creating pairs with pairwise())

The `pairwise()` function consumes a source iterator, emitting the items in pairs. See the `legs()` function in *Chapter 4, Working with Collections*, for an example of creating pairs from a source iterable.

Here's a small example of transforming a sequence of characters into adjacent pairs of characters:

```
>>> from itertools import pairwise

>>> text = "hello world"
>>> list(pairwise(text))
[('h', 'e'), ('e', 'l'), ('l', 'l'), ...]
```

This kind of analysis locates letter pairs, called "bigrams" or "digraphs." This can be helpful

when trying to understand a simple letter substitution cipher. The frequency of bigrams in encoded text can suggest possible ways to break the cipher.

In Python 3.10, this function was moved from being a recipe to being a proper `itertools` function.

Filtering with compress()

The built-in `filter()` function uses a predicate to determine whether an item is passed or rejected. Instead of a function that calculates a value, we can use a second, parallel iterable to determine which items to pass and which to reject.

In the *Re-iterating a cycle with cycle()* section of this chapter, we looked at data selection using a simple generator expression. Its essence was as follows:

```
from typing import TypeVar

DataT = TypeVar("DataT")

def subset_gen(
        data: Iterable[DataT], rule: Iterable[bool]
) -> Iterator[DataT]:
    return (
        v
        for v, keep in zip(data, rule)
        if keep
    )
```

Each value for the rule iterable must be a Boolean value. To choose all items, it can repeat a `True` value. To pick a fixed subset, it can cycle among a `True` value followed by copies of a `False` value. To pick 1/4 of the items, we could use `cycle([True] + 3*[False])`.

The list comprehension can be revised as `compress(some_source, selectors)`, using a function for the `selectors` argument value. If we make that change, the processing is simplified:

```
>>> import random
>>> random.seed(1)
>>> data = [random.randint(1, 12) for _ in range(12)]

>>> from itertools import compress

>>> copy = compress(data, all_rows())
>>> list(copy)
[3, 10, 2, 5, 2, 8, 8, 8, 11, 7, 4, 2]

>>> cycle_subset = compress(data, subset(3))
>>> list(cycle_subset)
[3, 5, 8, 7]

>>> random.seed(1)
>>> random_subset = compress(data, randomized(3))
>>> list(random_subset)
[3, 2, 2, 4, 2]
```

These examples rely on the alternative selection rules `all_rows()`, `subset()`, and `randomized()`, as shown previously. The `subset()` and `randomized()` functions must be defined with a proper parameter with the value for c to pick $\frac{1}{c}$ of the rows from the source. The `selectors` expression must build an iterable over `True` and `False` values based on one of the selection rule functions. The rows to be kept are selected by applying the `source` iterable to the row-selection iterable.

Since all of this is done as a lazy evaluation, rows are not read from the source until required. This allows us to process very large sets of data efficiently. Also, the relative simplicity of the Python code means that we don't really need a complex configuration file and an associated parser to make choices among the selection rules. We have the option to use this bit of Python code as the configuration for a larger data-sampling application.

We can think of the `filter()` function as having the following definition:

```
from itertools import compress, tee
from collections.abc import Iterable, Iterator, Callable
from typing import TypeVar

SrcT = TypeVar("SrcT")

def filter_concept(
        function: Callable[[SrcT], bool],
        source: Iterable[SrcT]
) -> Iterator[SrcT]:
    i1, i2 = tee(source, 2)
    return compress(i1, map(function, i2))
```

We cloned the iterable using the `tee()` function. We'll look at this function in detail later. The `map()` function will generate results of applying the filter predicate function, `function()`, to each value in the iterable, yielding a sequence of `True` and `False` values. The sequence of Booleans is used to compress the original sequence, passing only items associated with `True`. This builds the features of the `filter()` function from the `compress()` function.

The function's hint can be broadened to `Callable[[SrcT], Any]`. This is because the `compress()` function will make use of the truthiness or falsiness of the values returned. It seems helpful to emphasize that the values will be understood as Booleans, hence the use of `bool` in the type hint, not `Any`.

Picking subsets with islice()

In *Chapter 4, Working with Collections*, we looked at slice notation to select subsets from a collection. Our example was to pair up items sliced from a list object. The following is a simple list:

```
>>> from Chapter04.ch04_ex5 import parse_g

>>> with open("1000.txt") as source:
...     flat = list(parse_g(source))

>>> flat[:10]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]

>>> flat[-10:]
[7841, 7853, 7867, 7873, 7877, 7879, 7883, 7901, 7907, 7919]
```

We can create pairs using list slices as follows:

```
>>> list(zip(flat[0::2], flat[1::2]))
[(2, 3), (5, 7), (11, 13), ...]
```

The `islice()` function gives us similar capabilities without the overhead of materializing a list object. This will work with an iterable of any size. The `islice()` function accepts an `Iterable` source, and the three parameters that define a slice: the start, stop, and step values. This means `islice(source, 1, None, 2)` is similar to `source[1::2]`. Instead of the slice-like shorthand using `:`, optional parameter values are used; the rules match the built-in `range()` function. The important difference is that `source[1::2]` only works for a `Sequence` object like a list or tuple. The `islice(source, 1, None, 2)` function works for any iterable, including an iterator object, or a generator expression.

The following example will create pairs of values of an iterable using the `islice()` function:

```
>>> flat_iter_1 = iter(flat)
>>> flat_iter_2 = iter(flat)
>>> pairs = list(zip(
...     islice(flat_iter_1, 0, None, 2),
...     islice(flat_iter_2, 1, None, 2)
... ))
>>> len(pairs)
```

```
500
>>> pairs[:3]
[(2, 3), (5, 7), (11, 13)]
>>> pairs[-3:]
[(7877, 7879), (7883, 7901), (7907, 7919)]
```

We created two independent iterators over a collection of data points in the `flat` variable. These could be two separate iterators over an open file or a database result set. The two iterators need to be independent to ensure a change in one `islice()` source doesn't interfere with the other `islice()` source.

This will produce a sequence of two-tuples from the original sequence:

```
[(2, 3), (5, 7), (11, 13), (17, 19), (23, 29),
...
(7883, 7901), (7907, 7919)]
```

Since `islice()` works with an iterable, this kind of design can work with extremely large sets of data. We can use this to pick a subset out of a larger set of data. In addition to using the `filter()` or `compress()` functions, we can also use the `islice(source, 0, None, c)` method to pick a $\frac{1}{c}$-sized subset from a larger set of data.

Stateful filtering with dropwhile() and takewhile()

The `dropwhile()` and `takewhile()` functions are stateful filter functions. They start in one mode; the given predicate function is a kind of flip-flop that switches the mode. The `dropwhile()` function starts in reject mode; when the function becomes `False`, it switches to pass mode. The `takewhile()` function starts in pass mode; when the given function becomes `False`, it switches to reject mode. Since these are filters, they will consume the entire iterable argument value.

We can use these to skip header or footer lines in an input file. We use the `dropwhile()` function to reject header rows and pass the remaining data. We use the `takewhile()`

function to pass data and reject trailer rows. We'll return to the simple GPL file format shown in *Chapter 3, Functions, Iterators, and Generators*. The file has a header that looks as follows:

```
GIMP Palette
Name: Crayola
Columns: 16
#
```

This is followed by rows that look like the following example data:

```
255 73 108 Radical Red
```

Note that there's an invisible tab character, `\t`, between the RGB color triple and the color name. To make it more visible, we can typeset the example like this:

```
255 73 108\tRadical Red
```

This little typesetting technique seems a little misleading, since it doesn't look like that in most programming editors.

We can locate the final line of the headers—the `#` line—using a parser based on the `dropwhile()` function, as follows:

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

>>> source_path = Path("crayola.gpl")
>>> with source_path.open() as source:
...     rdr = csv.reader(source, delimiter='\\t')
...     row_iter = dropwhile(
...         lambda row: row[0] != '#', rdr
...     )
...     color_rows = islice(row_iter, 1, None)
```

```
...     colors = list(
...         (color.split(), name) for color, name in color_rows
...     )
```

We created a CSV reader to parse the lines based on tab characters. This will neatly separate the `color` three-tuple from the name. The three-tuple will need further parsing. This will produce an iterator that starts with the # line and continues with the rest of the file.

We can use the `islice()` function to discard the first item of an iterable. The `islice(rows, 1, None)` expression is similar to asking for a `rows[1:]` slice: the first item is quietly discarded. Once the last of the heading rows have been discarded, we can parse the color tuples and return more useful color objects.

For this particular file, we can also use the number of columns located by the CSV `reader()` function. Header rows only have a single column, allowing the use of the `dropwhile(lambda row: len(row) == 1, rdr)` expression to discard header rows. This isn't a good approach in general, because locating the last line of the headers is often easier than trying to define some general pattern that distinguishes all header (or trailer) lines from the meaningful file content. In this case, the header rows were distinguishable by the number of columns; this is a rarity.

Two approaches to filtering with filterfalse() and filter()

In *Chapter 5*, *Higher-Order Functions*, we looked at the built-in `filter()` function. The `filterfalse()` function from the `itertools` module could be defined from the `filter()` function, as follows:

```
filterfalse_concept = (
    lambda pred, iterable:
    filter(lambda x: not pred(x), iterable)
)
```

As with the `filter()` function, the predicate function can be the `None` value. The value of

the `filter(None, iterable)` method is all the `True` values in the iterable. The value of the `filterfalse(None, iterable)` method is all the `False` values from the iterable:

```
>>> from itertools import filterfalse

>>> source = [0, False, 1, 2]
>>> list(filter(None, source))
[1, 2]

>>> filterfalse(None, source)
<itertools.filterfalse object at ...>
>>> list(_)
[0, False]
```

The point of having the `filterfalse()` function is to promote reuse. If we have a succinct function that makes a filter decision, we should be able to use that function to partition input to pass as well as reject groups without having to fiddle around with logical negation.

The idea is to execute the following commands:

```
>>> iter_1, iter_2 = tee(iter(raw_samples), 2)

>>> rule_subset_iter = filter(rule, iter_1)
>>> not_rule_subset_iter = filterfalse(rule, iter_2)
```

This kind of processing into two subsets will include all items from the source. The `rule()` function is unchanged, and we can't introduce a subtle logic bug through improper negation of this function.

Applying a function to data via starmap() and map()

The built-in `map()` function is a higher-order function that applies a function to items from an iterable. We can think of the simple version of the `map()` function as follows:

```
map_concept = (
    lambda function, arg_iter:
    (function(a) for a in arg_iter)
)
```

This works well when the `arg_iter` parameter is an iterable that provides individual values. The actual `map()` function is quite a bit more sophisticated than this, and can also work with a number of iterables.

The `starmap()` function in the `itertools` module is the `*args` version of the `map()` function. We can imagine the definition as follows:

```
starmap_concept = (
    lambda function, arg_iter:
    (function(*a) for a in arg_iter)
             #^-- Adds this * to decompose tuples
)
```

This reflects a small shift in the semantics of the `map()` function to properly handle an iterable-of-tuples structure. Each tuple is decomposed and applied to the various positional parameters.

When we look at the trip data, from the preceding commands, we can redefine the construction of a `Leg` object based on the `starmap()` function.

We could use the `starmap()` function to assemble the `Leg` objects, as follows:

```
from Chapter04.ch04_ex1 import legs, haversine
from Chapter06.ch06_ex3 import row_iter_kml
from Chapter07.ch07_ex1 import float_lat_lon, LegNT, PointNT
import urllib.request
from collections.abc import Callable

def get_trip_starmap(url: str) -> List[LegNT]:
    make_leg: Callable[[PointNT, PointNT], LegNT] = (
```

```
        lambda start, end:
        LegNT(start, end, haversine(start, end))
    )
    with urllib.request.urlopen(url) as source:
        path_iter = float_lat_lon(
            row_iter_kml(source)
        )
        pair_iter = legs(path_iter)
        trip = list(starmap(make_leg, pair_iter))
                    #-------- Used here
    return trip
```

Here's how it looks when we apply this `get_trip_starmap()` function to read source data and iterate over the created `Leg` objects:

```
>>> from pprint import pprint
>>> source_url = "file:./Winter%202012-2013.kml"
>>> trip = get_trip_starmap(source_url)
>>> len(trip)
73
>>> pprint(trip[0])
LegNT(start=PointNT(latitude=37.54901619777347,
longitude=-76.33029518659048), end=PointNT(latitude=37.840832,
longitude=-76.273834), distance=17.724564798884984)

>>> pprint(trip[-1])
LegNT(start=PointNT(latitude=38.330166, longitude=-76.458504),
end=PointNT(latitude=38.976334, longitude=-76.473503),
distance=38.801864781785845)
```

The `make_leg()` function accepts a pair of `Point` objects, and returns a `Leg` object with the start point, end point, and distance between the two points. The `legs()` function from *Chapter 4, Working with Collections*, creates pairs of `Point` objects that reflect the start and end of a leg of the voyage. The pairs created by `legs()` are provided as input to `make_leg()` to create proper `Leg` objects.

The `map()` function can also accept multiple iterables. When we use `map(f, iter1, iter2, ...)`, it behaves as if the iterators are zipped together, and the `starmap()` function is applied.

We can think of the `map(function, iter1, iter2, iter3)` function as if it were `starmap(function, zip(iter1, iter2, iter3))`.

The benefit of the `starmap(function, some_list)` method is to replace a potentially wordy `(function(*args) for args in some_list)` generator expression with something that avoids the potentially overlooked `*` operator applied to the function argument values.

Cloning iterators with tee()

The `tee()` function gives us a way to circumvent one of the important Python rules for working with iterables. The rule is so important, we'll repeat it here:

Iterators can be used only once.

The `tee()` function allows us to clone an iterator. This seems to free us from having to materialize a sequence so that we can make multiple passes over the data. Because `tee()` can use a lot of memory, it is sometimes better to materialize a list and process it multiple times, rather than trying to use the potential simplification of the `tee()` function.

For example, a simple average for an immense dataset could be written in the following way:

```
from collections.abc import Iterable

def mean_t(source: Iterable[float]) -> float:
    it_0, it_1 = tee(iter(source), 2)
    N = sum(1 for x in it_0)
    sum_x = sum(x for x in it_1)
    return sum_x/N
```

This would compute an average without appearing to materialize the entire dataset in

memory. Note that the type hint of `float` doesn't preclude integers. The **mypy** program is aware of the numeric processing rules, and this definition provides a flexible way to specify that either `int` or `float` will work.

The itertools recipes

Within the *itertools* chapter of the Python library documentation, there's a subsection called *Itertools Recipes*, which contains outstanding examples of ways to use the various `itertools` functions. Since there's no reason to reproduce these, we'll reference them here. They should be considered as required reading on functional programming in Python.

For more information, visit
`https://docs.python.org/3/library/itertools.html#itertools-recipes`.

It's important to note that these aren't importable functions in the `itertools` modules. A recipe needs to be read and understood and then, perhaps, copied or modified before it's included in an application.

Some of the recipes involve some of the more advanced techniques shown in the next chapter; they're not in the following table. We've preserved the ordering of items in the Python documentation, which is not alphabetical. The following table summarizes some of the recipes that show functional programming design patterns built from the `itertools` basics:

Function Name	**Arguments**	**Results**
`take`	`(n, iterable)`	Yields the first *n* items of the iterable as a list. This wraps a use of `islice()` in a simple name.
`tabulate`	`(function, start=0)`	Yields `function(0)`, `function(1)`, and so on. This is based on a `map(function, count())`.

Function Name	**Arguments**	**Results**
`consume`	`(iterator, n)`	Advance the iterator *n* steps ahead. If *n* is `None`, it consumes all of the values from the iterator.
`nth`	`(iterable, n, default=None)`	Return only the *n*th item or a default value. This wraps the use of `islice()` in a simple name.
`quantify`	`(iterable, pred=bool)`	Returns the count of how many times the predicate is true. This uses `sum()` and `map()` and relies on the way a Boolean predicate is effectively 1 when converted to an integer value.
`padnone`	`(iterable)`	Yields the iterable's elements and then yields `None` indefinitely. This can create functions that behave like `zip_longest()` or `map()`.
`ncycles`	`(iterable, n)`	Yields the sequence elements *n* times.
`dotproduct`	`(vec1, vec2)`	A dot product multiplies two vector's values and finds the sum of the result.
`flatten`	`(listOfLists)`	This function flattens one level of nesting. This chains the various lists together into a single list.
`repeatfunc`	`(func, times=None, *args)`	This calls the given function, `func`, repeatedly with specified arguments.
`grouper`	`(iterable, n, fillvalue=None)`	Yields the iterable's elements as a sequence of fixed-length chunks or blocks.

Function Name	Arguments	Results
`roundrobin`	`(*iterables)`	Yields values taken from each of the iterables. For example, `roundrobin('ABC', 'D', 'EF')` is `'A', 'D', 'E', 'B', 'F', 'C'`.
`partition`	`(pred, iterable)`	This uses a predicate to partition entries into `False` entries and `True` entries. The return value is a pair of iterators.
`unique_everseen`	`(iterable, key=None)`	Yields the unique elements of the source iterable, preserving order. It also remembers all elements ever seen.
`unique_justseen`	`(iterable, key=None)`	Yields unique elements, preserving order. It remembers only the element most recently seen. This is useful for deduplicating or grouping a sorted sequence.
`iter_except`	`(func, exception, first=None)`	Yields results of calling a function repeatedly until an exception is raised. The exception is silenced. This can be used to iterate until `KeyError` or `IndexError`.

Summary

In this chapter, we've looked at a number of functions in the `itertools` module. This library module helps us to work with iterators in sophisticated ways.

We've looked at the infinite iterators; they repeat without terminating. They include the `count()`, `cycle()`, and `repeat()` functions. Since they don't terminate, the consuming function must determine when to stop accepting values.

We've also looked at a number of finite iterators. Some of them are built-in, and some of them are a part of the `itertools` module. They work with a source iterable, so they terminate when that iterable is exhausted. These functions include `enumerate()`, `accumulate()`, `chain()`, `groupby()`, `zip_longest()`, `zip()`, `pairwise()`, `compress()`, `islice()`, `dropwhile()`, `takewhile()`, `filterfalse()`, `filter()`, `starmap()`, and `map()`. These functions allow us to replace possibly complex generator expressions with simpler-looking functions.

We've noted that functions like the `tee()` function are available, and can create a helpful simplification. It has the potential cost of using a great deal of memory, and needs to be considered carefully. In some cases, materializing a list may be more efficient than applying the `tee()` function.

Additionally, we looked at the recipes from the documentation, which provide yet more functions we can study and copy for our own applications. The recipes list shows a wealth of common design patterns.

In *Chapter 9*, *Itertools for Combinatorics – Permutations and Combinations*, we'll continue our study of the `itertools` module, focusing on permutations and combinations. These operations can produce voluminous results. For example, enumerating all possible 5-card hands from a deck of 52 cards will yield over 3.12×10^8 permutations. For small domains, however, it can be helpful to examine all possible orderings to understand how well observed samples match the domain of possible values.

Exercises

This chapter's exercises are based on code available from Packt Publishing on GitHub. See `https://github.com/PacktPublishing/Functional-Python-Programming-3rd-Edition`.

In some cases, the reader will notice that the code provided on GitHub includes partial solutions to some of the exercises. They serve as hints, allowing the reader to explore alternative solutions.

In many cases, exercises will need unit test cases to confirm they actually solve the problem. These are often identical to the unit test cases already provided in the GitHub repository. The reader should replace the book's example function name with their own solution to confirm that it works.

Optimize the find_first() function

In *Counting with float arguments*, we defined a `find_first()` function to locate the first pair of an iterator that passed a given test criteria. In most of the examples, the test was a comparison between values to see if the difference between the values was larger than 10^{-12}.

The definition of the `find_first()` function used a simpler recursion. This limits the size of the iterable that can be examined: only about 1,000 values can be consumed before hitting the stack size limitation.

First, create a comparison function that will consume enough values to fail with a recursion limit exception.

Then, rewrite the `find_first()` function to replace the tail call with iteration using the `for` statement.

Using the comparison function found earlier, demonstrate that the revised function will readily pass 1,000 elements, looking for the first that matches the revised criteria.

Compare Chapter 4 with the itertools.pairwise() recipe

In *Chapter 4, Working with Collections*, the `legs()` function created overlapping pairs from a source iterable. Compare the implementation provided in this book with the `pairwise()` function.

Create a very, very large iterable and compare the performance of the `legs()` function and

the `pairwise()` function. Which is faster?

Compare Chapter 4 with itertools.tee() recipe

In the *Using sums and counts for statistics* section of *Chapter 4, Working with Collections*, a `mean()` function was defined that had a limitation of only working with sequences. If `itertools.tee()` is used, a `mean()` function can be written that will apply to iterators in general, without being limited to collection objects that can produce multiple iterators. Define a `mean_i()` function based on the `itertools.tee()` function that works with any iterator. Which variant of mean computations is easier to understand?

Create a very, very large iterable and compare the performance of the `mean_i()` function and the `mean()` function shown in the text. Which is faster? It takes some time to explore, but locating a collection that breaks the `itertools.tee()` function while still working with a materialized list object is an interesting thing to find.

Splitting a dataset for training and testing purposes

Given a pool of samples, it's sometimes necessary to partition the data into a subset used for building (or "training") a model, and a separate subset used to test the model's predictive ability. It's common practice to use subsets of 20%, 25%, or even 33% of the source data for testing. Develop a set of functions to partition the data into subsets with ratios of `1 : 3`, `1 : 4`, or `1 : 5` for test *vs.* training data.

Rank ordering

In *Chapter 7, Complex Stateless Objects*, we looked at ranking items in a set of data. The approach shown in that chapter was to build a dictionary of items with the same key value. This made it possible to create a rank that was the mean of the various items. For example, the sequence `[0.8, 1.2, 1.2, 2.3, 18]` should have rank values of 1, 2.5, 2.5, 4, 5. The two matching key values in positions 1 and 2 of the sequence should have the midpoint value of 2.5 as their common rank.

This can be computed using `itertools.groupby()`. Each group will have some number of members, provided by the `groupby()` function. The sequence of rank values for a group

of n items with matching keys is $r_0, r_0 + 1, r_0 + 2, ..., r_0 + n$. The value of r_0 is the starting rank for the group. The mean of this sequence is $r_0 + \frac{n}{2}$. This processing requires creating a temporary sequence of values in order to emit each item from the group of values with the same key with their matching ranks.

Write this `rank()` function, using the `itertools.groupby()` function. Compare the code with the examples in *Chapter 7, Complex Stateless Objects*. What advantages does the `itertools` variant offer?

9

Itertools for Combinatorics – Permutations and Combinations

Functional programming emphasizes stateless algorithms. In Python, this leads us to work with generator expressions, generator functions, and iterables. In this chapter, we'll continue our study of the `itertools` library, with numerous functions to help us work with iterable collections.

In the previous chapter, we looked at three broad groupings of iterator functions. They are as follows:

- Functions that work with infinite iterators, which can be applied to any iterable or an iterator over any collection; they consume the entire source
- Functions that work with finite iterators, which can either accumulate a source multiple times, or produce a reduction of the source

- The `tee()` iterator function, which clones an iterator into several copies that can each be used independently

In this chapter, we'll look at the `itertools` functions that work with permutations and combinations. These include several combinatorial functions and a few recipes built on these functions. The functions are as follows:

- `product()`: This function forms a Cartesian product equivalent to the nested `for` statements or nested generator expressions.
- `permutations()`: This function yields tuples of length r from a universe p in all possible orderings; there are no repeated elements.
- `combinations()`: This function yields tuples of length r from a universe p in sorted order; there are no repeated elements.
- `combinations_with_replacement()`: This function yields tuples of length r from p in sorted order, allowing the repetition of elements.

These functions embody algorithms that can create potentially large result sets from small collections of input data. Some kinds of problems have exact solutions based on exhaustively enumerating the universe of permutations. As a trivial example, when trying to find out if a hand of cards contains a *straight* (all numbers in adjacent, ascending order), one solution is to compute all permutations and see if at least one arrangement of cards is ascending. For a 5-card hard, there are only 120 arrangements. These functions make it simple to yield all the permutations; in some cases, this kind of simple enumeration isn't optimal or even desirable.

Enumerating the Cartesian product

The term **Cartesian product** refers to the idea of enumerating all the possible combinations of elements drawn from the elements of sets.

Mathematically, we might say that the product of two sets, $\{1, 2, 3, \ldots, 13\} \times \{\clubsuit, \blacklozenge, \heartsuit, \spadesuit\}$,

has 52 pairs, as follows:

$$\{(1, ♣), (1, ♦), (1, ♥), (1, ♠), (2, ♣), (2, ♦), (2, ♥), (2, ♠), ..., (13, ♣), (13, ♦), (13, ♥), (13, ♠)\}$$

We can produce the preceding results by executing the following commands:

```
>>> cards = list(product(range(1, 14), '♣♦♥♠'))
>>> cards[:4]
[(1, '♣'), (1, '♦'), (1, '♥'), (1, '♠')]
>>> cards[4:8]
[(2, '♣'), (2, '♦'), (2, '♥'), (2, '♠')]
>>> cards[-4:]
[(13, '♣'), (13, '♦'), (13, '♥'), (13, '♠')]
```

The calculation of a product can be extended to any number of iterable collections. Using a large number of collections can lead to a very large result set.

Reducing a product

In relational database theory, a join between tables can be thought of as a filtered product. For those who know SQL, the `SELECT` statement joining tables without a `WHERE` clause will produce a Cartesian product of rows in the tables. This can be thought of as the worst-case algorithm—a vast product without any useful filtering to pick the desired subset of results. We can implement this using the `itertools.product()` function to enumerate all possible combinations and filter those to keep the few that match properly.

We can define a `join()` function to join two iterable collections or generators, as shown in the following commands:

```
from collections.abc import Iterable, Iterator, Callable
from itertools import product
from typing import TypeVar

JTL = TypeVar("JTL")
```

```
JTR = TypeVar("JTR")

def join(
    t1: Iterable[JTL],
    t2: Iterable[JTR],
    where: Callable[[tuple[JTL, JTR]], bool]
) -> Iterable[tuple[JTL, JTR]]:
    return filter(where, product(t1, t2))
```

All combinations of the two iterables, `t1` and `t2`, are computed. The `filter()` function will apply the given `where()` function to pass or reject two-tuples, hinted as `tuple[JTL, JTR]`, that match properly. The `where()` function has the hint `Callable[[tuple[JTL, JTR]], bool]` to show that it returns a Boolean result. This is typical of how SQL database queries work in the worst-case situation where there are no useful indexes or cardinality statistics to suggest a better algorithm.

While this algorithm always works, it can be terribly inefficient. We often need to look carefully at the problem and the available data to find a more efficient algorithm.

First, we'll generalize the problem slightly by replacing the simple Boolean matching function. Instead of a binary result, it's common to look for a minimum or maximum of some distance between items. In this case, the comparison yields a float value.

Assume that we have a class to define instances in a table of `Color` objects, as follows:

```
from typing import NamedTuple
class Color(NamedTuple):
    rgb: tuple[int, int, int]
    name: str
```

Here's an example of using this definition to create some `Color` instances:

```
>>> palette = [Color(rgb=(239, 222, 205), name='Almond'),
...  Color(rgb=(255, 255, 153), name='Canary'),
...  Color(rgb=(28, 172, 120), name='Green'),
...  Color(rgb=(255, 174, 66), name='Yellow Orange')
... ]
```

For more information, see *Chapter 6, Recursions and Reductions*, where we showed you how to parse a file of colors to create `NamedTuple` objects. In this case, we've left the RGB color as a `tuple[int, int, int]`, instead of decomposing each individual field.

An image will have a collection of pixels, each of which is an RGB tuple. Conceptually, an image contains data like this:

```
pixels = [(r, g, b), (r, g, b), (r, g, b), ...]
```

As a practical matter, the **Python Imaging Library** (**PIL**) package presents the pixels in a number of forms. One of these is the mapping from the (x, y) coordinate to the RGB triple. For the Pillow project documentation, visit `https://pypi.python.org/pypi/Pillow`.

Given a `PIL.Image` object, we can iterate over the collection of pixels with something like the following commands:

```
from collections.abc import Iterator
from typing import TypeAlias
from PIL import Image  # type: ignore[import]

Point: TypeAlias = tuple[int, int]
RGB: TypeAlias = tuple[int, int, int]
Pixel: TypeAlias = tuple[Point, RGB]

def pixel_iter(img: Image) -> Iterator[Pixel]:
    w, h = img.size
    return (
```

```
        (c, img.getpixel(c))
        for c in product(range(w), range(h))
    )
```

This function determines the range of each coordinate based on the image size, `img.size`. The values of the `product(range(w), range(h))` method create all the possible combinations of coordinates. It has a result identical to two nested `for` statements in a single expression.

This has the advantage of enumerating each pixel with its coordinates. We can then process the pixels in no particular order and still reconstruct an image. This is particularly handy when using multiprocessing or multithreading to spread the workload among several cores or processors. The `concurrent.futures` module provides an easy way to distribute work among cores or processors.

Computing distances

A number of decision-making problems require that we find a close enough match. We might not be able to use a simple equality test. Instead, we have to use a distance metric and locate items with the shortest distance to our target. The ***k*-Nearest Neighbors (k-NN)** algorithm, for example, uses a training set of data and a distance measurement function. It locates the k nearest neighbors to an unknown sample, and uses the majority of those neighbors to classify the unknown sample.

To explore this concept of enumerating all possible matches, we'll use a slightly simpler example. However, even though it's superficially simpler, it doesn't work out well if we approach it naively and exhaustively enumerate all potential matches.

When doing color matching, we won't have a simple equality test. A color, C, for our purposes, is a triple $\langle r, g, b \rangle$. It's a point in three-dimensional space. It's often sensible to define a minimal distance function to determine whether two colors are close enough, without having the same three values of $\langle r_1, g_1, b_1 \rangle = \langle r_2, g_2, b_2 \rangle$. We need a multi-dimensional distance computation, using the red, green, and blue axes of a color space. There are several

common approaches to measuring distance, including the Euclidean distance, Manhattan distance, and other more complex weightings based on visual preferences.

Here are the Euclidean and Manhattan distance functions:

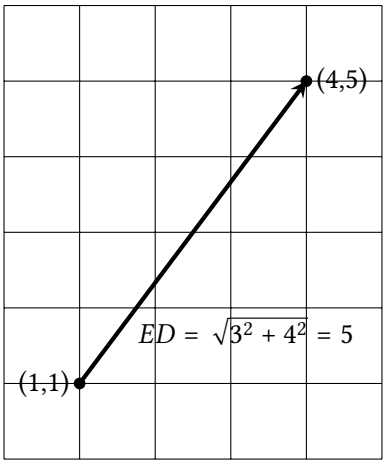
```
import math
def euclidean(pixel: RGB, color: Color) -> float:
    return math.sqrt(
        sum(map(
            lambda x_1, x_2: (x_1 - x_2) ** 2,
            pixel,
            color.rgb))
        )

def manhattan(pixel: RGB, color: Color) -> float:
    return sum(map(
        lambda x_1, x_2: abs(x_1 - x_2),
        pixel,
        color.rgb))
```

The **Euclidean distance** measures the hypotenuse of a right-angled triangle among the three points in an RGB space. Here's the formal definition for three dimensions:

$$ED(c_1, c_2) = \sqrt{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2}$$

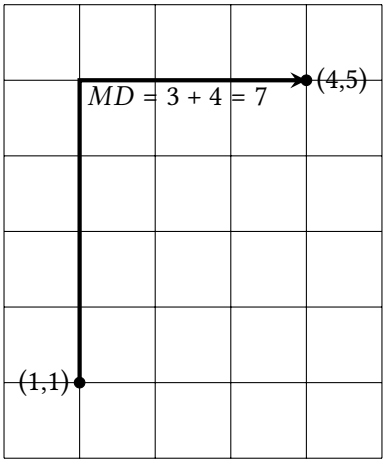
Here's a two-dimensional sketch of the Euclidean distance between two points:



The **Manhattan distance** sums the edges of each leg of the right-angled triangle among the three points. It's named after the gridded layout of the Borough of Manhattan in New York City. To get around, one is forced to travel only on the streets and avenues. Here's the formal definition for three dimensions:

$$MD(c_1, c_2) = (r_1 - r_2) + (g_1 - g_2) + (b_1 - b_2)$$

Here's a two-dimensional sketch of the Manhattan distance between two points:



The Euclidean distance offers precision, while the Manhattan distance offers calculation

speed.

Looking forward, we're aiming for a structure that looks like this. For each individual pixel, we can compute the distance from that pixel's color to the available colors in a limited color set. The results of this calculation for one individual pixel of an image might start like the following example:

```
[((0, 0),
  (92, 139, 195),
  Color(rgb=(239, 222, 205), name='Almond'),
  169.10943202553784),
 ((0, 0),
  (92, 139, 195),
  Color(rgb=(255, 255, 153), name='Canary'),
  204.42357985320578),
 ((0, 0),
  (92, 139, 195),
  Color(rgb=(28, 172, 120), name='Green'),
  103.97114984456024),
 ((0, 0),
  (92, 139, 195),
  Color(rgb=(48, 186, 143), name='Mountain Meadow'),
  82.75868534480233),
```

We've shown a sequence of tuples; each tuple has four items:

- The pixel's coordinates; for example, (0,0)
- The pixel's original color; for example, (92, 139, 195)
- A `Color` object from a set of seven colors; for example, `Color(rgb=(48, 186, 143), name='Mountain Meadow')`
- The Euclidean distance between the original color and the given `Color` object; for example, 82.75868534480233

It can help to create a `NamedTuple` to encapsulate the four items in each tuple. We could call it an X-Y, Pixel, Color, Distance tuple, something like "XYPCD." This would make it slightly easier to identify the (x, y) coordinate, the original pixel's color, the matching color,

and the distance between the original color and the selected match.

The smallest Euclidean distance is a closest match color. For the four example colors, the Mountain Meadow is the closest match for this pixel. This kind of reduction is done with the `min()` function. If the overall four-tuple of (x,y), pixel, color, and distance is assigned to a variable name, `choices`, the pixel-level reduction would look like this:

```
min(choices, key=lambda xypcd: xypcd[3])
```

This expression will pick a single tuple as the optimal match between a pixel and color. It uses a lambda to select item 3 from the tuple, the distance metric.

Getting all pixels and all colors

How do we get to the structure that contains all pixels and all colors? One seemingly simple answer turns out to be less than optimal.

One way to map pixels to colors is to enumerate all pixels and all colors using the `product()` function:

```
from collections.abc import Iterable
from itertools import groupby

def matching_1(
        pixels: Iterable[Pixel],
        colors: Iterable[Color]
    ) -> Iterator[tuple[Point, RGB, Color, float]]:

    distances = (
        (pixel[0], pixel[1], color, euclidean(pixel[1], color))
        for pixel, color in product(pixels, colors)
    )
    for _, choices in groupby(distances, key=lambda xy_p_c_d:
    xy_p_c_d[0]):
        yield min(choices, key=lambda xypcd: xypcd[3])
```

The core of this is the `product(pixel_iter(img), colors)` expression that creates a

sequence of all pixels combined with all colors. The overall expression then applies the `euclidean()` function to compute distances between pixels and `Color` objects. The result is a sequence of four-tuple objects with the original (x, y) coordinate, the original pixel, an available color, and the distance between the original pixel color and the available color.

The final selection of colors uses the `groupby()` function and the `min(choices, ...)` expression to locate the closest match.

The `product()` function applied to pixels and colors creates a long, flat iterable. We grouped the iterable into smaller collections where the coordinates match. This will break the big iterable into smaller iterables of only the pool of colors associated with a single pixel. We can then pick the minimal color distance for each available color for a pixel.

In a picture that's 3,648×2,736 pixels with 133 Crayola colors, we have an iterable with $3,648 \times 2,736 \times 133 = 1,327,463,424$ items to be evaluated. That is a billion combinations created by this `distances` expression. The number is not necessarily impractical; it's well within the limits of what Python can do. However, it reveals an important flaw in the naive use of the `product()` function.

We can't trivially do this kind of large-scale processing without first doing some analysis to see how large the intermediate data will be. Here are some `timeit` numbers for these two distance functions. This is the overall number of seconds to do each of these calculations only 1,000,000 times:

- Euclidean: 1.761
- Manhattan: 0.857

Scaling up by a factor of 1,000—from 1 million combinations to 1 billion—means the processing will take at least 1,800 seconds; that is, about half an hour for the Manhattan distance and 46 minutes to calculate the Euclidean distance. It appears this kind of naive bulk processing is ineffective for large datasets.

More importantly, we're doing it wrong. This kind of *width* × *height* × *color* processing is simply a bad design. In many cases, we can do much better.

Performance improvements

A key feature of any big data algorithm is locating a way to execute some kind of a divide-and-conquer strategy. This is true of functional programming design as well as imperative design.

Here are three options to speed up this processing:

- We can try to use parallelism to do more of the calculations concurrently. On a four-core processor, the time can be cut to approximately 25 percent. This reduces the time to 8 minutes for Manhattan distances.
- We can see if caching intermediate results will reduce the amount of redundant calculation. The question arises of how many colors are the same and how many colors are unique.
- We can look for a radical change in the algorithm.

We'll combine the last two points by computing all the possible comparisons between source colors and target colors. In this case, as in many other contexts, we can easily enumerate the entire mapping of pixels and colors. If colors are repeated, we avoid doing redundant calculations to locate the closest color. We'll also change the algorithm from a series of comparisons to a series of lookups in a mapping object.

In many problem domains, the source data is a collection of floating-point values. While these `float` values are flexible, and correspond in some ways with the mathematical abstraction of real numbers, they introduce some additional costs. Float operations can be slower than integer operations. More importantly, float values can contain a number of "noise" bits. For example, common RGB color definitions use 256 distinct values for each of the Red, Green, and Blue components. These values are represented exactly with 8 bits. A floating-point variant, using values from 0.0 to 1.0, would use the full 64 bits for each color. Any arithmetic that led to floating-point truncation would introduce noise. While `float` values seem simple, they introduce troubling problems.

Here's an example, using a red `r=15`:

```
>>> r = 15
>>> r_f = 15/256
>>> r_f
0.05859375
>>> r_f + 1/100 - 1/100
0.05859374999999999
```

Algebraically, $r_f + \frac{1}{100} - \frac{1}{100} = r_f$. However, the `float` definition is only an approximation of the abstract concept of a real number. The value of $\frac{1}{100}$ doesn't have an exact representation in binary-based floating-point. Using a value like this introduces truncation errors that propagate through subsequent computations. We've chosen to use integer-based color matching to show a way to minimize the additional complications that can arise from floating-point values.

When looking at this idea of pre-computing all transformations from source color to target color, we need some overall statistics for an arbitrary image. The code associated with this book includes `IMG_2705.jpg`. Here is a basic algorithm to collect all of the distinct color tuples from the specified image:

```
from collections import defaultdict, Counter
def gather_colors() -> defaultdict[RGB, list[Point]]:
    img = Image.open("IMG_2705.jpg")

    palette = defaultdict(list)
    for xy, rgb in pixel_iter(img):
        palette[rgb].append(xy)

    w, h = img.size
    print(f"total pixels {w*h}")
    print(f"total colors {len(palette)}")
    return palette
```

We collected all pixels of a given color into a list organized by color. From this, we'll learn the following facts:

- The total number of pixels is 9,980,928. This fits the expectation for a 10-megapixel image.
- The total number of colors is 210,303. If we try to compute the Euclidean distance between actual colors and the 133 target colors, we would do 27,970,299 calculations, which might take about 76 seconds.
- If we use a less accurate representation, one with fewer bits, we can speed things up. We'll call this "masking" to remove some of the irrelevant least-significant bits. Using a 3-bit mask, `0b11100000`, the total number of colors actually used is reduced to 214 out of a domain of $2^3 \times 2^3 \times 2^3 = 512$ possible colors.
- Using a 4-bit mask, `0b11110000`, 1,150 colors are actually used.
- Using a 5-bit mask, `0b11111000`, 5,845 colors are actually used.
- Using a 6-bit mask, `0b11111100`, 27,726 colors are actually used. The domain of possible colors swells to $2^6 \times 2^6 \times 2^6 = 262,144$.

This gives us some insight into how we can rearrange the data structure, calculate the matching colors quickly, and then rebuild the image without doing a billion comparisons and avoiding any additional complications from floating-point approximations. There are a number of changes required to avoid needless (and error-introducing) computations.

The core idea behind masking is to preserve the most significant bits of a value and eliminate the least significant bits. Consider a color with a red value of 200. We can use the Python `bin()` function to see the binary representation of that value:

```
>>> bin(200)
'0b11001000'
>>> 200 & 0b11100000
192
>>> bin(192)
'0b11000000'
```

The computation of `200 & 0b11100000` applied a mask to conceal the least significant 5 bits and preserve the most significant 3 bits. What remains after the mask is applied as a red value of 192.

We can apply mask values to the RGB three-tuple with the following command:

```
masked_color = tuple(map(lambda x: x & 0b11100000, c))
```

This will pick out the most significant 3 bits of the red, green, and blue values of a color tuple by using the `&` operator to select particular bits from an integer value. If we use this masked value instead of the original color to create a `Counter` object, we'll see that the image only uses 214 distinct values after the mask is applied. This is fewer than half the theoretical number of colors.

Rearranging the problem

The naive use of the `product()` function to compare all pixels and all colors was a bad idea. There are 10 million pixels, but only 200,000 unique colors. When mapping the source colors to target colors, we only have to save 200,000 values in a simple map.

We'll approach it as follows:

1. Compute the source-to-target color mapping. In this case, let's use 3-bit color values as output. Each R, G, and B value comes from the eight values in the `range(0, 256, 32)` expression. We can use this expression to enumerate all the output colors:

   ```
   product(range(0, 256, 32), range(0, 256, 32), range(0, 256, 32))
   ```

2. We can then compute the Euclidean distance to the nearest color in our source palette, doing just 68,096 calculations. This takes about 0.14 seconds. It's done one time only and computes the 200,000 mappings.
3. In one pass through the source image, we build a new image using the revised color table. In some cases, we can exploit the truncation of integer values. We can use an expression such as `(0b11100000&r, 0b11100000&g, 0b11100000&b)` to remove the least significant bits of an image color. We'll look at this additional reduction in computation later.

This will replace a billion distance calculations with 10 million dictionary lookups, transforming a potential 30 minutes of calculation into about 30 seconds.

Given a source palette of approximately 200,000 colors, we can apply a fast Manhattan distance to locate the nearest color in a target palette, such as the Crayola colors.

We'll fold in yet another optimization—truncation. This will give us an even faster algorithm.

Combining two transformations

When combining multiple transformations, we can build a more complex mapping from the source through intermediate targets to the result. To illustrate this, we'll truncate the colors as well as applying a mapping.

In some problem contexts, truncation can be difficult. In other cases, it's often quite simple. For example, truncating US postal ZIP codes from nine to five characters is common. Postal codes can be further truncated to three characters to determine a regional facility that represents a larger geography.

For colors, we can use the bit-masking shown previously to truncate colors from three 8-bit values (24 bits, 16 million colors) to three 3-bit values (9 bits, 512 colors).

Here is a way to build a color map that combines distances to a given set of colors and truncation of the source colors:

```
from collections.abc import Sequence
def make_color_map(colors: Sequence[Color]) -> dict[RGB, Color]:
    bit3 = range(0, 256, 0b0010_0000)

    best_iter = (
        min((euclidean(rgb, c), rgb, c) for c in colors)
        for rgb in product(bit3, bit3, bit3)
    )
    color_map = dict((b[1], b[2]) for b in best_iter)
    return color_map
```

We created a range object, `bit3`, that will iterate through all eight of the 3-bit color values.

The use of the binary value, `0b0010_0000`, can help visualize the way the bits are being used. The least significant 5 bits will be ignored; only the upper 3 bits will be used.

The `range` objects aren't like ordinary iterators; they can be used multiple times. As a result of this, the `product(bit3, bit3, bit3)` expression will produce all 512 color combinations that we'll use as the output colors.

For each truncated RGB color, we created a three-tuple that has (0) the distance from all crayon colors, (1) the RGB color, and (2) the crayon `Color` object. When we ask for the minimum value of this collection, we'll get the closest crayon `Color` object to the truncated RGB color.

We built a dictionary that maps from the truncated RGB color to the closest crayon. In order to use this mapping, we'll truncate a source color before looking up the nearest crayon in the mapping. This use of truncation coupled with the pre-computed mapping shows how we might need to combine mapping techniques.

The following function will build a new image from a color map:

```
def clone_picture(
    color_map: dict[RGB, Color],
    filename: str = "IMG_2705.jpg"
) -> None:
    mask = 0b1110_0000
    img = Image.open(filename)
    clone = img.copy()
    for xy, rgb in pixel_iter(img):
        r, g, b = rgb
        repl = color_map[(mask & r, mask & g, mask & b)]
        clone.putpixel(xy, repl.rgb)
    clone.show()
```

This uses the PIL `putpixel()` function to replace all of the pixels in a picture with other pixels. The mask value preserves the upper-most three bits of each color, reducing the number of colors to a subset.

What we've seen is that the naive use of some functional programming tools can lead to algorithms that are expressive and succinct, but also inefficient. The essential tools to compute the complexity of a calculation (sometimes called **Big-O analysis**) is just as important for functional programming as it is for imperative programming.

The problem is not that the `product()` function is inefficient. The problem is that we can use the `product()` function to create an inefficient algorithm.

Permuting a collection of values

When we permute a collection of values, we'll generate all the possible orders for the values in the collection. There are $n!$ permutations of n items. We can use a sequence of permutations as a kind of brute-force solution to a variety of optimization problems.

Typical combinatorial optimization problems are the **Traveling Salesman problem**, the **Minimum Spanning Tree problem**, and the **Knapsack problem**. These problems are famous because they involve potentially vast numbers of permutations. Approximate solutions are necessary to avoid exhaustive enumeration of all permutations. The use of the `itertools.permutations()` function is only handy for exploring very small problems.

One popular example of these combinatorial optimization problems is the **assignment problem**. We have n agents and n tasks, but the cost of each agent performing a given task is not equal. Imagine that some agents have trouble with some details, while other agents excel at these details. If we can properly assign tasks to agents, we can minimize the costs.

We can create a simple grid that shows how well a given agent is able to perform a given task. For a small problem of seven agents and tasks, there will be a grid of 49 costs. Each cell in the grid shows agents A_0 to A_6 performing tasks T_0 to T_6:

	Agent						
Task	A_0	A_1	A_2	A_3	A_4	A_5	A_6
T_0	14	11	6	20	12	9	4
T_1	15	28	34	4	12	24	21
T_2	16	31	22	18	31	15	23
T_3	20	18	9	15	30	4	18
T_4	24	8	24	30	28	25	4
T_5	3	23	22	11	5	30	5
T_6	13	7	5	10	7	7	32

Given this grid, we can enumerate all the possible permutations of agents and their tasks. However, this approach doesn't scale well. For this problem, there are 720 alternatives. If we have more agents, for example 10, the value of 10! is 3,628,800. We can create the entire sequence of 3 million items with the `list(permutations(range(10)))` expression.

We would expect to solve a problem of this tiny size in a fraction of a second. For 10!, we might take a few seconds. When we double the size of the problem to 20!, we have a bit of a scalability problem: there will be 2.433×10^{18} permutations. On a computer where it takes about 0.56 seconds to generate 10! permutations, the process of generating 20! permutations would take about 12,000 years.

We can formulate the exhaustive search for the optimal solution as follows:

```
from itertools import permutations

def assignment(cost: list[tuple[int, ...]]) -> list[tuple[int, ...]]:
    n_tasks = len(cost)
    perms = permutations(range(n_tasks))
    alt = [
        (
            sum(
                cost[task][agent] for agent, task in enumerate(perm)
            ),
            perm
```

```
        )
        for perm in perms
    ]
    m = min(alt)[0]
    return [ans for s, ans in alt if s == m]
```

We've created all permutations of tasks for a group of agents and assigned this to `perms`. From this, we've created two-tuples of the sum of all costs in the cost matrix for a given permutation. To locate the relevant costs, a specific permutation is enumerated to create two-tuples showing the agent and the task assignment for that agent. For example, one of the permutations is tasks `(2, 4, 6, 1, 5, 3, 0)`. We can assign agent index values using the expression `list(enumerate((2, 4, 6, 1, 5, 3, 0)))`. The result, `[(0, 2), (1, 4), (2, 6), (3, 1), (4, 5), (5, 3), (6, 0)]`, has all seven agent index values and their associated task assignments. We can translate the index numbers to agent names and task names by incorporating a dictionary lookup. The sum of the values in the cost matrix tells us how expensive this specific task assignment would be.

One of the optimal solutions might look like the assignment above. It requires folding in the agent names and task names to translate the task permutation into a specific list of assignments:

A_0	T_2
A_1	T_4
A_2	T_6
A_3	T_1
A_4	T_5
A_5	T_3
A_6	T_0

In some cases, there might be multiple optimal solutions; this algorithm will locate all of them. The expression `min(alt)[0]` selects the first of the set of minima.

For small textbook examples, this seems to be reasonably fast. There are linear programming

approaches which avoid exhaustive enumeration of all permutations. The Python Linear Programming module PuLP can be used to solve the assignment problem. See `https://coin-or.github.io/pulp/`.

Generating all combinations

The `itertools` module also supports computing all combinations of a set of values. When looking at combinations, the order doesn't matter, so there are far fewer combinations than permutations. The number of combinations is often stated as $\binom{p}{r} = \frac{p!}{r!(p-r)!}$. This is the number of ways that we can take combinations of r things at a time from a universe of p items overall.

For example, there are 2,598,960 five-card poker hands. We can actually enumerate all 2 million hands by executing the following command:

```
>>> from itertools import combinations, product

>>> hands = list(
...     combinations(
...         tuple(
...             product(range(13), '♠♥♦♣')
...         ), 5
...     )
... )
```

More practically, assume we have a dataset with a number of variables. A common exploratory technique is to determine the correlation among all pairs of variables in a set of data. If there are v variables, then we will enumerate all variables that must be compared by executing the following command:

```
>>> combinations(range(v), 2)
```

A fun source of data for simple statistical analysis is the **Spurious Correlations** site. This has a great many datasets with surprising statistical properties. Let's get some sample data

from **Spurious Correlations**, `http://www.tylervigen.com`, to show how this will work. We'll pick three datasets with the same time range, datasets numbered 7, 43, and 3,890. We'll simply catenate the data into a grid. Because the source data repeats the `year` column, we'll start with data that includes the repeated `year` column. We'll eventually remove the obvious redundancy, but it's often best to start with all of the data present as a way to confirm that the various sources of data align with each other properly.

This is how the first and the remaining rows of the yearly data will look:

```
[('year', 'Per capita consumption of cheese (US)Pounds (USDA)',
'Number of people who died by becoming tangled in their
bedsheets Deaths (US) (CDC)',
'year', 'Per capita consumption of mozzarella cheese (US)Pounds
(USDA)', 'Civil engineering doctorates awarded (US) Degrees awarded
(National Science Foundation)',
'year', 'US crude oil imports from Venezuela Millions of barrels
(Dept. of Energy)', 'Per capita consumption of high fructose corn
syrup (US) Pounds (USDA)'),

(2000, 29.8, 327, 2000, 9.3, 480, 2000, 446, 62.6),
(2001, 30.1, 456, 2001, 9.7, 501, 2001, 471, 62.5),
(2002, 30.5, 509, 2002, 9.7, 540, 2002, 438, 62.8),
(2003, 30.6, 497, 2003, 9.7, 552, 2003, 436, 60.9),
(2004, 31.3, 596, 2004, 9.9, 547, 2004, 473, 59.8),
(2005, 31.7, 573, 2005, 10.2, 622, 2005, 449, 59.1),
(2006, 32.6, 661, 2006, 10.5, 655, 2006, 416, 58.2),
(2007, 33.1, 741, 2007, 11, 701, 2007, 420, 56.1),
(2008, 32.7, 809, 2008, 10.6, 712, 2008, 381, 53),
(2009, 32.8, 717, 2009, 10.6, 708, 2009, 352, 50.1)]
```

This is how we can use the `combinations()` function to yield all the combinations of the nine variables in this dataset, taken two at a time:

```
>>> combinations(range(9), 2)
```

There are 36 possible combinations. We'll have to reject the combinations that involve

matching columns `year` and `year`. These will trivially correlate with a value of 1.00.

Here is a function that picks a column of data out of our dataset:

```
from typing import TypeVar
from collections.abc import Iterator, Iterable
T = TypeVar("T")

def column(source: Iterable[list[T]], x: int) -> Iterator[T]:
    for row in source:
        yield row[x]
```

This allows us to use the `corr()` function from *Chapter 4*, *Working with Collections*, to compute the correlation between the two columns of data.

This is how we can compute all combinations of correlations:

```
from collections.abc import Iterator
from itertools import *
from Chapter04.ch04_ex4 import corr

def multi_corr(
        source: list[list[float]]
    ) -> Iterator[tuple[float, float, float]]:
    n = len(source[0])
    for p, q in combinations(range(n), 2):
        header_p, *data_p = list(column(source, p))
        header_q, *data_q = list(column(source, q))
        if header_p == header_q:
            continue
        r_pq = corr(data_p, data_q)
        yield header_p, header_q, r_pq
```

For each combination of columns, we've extracted the two columns of data from our dataset. The `header_p, *data_p =...` statement uses multiple assignments to separate the first item in the sequence, the header, from the remaining rows of data. If the headers match, we're comparing a variable to itself. This will be `True` for the three combinations of year

and year that stem from the redundant year columns.

Given a combination of columns, we will compute the correlation function and then print the two headings along with the correlation of the columns. We've intentionally chosen two datasets that show spurious correlations with a third dataset that does not follow the same pattern as closely. In spite of this, the correlations are remarkably high.

The results look like this:

```
0.96: year vs Per capita consumption of cheese (US) Pounds (USDA)

0.95: year vs Number of people who died by becoming tangled in their
bedsheets Deaths (US) (CDC)

0.92: year vs Per capita consumption of mozzarella cheese (US) Pounds
(USDA)

0.98: year vs Civil engineering doctorates awarded (US) Degrees
awarded
(National Science Foundation)

-0.80: year vs US crude oil imports from Venezuela Millions of
barrels
(Dept. of Energy)

-0.95: year vs Per capita consumption of high fructose corn syrup
(US)
Pounds (USDA)

0.95: Per capita consumption of cheese (US) Pounds (USDA) vs Number
of
people who died by becoming tangled in their bedsheets Deaths (US)
(CDC)

0.96: Per capita consumption of cheese (US)Pounds (USDA) vs year

0.98: Per capita consumption of cheese (US)Pounds (USDA) vs Per
capita
```

```
consumption of mozzarella cheese (US)Pounds (USDA)

...

0.88: US crude oil imports from Venezuela Millions of barrels (Dept.
of
Energy) vs Per capita consumption of high fructose corn syrup
(US)Pounds
(USDA)
```

It's not at all clear what this pattern means. Why do these values correlate? The presence of spurious correlations with no significance can cloud statistical analysis. We've located data that has strangely high correlations with no obvious causal factors.

What's important is that a simple expression, `combinations(range(9), 2)`, enumerates all the possible combinations of data. This kind of succinct, expressive technique makes it easier to focus on the data analysis issues instead of the combinatoric algorithm considerations.

Combinations with replacement

The `itertools` library has two functions for generating combinations of items selected from some set of values. The `combinations()` function reflects our expectations when dealing hands from a deck of cards: each card will appear at most once. The `combinations_with_replacement()` function reflects the idea of taking a card from a deck, writing it down, and then shuffling it back into the deck before selecting another card. This second procedure could potentially yield a five-card sample with five aces of spades.

We can see this more clearly by using the following kind of expression:

```
>>> import itertools
>>> from pprint import pprint
>>> pprint(
... list(itertools.combinations([1,2,3,4,5,6], 2))
... )
[(1, 2),
```

```
 (1, 3),
 (1, 4),
...
 (4, 6),
 (5, 6)]
>>> pprint(
... list(itertools.combinations_with_replacement([1,2,3,4,5,6], 2))
... )
[(1, 1),
 (1, 2),
 (1, 3),
...
 (5, 5),
 (5, 6),
 (6, 6)]
```

There are $\binom{6}{2} = 15$ combinations of six things taken two at a time. There are $6^2 = 36$ combinations when replacement is permitted, since any value is a possible member of the result.

Recipes

The `itertools` chapter of the Python library documentation is outstanding. The basic definitions are followed by a series of recipes that are extremely clear and helpful. Since there's no reason to reproduce these, we'll reference them here. They are required reading materials on functional programming in Python.

The *Itertools Recipes* section in the *Python Standard Library* is a wonderful resource. Visit `https://docs.python.org/3/library/itertools.html#itertools-recipes` for more details.

These function definitions aren't importable functions in the `itertools` modules. These are ideas that need to be read and understood and then, perhaps, copied or modified before inclusion in an application.

The following table summarizes some recipes that show functional programming algorithms built from the itertools basics:

Function Name	Arguments	Results
`powerset`	`(iterable)`	Generate all the subsets of the iterable. Each subset is a tuple object, not a set instance.
`random\_product`	`(*args, repeat=1)`	Randomly select from `product()`.
`random\_permutation`	`(iterable, r=None)`	Randomly select from `permutations()`.
`random\_combination`	`(iterable, r)`	Randomly select from `combinations()`.

Summary

In this chapter, we looked at a number of functions in the `itertools` module. This standard library module provides a number of functions that help us work with iterators in sophisticated ways.

We looked at these combination-producing functions:

- The `product()` function computes all the possible combinations of the elements chosen from two or more collections.
- The `permutations()` function gives us different ways to reorder a given set of values.
- The `combinations()` function returns all the possible subsets of an original set.

We also looked at ways in which the `product()` and `permutations()` functions can be used naively to create extremely large result sets. This is an important cautionary note. A succinct and expressive algorithm can also involve a vast amount of computation. We must perform basic complexity analysis to be sure that the code will finish in a reasonable amount of time.

In the next chapter, we'll look at the `functools` module. This module includes some tools to work with functions as first-class objects. This builds on some material shown in *Chapter 2, Introducing Essential Functional Concepts*, and *Chapter 5, Higher-Order Functions*.

Exercises

This chapter's exercises are based on code available from Packt Publishing on GitHub. See `https://github.com/PacktPublishing/Functional-Python-Programming-3rd-Editi` on.

In some cases, the reader will notice that the code provided on GitHub includes partial solutions to some of the exercises. These serve as hints, allowing the reader to explore alternative solutions.

In many cases, exercises will need unit test cases to confirm they actually solve the problem. These are often identical to the unit test cases already provided in the GitHub repository. The reader should replace the book's example function name with their own solution to confirm that it works.

Alternative distance computations

See *Effects of Distance Measure Choice on KNN Classifier Performance - A Review.* This is available at `https://arxiv.org/pdf/1708.04321`. In this paper, dozens of distance metrics are examined for their utility in implementing the ***k*-Nearest Neighbors (*k*-NN)** classifier.

Some of these are also suitable for the color-matching algorithm presented in this chapter. We defined a color, c, a three-tuple, (r, g, b), based on the Red, Green, and Blue components of the color. We can compute the distance between two colors, $D(c_1, c_2)$, based on their RGB components:

$$D(c_1, c_2) = D\big((r_1, g_1, b_1), (r_2, g_2, b_2)\big)$$

We showed two: Euclidean and Manhattan distances. Here are some more formal defini-

tions:

$$ED = \sqrt{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2}$$

$$MD = |r_1 - r_2| + |g_1 - g_2| + |b_1 - b_2|$$

Some additional examples include these:

- **The Chebyshev Distance** (**CD**) is the maximum of the absolute values of each color difference:

$$CD = \max(|r_1 - r_2|, |g_1 - g_2|, |b_1 - b_2|)$$

- The **Sorensen Distance** (**SD**) is a modification of the Manhattan distance that tends to normalize the distance:

$$SD = \frac{|r_1 - r_2| + |g_1 - g_2| + |b_1 - b_2|}{r_1 + r_2 + g_1 + g_2 + b_1 + b_2}$$

Redefine the `make_color_map()` function to be a higher-order function that will accept a distance function as a parameter. All of the distance functions should have a type hint of `Callable[[RGB, Color], float]`. Once the `make_color_map()` function has been changed, it becomes possible to create alternative color maps with alternative distance functions.

This function creates a mapping from "masked" RGB values to a defined set of colors. Using a 3-bit mask defines a mapping from $2^3 \times 2^3 \times 2^3 = 512$ possible RGB values to the domain of 133 colors.

The function's definition should look like this:

```
def make_color_map(colors: Sequence[Color], distance: Callable[[RGB,
Color], float]) -> dict[RGB, Color]:
```

How much difference does the choice of distance function make? How can we characterize the mapping from the large collection of possible RGB values to the limited domain of subset colors? Is a histogram showing how many distinct RGB values map to a subset color sensible and informative?

Actual domain of pixel color values

When creating a color map, a mask was used to reduce the domain of possible colors from $2^8 \times 2^8 \times 2^8 = 16,777,216$ to a more manageable $2^3 \times 2^3 \times 2^3 = 512$ possible values.

Does it make sense to scan the original image using the mask value to determine the actual domain of available colors? A given image, may, for example, have only 210 distinct colors when a 3-bit mask is used. How much additional time is required to create this summary of colors actually in use?

Can the color summary be further optimized? Could we, for example, exclude rarely used colors? If we do exclude these rarely used colors, how do we replace a pixel's color with more commonly used colors? What changes in an image if we use the color of the majority of the neighboring pixels to replace pixels with rarely used colors?

Consider an algorithm that makes the following two passes over an image's pixels:

1. Create a `Counter` with the frequency of each color.
2. For colors with fewer than some threshold, ϵ, locate the neighboring pixels. In a corner, there may be as few as three. In the middle, there will be no more than eight. Find the color of the majority of those pixels and replace the outlier.

Algorithm 8 Imperative iteration

Require: An image, $I = \{p_{0,0}, p_{0,1}, p_{0,2}, ..., p_{x,y}, ..., p_{w,h}\}$; each pixel $p_{x,y}$ is a 3-byte color.

1: $h \leftarrow$ **Counter**(I)
2: **for** $p_{x,y}, f \in h$.**items**() **if** $f < \epsilon$ **do**
3: $\quad a \leftarrow \{p_{x-1,y-1}, p_{x,y-1}, p_{x+1,y-1}, p_{x,y-1}, ..., p_{x+1,y+1}\}$ ▷ Locations adjacent to $p_{x,y}$
4: $\quad n \leftarrow$ **Counter**(a)
5: $\quad m \leftarrow n$.**most_common**()
6: $\quad p_{x,y} \leftarrow m$ ▷ Updates the color
7: **end for**

Before starting on this algorithm, it's important to consider any "edge" cases. Specifically, there is a potential complication when rarely used colors are adjacent.

Consider this case:

$p_{(0,0)}$	$p_{(1,0)}$	$p_{(2,0)}$
$p_{(0,1)}$	$p_{(1,1)}$	$p_{(2,1)}$
$p_{(0,2)}$	$p_{(1,2)}$	$p_{(2,2)}$

If the four pixels in the top-left corner, $p_{(0,0)}$, $p_{(1,0)}$, $p_{(0,1)}$, and $p_{(1,1)}$, all had rarely used colors, then it would be difficult to pick a majority color to replace $p_{(0,0)}$.

In the case where there are no non-rare colors surrounding a pixel with a rare color, the algorithm would need to queue this up for later resolution after the neighbors have been processed. In this example, the color for pixel $p_{(1,0)}$ can be computed using neighbors that are not rare colors. After $p_{(0,1)}$ and $p_{(1,1)}$ are also resolved, then $p_{(0,0)}$ can be replaced with the majority color of the three neighbors.

Is this algorithmic complexity helpful for a picture with 10 million pixels? Choosing one or a few photos arbitrarily isn't a sophisticated survey. However, it can help to avoid overthinking potential problems.

Survey the colors in a collection of images. How common is it to see a single pixel with a unique color? If you don't have a private collection of images, visit `kaggle.com` to look for image datasets that can be examined.

Cribbage hand scoring

The card game of cribbage involves a phase where a player's hand is evaluated. A player will use four cards that are dealt to them, plus a fifth card, called the starter.

To avoid overusing the word "points," we'll consider each card to have a number of pips. Each face card is counted as having 10 pips; all other cards have a number of pips equal to their rank. Aces have a single pip.

The scoring involves the following combinations of cards:

- Any combination of cards that totals 15 pips adds 2 points to the score.
- Pairs – two cards of the same rank – add 2 points to the score.

- Any run of three, four, or five cards adds 3, 4, or 5 points to the score.
- A flush of four cards in a hand adds 4 points to the score. If the starter card is of the same suit, then the flush, as a whole, is 5 points.
- If a jack in a hand has the same suit as the starter card, this adds 1 point to the score.

If a hand contains three cards of the same rank, this is counted as three separate pairs, worth 6 points in aggregate.

An interesting hand involves runs with a pair. For example, a hand 7C, 7D, 8H, and 9S, with an irrelevant starter card of a Queen, has two runs—7C, 8H, 9S, and 7D, 8H, 9S—and a pair of 7's. This is a total of 8 points. Furthermore, there are two combinations that add to 15: 7C, 8H and 7D, 8H, bringing the hand's value to 12 points.

Note that a 4-card run is *not* counted as two overlapping 3-card runs. It's only worth 4 points.

Another interesting example is holding 4C, 5D, 5H, 6S, and the starter card is 3C. There are two distinct runs of 4 cards: 3C, 4C, 5D, 6S, and 3C, 4C, 5H, 6S, as well a pair of 5s, leading to 10 points for this pattern. Additionally, there are two distinct ways to count 15 pips: 4C, 5D, 6S and 4C, 5H, 6S, adding 4 more points to the score.

A handy algorithm for this is to enumerate several combinations and permutations of cards to locate all the scoring. The following rules can be applied:

1. Iterate over the powerset of the cards. This is the set of all subsets: all of the singletons, all of the pairs, all of the triples, etc., up to the set of all five cards. Each of these is a distinct set, some of which will tally to 15 pips. For more information on generating the powerset, see the *Itertools Recipes* section of the Python Standard Library documentation.
2. Enumerate all pairs of cards to compute scores for any pairs. The `combinations()` function works well for this.
3. For sets of five cards, if they're adjacent, ascending values, this is a run. If they're not adjacent, ascending values, then enumerate the sets of four-card runs to see if either of these have adjacent numbers. Failing that test, enumerate all sets of three-card

runs to see if any of these have adjacent numbers. The longest run applies to the score, and shorter runs are ignored.

4. Check the hand and starter for a five-flush. If there's no five-flush, check the hand only for a four-flush. Only one of these two combinations is scored.
5. Also, check to see if the hand has a jack of the same suit as the starter.

Since there are only five cards involved in this, the number of permutations and combinations is rather small. Be prepared to summarize exactly how many combinations and permutations are required for a five-card hand.

10

The Functools Module

Functional programming considers functions to be first-class objects. We've seen several higher-order functions that accept functions as arguments or return functions as results. In this chapter, we'll look at the `functools` library, which contains some tools to help us implement some common functional design patterns.

We'll look at some higher-order functions. This extends the material from *Chapter 5*, *Higher-Order Functions*. We'll continue looking at higher-order function techniques in *Chapter 12*, *Decorator Design Techniques*, as well.

We'll look at the following functions in this module:

- `@cache` and `@lru_cache`: These decorators can be a huge performance boost for certain types of applications.
- `@total_ordering`: This decorator can help create rich comparison operators. Additionally, it lets us look at the more general question of object-oriented design mixed with functional programming.
- `partial()`: This function creates a new function from a function and some parameter

value bindings.

- `reduce()`: This is a higher-order function that generalizes reductions such as `sum()`.
- `singledispatch()`: This function allows us to assemble alternative implementations based on the argument type. It saves us from writing the `match` statement to choose the implementation, keeping the implementations cleanly separated.

We'll defer two additional members of this library to *Chapter 12*, *Decorator Design Techniques*: the `update_wrapper()` and `wraps()` functions. We'll also look more closely at writing our own decorators in the next chapter.

We'll ignore the `cmp_to_key()` function entirely. Its purpose is to help with converting Python 2 code to run under Python 3. Since Python 2 is no longer being actively maintained, we can safely ignore this function.

Function tools

We looked at a number of higher-order functions in *Chapter 5*, *Higher-Order Functions*. Those functions either accept a function as an argument or return a function (or generator expression) as a result. All those higher-order functions have an essential algorithm that is customized by injecting another function. Functions such as `max()`, `min()`, and `sorted()` accept a `key=` function to customize their behavior. Functions such as `map()` and `filter()` accept a function and an iterable and apply the given function to the argument iterable. In the case of the `map()` function, the results of the function are simply yielded. In the case of the `filter()` function, the Boolean result of the function is used to yield or reject values from an iterable source.

All the functions in *Chapter 5*, *Higher-Order Functions*, are part of the Python `__builtins__` package, meaning these functions are available without the need to use the `import` statement. They were made ubiquitous because they seem universally useful. The functions in this chapter must be introduced with an `import` statement because they're not quite so universally helpful.

The `reduce()` function straddles this fence. It was originally built in. After some discussion, it was moved from the `__builtins__` package to the `functools` module because of the possibility of really poor performance. Later in this chapter, we'll see how seemingly simple operations can perform remarkably poorly.

Memoizing previous results with cache

The `@cache` and `@lru_cache` decorators transform a given function into a function that might perform more quickly. **LRU** means **Least Recently Used**—a finite pool of recently used items is retained. Items not recently used are discarded to keep the pool to a bounded size. The `@cache` has no storage management and requires a little bit of consideration to be sure it won't consume all available memory.

Since these are decorators, we can apply one of them to any function that might benefit from caching previous results. We can use it as follows:

```
from functools import lru_cache

@lru_cache(128)
def fibc(n: int) -> int:
    if n == 0: return 0
    if n == 1: return 1
    return fibc(n-1) + fibc(n-2)
```

This is an example based on *Chapter 6, Recursions and Reductions*. We've applied the `@lru_cache` decorator to the naive Fibonacci number calculation. Because of this decoration, each evaluation of the `fibc(n)` function will now be checked against a cache maintained by the decorator. If the argument value, `n`, is in the cache, the previously computed result is used instead of doing a potentially expensive re-calculation. Each new collection of argument values and return value updates the cache.

We highlight this example because the naive recursion is quite expensive in this case. The complexity of computing any given Fibonacci number, $F_n = F_{n-1} + F_{n-2}$, involves not merely computing F_{n-1} but also F_{n-2}. This tree of values leads to a complexity in the order of $\mathbf{O}(2^n)$.

The argument value of `128` is the size of the cache. This is used to limit the amount of memory used for the cache. When the cache is full, the LRU item is replaced.

We can try to confirm the benefits empirically using the `timeit` module. We can execute the two implementations 1,000 times each to see how the timing compares. Using the `fib(20)` and `fibc(20)` methods shows just how costly this calculation is without the benefit of caching. Because the naive version is so slow, the `timeit` number of repetitions was reduced to only 1,000. Here are the results (in seconds):

- Naive: 3.23
- Cached: 0.0779

Note that we can't trivially use the `timeit` module on the `fibc()` function. The cached values will remain in place; we'll only evaluate the complete `fibc(20)` calculation once, which populates values in the cache. Each of the remaining 999 iterations will simply fetch the value from the cache. We need to actually clear the cache between uses of the `fibc()` function or the time drops to almost zero. This is done with a `fibc.cache_clear()` method built by the decorator.

The concept of memoization is powerful. There are many algorithms that can benefit from memoization of results. Because the `@cache` decorator applies to a function, it means using a functional programming approach can also lead to high-performance software. Functions with side effects are rarely good candidates for memoization; pure functions will work out best.

We'll look at one more example of the benefits of caching. This will involve a small computation that also has repeated values. The number of combinations of p things taken in groups of r is often stated as follows:

$$\binom{p}{r} = \frac{p!}{r!(p - r!)}$$

This binomial function involves computing three factorial values. It might make sense to use the `@cache` decorator on a factorial function. A program that calculates a number

of binomial values will not need to re-compute all of those factorials. For cases where similar values are computed repeatedly, the speedup can be impressive. For situations where the cached values are rarely reused, the overheads of maintaining the cached values may outweigh any speedups.

We've omitted the details of the actual binomial function. It's only one line of code. Caching a built-in function is done like this:

```
from functools import cache
from math import factorial

factorial = cache(factorial)
```

This applies the decorator to an existing function. For more information on this approach to decoration, see *Chapter 12, Decorator Design Techniques*.

When evaluating a binomial function repeatedly, we see the following:

- Naive factorial: 0.174
- Cached factorial: 0.046

It's important to recognize that the cache is a stateful object. This design pushes the edge of the envelope on purely functional programming. One functional ideal is to avoid changes of state. This concept of avoiding stateful variables is exemplified by a recursive function; the current state is contained in the argument values, and not in the changing values of variables. We've seen how tail-call optimization is an essential performance improvement to ensure that this idealized recursion actually works nicely with the available processor hardware and limited memory budgets. In Python, we can do this tail-call optimization manually by replacing the tail recursions with a `for` loop. Caching is a similar kind of optimization; we must implement it manually as needed, knowing that it isn't purely functional programming.

Further, if our design is centered on pure functions—free of side effects—then there are no problems with introducing caching. Applying an `@cache` decorator to a function that

has side effects, for example, the `print()` function, will create confusion: we'll note that evaluations of `print()` with the same argument values won't produce any output because the result value, `None`, will be fetched from cache.

In principle, each call to a function with a cache has two results: the expected result and a new `cache` object available for future evaluations of the function. Pragmatically, in our example, the `cache` object is encapsulated inside the decorated version of the `fibc()` function, and it isn't available for inspection or manipulation.

Caching is not a panacea. Applications that work with float values might not benefit much from memoization because float values are often approximations. The least-significant bits of a float value should be seen as random noise that can prevent the exact equality test in the `@lru_cache` or `@cache` decorator from working.

Defining classes with total ordering

The `@total_ordering` decorator is helpful for creating new class definitions that implement a rich set of comparison operators. This might apply to numeric classes that subclass `numbers.Number`. It may also apply to semi-numeric classes.

As an example of a semi-numeric class, consider a playing card. It has a numeric rank and a symbolic suit. The suit, for example, may not matter for some games. Like ordinary integers, cards have an ordering. We often sum the point values of each card, making them number-like. However, multiplication of cards, *card* × *card*, doesn't really make any sense; a card isn't quite like a number.

We can almost emulate a playing card with a `NamedTuple` base class as follows:

```
from typing import NamedTuple

class Card1(NamedTuple):
    rank: int
    suit: str
```

This model suffers from a profound limitation: all comparisons between cards will include

both the rank and the suit. This leads to the following awkward behavior when we compare the two of spades against the two of clubs:

```
>>> c2s = Card1(2, '\u2660')
>>> c2h = Card1(2, '\u2665')
>>> c2s
Card1(rank=2, suit='♠')
>>> c2h
Card1(rank=2, suit='♥')

# The following is undesirable for some games:

>>> c2h == c2s
False
```

The default comparison rule is fine for some games. It does not work well for those games in which comparisons focus on rank and ignore suit.

For some games, it can be better for the default comparisons between cards to be based on only their rank.

The following class definition is appropriate for games where the suit isn't a primary concern:

```
from functools import total_ordering
from typing import NamedTuple

@total_ordering
class Card2(NamedTuple):
    rank: int
    suit: str

    def __str__(self) -> str:
        return f"{self.rank:2d}{self.suit}"
    def __eq__(self, other: Any) -> bool:
        match other:
            case Card2():
```

```
                return self.rank == other.rank
            case int():
                return self.rank == other
            case _:
                return NotImplemented
    def __lt__(self, other: Any) -> bool:
        match other:
            case Card2():
                return self.rank < other.rank
            case int():
                return self.rank < other
            case _:
                return NotImplemented
```

This class extends the `NamedTuple` class. We've provided a `__str__()` method to print a string representation of a `Card2` object.

There are two comparisons defined—one for equality and one for ordering. A wide variety of comparisons can be defined, and the `@total_ordering` decorator handles the construction of the remaining comparisons. In this case, the decorator creates `__le__()`, `__gt__()`, and `__ge__()` from these two definitions. The default implementation of `__ne__()` uses `__eq__()`; this works without using a decorator.

Both the methods provided in this class allow two kinds of comparisons—between `Card2` objects, and also between a `Card2` object and an integer. The type hint must be `Any` to remain compatible with the superclass definition of `__eq__()` and `__lt__()`. It's clear that it could be narrowed to `Union[Card2, int]`, but this conflicts with the definition inherited from the superclass.

First, this class offers proper comparison of only the ranks, as follows:

```
>>> c2s = Card2(2, '\u2660')
>>> c2h = Card2(2, '\u2665')
>>> c2h == c2s
True
```

```
>>> c2h == 2
True
>>> 2 == c2h
True
```

We can use this class for a number of simulations with simplified syntax to compare ranks of cards. Furthermore, the decorator builds a rich set of comparison operators as follows:

```
>>> c2s = Card2(2, '\u2660')
>>> c3h = Card2(3, '\u2665')
>>> c4c = Card2(4, '\u2663')
>>> c2s <= c3h < c4c
True
```

We didn't need to write all of the comparison method functions; they were generated by the decorator. The decorator's creation of operators isn't perfect. In our case, we've asked for comparisons with integers as well as between `Card` instances. This reveals some problems.

Operations such as the `c4c > 3` and `3 < c4c` comparisons would raise `TypeError` exceptions because of the way the operators are resolved to find a class that implements the comparison. This exposes a limitation of the methods created by the `@total_ordering` decorator. Specifically, the generated methods won't have clever type matching rules. If we need type matching in all of the comparisons, we'll need to write all of the methods.

Applying partial arguments with partial()

The `partial()` function leads to something called a **partial application**. A partially applied function is a new function built from an old function and a subset of the required argument values. It is closely related to the concept of **currying**. Much of the theoretical background is not relevant here, since currying doesn't apply directly to the way Python functions are implemented. The concept, however, can lead us to some handy simplifications.

We can look at trivial examples as follows:

```
>>> exp2 = partial(pow, 2)
>>> exp2(12)
4096
>>> exp2(17)-1
131071
```

We've created the function `exp2(y)`, which is the `pow(2, y)` function. The `partial()` function binds the first positional parameter to the `pow()` function. When we evaluate the newly created `exp2()` function, we get values computed from the argument bound by the `partial()` function, plus the additional argument provided to the `exp2()` function.

The bindings of positional parameters are handled in a strict left-to-right order. Functions that accept keyword parameters can also be provided when building the partially applied function.

We can also create this kind of partially applied function with a lambda form as follows:

```
>>> exp2 = lambda y: pow(2, y)
```

Neither is clearly superior. The use of `partial()` can help a reader understand the design intent. The use of a lambda may not have the same explanatory power.

Partial functions can be very handy in a context where we want to avoid repeating argument values to a function. We may, for example, be normalizing data after computing the mean and standard deviation. These normalized values are sometimes called Z-scores. While we can define a function `z(mean: float, stdev: float, score: float) -> float:`, this has the clutter of many argument values that don't change once the mean and standard deviation are known.

We prefer something like the following example:

```
>>> m = mean(some_data)
>>> std = stdev(some_data)
>>> z_value = partial(z, m, std)
>>> normalized_some_data = [z_value(x) for x in some_data]
```

The creation of the `z_value()` partial function is not – strictly speaking – needed. Having this function can clarify the expression that creates the `normalized_some_data` object. Using `z_value(x)` seems slightly more readable than `z_value(m, std, x)`.

We'll return to the `partial()` function in *Chapter 13*, *The PyMonad Library*, and look at how we can accomplish this same kind of function definition using *currying*.

Reducing sets of data with the reduce() function

The `sum()`, `len()`, `max()`, and `min()` functions are, in a way, all specializations of a more general algorithm expressed by the `reduce()` function. See *Chapter 5*, *Higher-Order Functions* for more on these functions. The `reduce()` function is a higher-order function that folds a binary operation into each pair of items in an iterable.

A sequence object is given as follows:

```
>>> d = [2, 4, 4, 4, 5, 5, 7, 9]
```

The expression `reduce(lambda x, y: x+y, d)` will fold in `+` operators to the list as if we were evaluating the following:

```
>>> from functools import reduce

>>> reduce(lambda x, y: x+y, d)
40
>>> 2+4+4+4+5+5+7+9
40
```

It can help to include `()` to show the effective left-to-right grouping as follows:

```
>>> ((((((2+4)+4)+4)+5)+5)+7)+9
40
```

Python's standard interpretation of expressions involves a left-to-right evaluation of operators. Consequently, a fold left doesn't involve a change in meaning. Many functional programming languages including **Haskell** and **OCaml** (among many others) offer a fold-right alternative. When used in conjunction with recursion, a compiler can do a number of clever optimizations. This isn't available in Python; a reduction is always left to right.

We can also provide an initial value as follows:

```
>>> reduce(lambda x, y: x+y**2, d, 0)
232
```

If we don't supply an initial value, the initial value from the sequence is used as the initialization. Providing an initial value is essential when there's a `map()` function as well as a `reduce()` function. The following is how the right answer is computed with an explicit `0` initializer:

```
>>> 0 + 2**2 + 4**2 + 4**2 + 4**2 + 5**2 + 5**2 + 7**2 + 9**2
232
```

If we omit the initialization of `0`, the `reduce()` function uses the first item as an initial value. This value does not have the transformation function applied, which leads to the wrong answer. In effect, the `reduce()` without a proper initial value is computing this:

```
>>> 2 + 4**2 + 4**2 + 4**2 + 5**2 + 5**2 + 7**2 + 9**2
230
```

This kind of mistake is part of the reason why `reduce()` must be used carefully.

We can define a number of common and built-in reductions using the `reduce()` higher-order function as follows:

```
from collections.abc import Callable
from functools import reduce
from typing import cast, TypeAlias

FloatFT: TypeAlias = Callable[[float, float], float]

sum2 = lambda data: reduce(cast(FloatFT, lambda x, y: x+y**2), data,
0.0)
sum = lambda data: reduce(cast(FloatFT, lambda x, y: x+y), data, 0.0)
count = lambda data: reduce(cast(FloatFT, lambda x, y: x+1), data,
0.0)
min = lambda data: reduce(cast(FloatFT, lambda x, y: x if x < y else
y), data)
max = lambda data: reduce(cast(FloatFT, lambda x, y: x if x > y else
y), data)
```

The `sum2()` reduction function is the sum of squares, useful for computing the standard deviation of a set of samples. This `sum()` reduction function mimics the built-in `sum()` function. The `count()` reduction function is similar to the `len()` function, but it can work on an iterable, whereas the `len()` function can only work on a materialized collection object.

The `cast()` functions notify **mypy** of the intended types for the lambda objects. Without this, the default type hint for a lambda object is `Any`, which isn't the intent for these functions. The type hint `FloatFT` describes a float function that accepts two float argument values and returns a float object as a result.

The `min()` and `max()` functions mimic the built-in reductions. Because the first item of the iterable is used for initialization, these two functions will work properly. If we provided an initial value to these `reduce()` functions, we might incorrectly use a value that never occurred in the original iterable.

The complexity of the type hints is a suggestion that lambda objects don't convey enough information to tools like **mypy**. While a lambda is valid Python, it can be difficult to examine in detail. This leads to the following tip:

A good design uses small function definitions.

A complete function definition lets us provide default values, documentation, and `doctest` test cases.

Combining map() and reduce()

We can see how to build higher-order functions around these foundational definitions. We can define a map-reduce function that combines the `map()` and `reduce()` functions as follows:

```
from collections.abc import Callable, Iterable
from functools import reduce
from typing import TypeVar, cast

ST = TypeVar("ST")

def map_reduce(
    map_fun: Callable[[ST], float],
    reduce_fun: Callable[[float, float], float],
    source: Iterable[ST],
    initial: float = 0
) -> float:
    return reduce(reduce_fun, map(map_fun, source), initial)
```

This definition has a few formal type constraints. First, the `source` iterator produces some consistently typed data. We'll bind the source type to the `ST` type variable to show where consistent types are required. Second, the provided `map_fun()` function accepts one argument of whatever type could be bound to `ST` and produces a float object. Third, the provided `reduce_fun()` function will reduce float objects to return a result of the same type. Because **mypy** is aware of the way Python operators work with integers as well as float values, this works in an integer context as well as a float context.

We can build a sum-of-squares reduction using the `map_fun()` and `reduce_fun()` functions separately as follows:

```
from collections.abc import Iterable

def sum2_mr(source_iter: Iterable[float]) -> float:
    return map_reduce(
        map_fun=lambda y: y**2,
        reduce_fun=lambda x, y: x+y,
        source=source_iter,
        initial=0)
```

In this case, we've used a `lambda y: y**2` argument value as a mapping to square each value. The reduction is the `lambda x, y: x+y` argument value. We don't need to explicitly provide an initial value because the initial value will be the first item in the iterable after the provided `map_fun()` lambda has squared it.

The `lambda x, y: x+y` argument value is the `+` operator. Python offers all of the arithmetic operators as short functions in the `operator` module. (We'll look at this in *Chapter 11*, *The Toolz Package*.) The following is how we can slightly simplify our map-reduce operation:

```
from collections.abc import Iterable
import operator

def sum2_mr2(source: Iterable[float]) -> float:
    return map_reduce(
        lambda y: y**2,
        operator.add,
        source,
        0)
```

We've used the `operator.add` function to sum our values instead of the longer lambda form.

The following is how we can count values in an iterable:

```
def count_mr(source: Iterable[float]) -> float:
    return map_reduce(
        lambda y: 1,
        lambda x, y: x+y,
        source,
        0)
```

We've used the `lambda y: 1` parameter to map each value to the value `1`. The count is then reduced using a lambda or the `operator.add` function.

The general-purpose `reduce()` function allows us to create any species of reduction from a large dataset to a single value. There are some limitations, however, on what we should do with the `reduce()` function.

Using the reduce() and partial() functions

As we saw earlier, the `reduce()` function has a provision for an initial value. The default initial value is zero. This initial value seeds the reduction and will be the default value if the source iterable is empty.

In the following example, we've provided an absurd initial value:

```
>>> import operator
>>> from functools import reduce
>>> d = []
>>> reduce(operator.add, d, "hello world")
'hello world'
```

The initial value provided to the `reduce()` function is a string. Because the iterable source of data, `d` is empty, no operations were performed and the initial value is the final result, even though it's absurdly invalid.

We note a complication here when we try to create a partial function using `reduce()`: there's no sensible way to provide an initial value. This stems from the following root cause: the `reduce()` function has no keyword parameters. For some reductions, we need

to provide values for the first and third positional parameters to the `reduce()` function.

Here's the result of attempting to combine `partial()` and `reduce()`. The following example definitions of partial functions do *not* work correctly:

```
from functools import partial, reduce

psum2 = partial(reduce, lambda x, y: x+y**2)
pcount = partial(reduce, lambda x, y: x+1)
```

The `psum2()` function should compute a sum of squares of a source of values. As we'll see, this does not work as hoped. Here's an example of trying to use these functions based on the `partial()` function:

```
>>> d = [2, 4, 4, 4, 5, 5, 7, 9]
>>> sum2(d)
232.0
>>> psum2(d)
230

>>> count(d)
8.0
>>> pcount(d)
9
```

The sum-of-squares defined as a partial does not use a proper initialization for the sequence of values.

The reduction should start with 0. It will apply the lambda to each value, and sum `0 + 2**2`, `0 + 2**2 + 4**2`, etc. In actual fact, it starts with the first of the values, `2`. Then it applies the lambda to the remaining values, computing `2 + 4**2`, `2 + 4**2 + 4**2`, etc.

There's no work-around using `partial()`. A lambda must be used in these cases where we'd like to apply a transformation while using `reduce()`.

A partial function is an important technique for simplifying a particularly complicated

calculation. When there are numerous parameters, few of which change, then a partial function can be helpful. A partial function can make it easier to refactor a complex computation to use alternative implementations of discrete parts. Since each discrete part is a separately defined function, unit testing can confirm that the results are as expected.

The limitation around the `reduce()` function is a result of a function with two properties:

- Only positional parameters
- The parameters being provided in an awkward order

In the case of `reduce()`, the initial value comes *after* the source of values, making it difficult to provide via `partial()`.

Using the map() and reduce() functions to sanitize raw data

When doing data cleansing, we'll often introduce filters of various degrees of complexity to exclude invalid values. We may also include a mapping to sanitize values in the cases where a valid but improperly formatted value can be replaced with a valid and proper value.

We might produce the following output:

```
from collections.abc import Callable, Iterable
from functools import reduce

def comma_fix(data: str) -> float:
    try:
        return float(data)
    except ValueError:
        return float(data.replace(",", ""))

def clean_sum(
    cleaner: Callable[[str], float],
    data: Iterable[str]
) -> float:
    return reduce(operator.add, map(cleaner, data))
```

We've defined a mapping, the `comma_fix()` function, that will convert data from a nearly correct string format into a usable floating-point value. This will remove the comma character. Another common variation could remove dollar signs and convert to `decimal.Decimal`. We've left this as an exercise for the reader.

We've also defined a map-reduce operation that applies a given cleaner function, the `comma_fix()` function in this case, to the data before doing a `reduce()` function, using the `operator.add` method.

We can apply the previously described function as follows:

```
>>> d = ('1,196', '1,176', '1,269', '1,240', '1,307',
... '1,435', '1,601', '1,654', '1,803', '1,734')

>>> clean_sum(comma_fix, d)
14415.0
```

We've cleaned the data by fixing the commas, as well as computing a sum. The syntax is very convenient for combining these two operations.

We have to be careful, however, of using the cleaning function more than once. If we're also going to compute a sum of squares, we really should not execute the following kinds of processing steps:

```
>>> sum = clean_sum(comma_fix, d)
>>> comma_fix_squared = lambda x: comma_fix(x)**2
>>> sum_2 = clean_sum(comma_fix_squared, d)
```

Using `clean_sum()` expressions more than once means we'll do the comma-fixing operation more than once on the source data. This is a poor design. It would be better to cache the intermediate numeric results of the `comma_fix()` function. Using a `@cache` decorator can help. Materializing the sanitized intermediate values as a temporary sequence object is better. Comparing the performance of different caching options is left as an exercise.

Using the groupby() and reduce() functions

A common requirement is to summarize data after partitioning it into groups. We can use a `defaultdict(list)` method to partition data. We can then analyze each partition separately. In *Chapter 4*, *Working with Collections*, we looked at some ways to group and partition. In *Chapter 8*, *The Itertools Module*, we looked at others.

The following is some sample data that we need to analyze:

```
>>> data = [('4', 6.1), ('1', 4.0), ('2', 8.3), ('2', 6.5),
... ('1', 4.6), ('2', 6.8), ('3', 9.3), ('2', 7.8),
... ('2', 9.2), ('4', 5.6), ('3', 10.5), ('1', 5.8),
... ('4', 3.8), ('3', 8.1), ('3', 8.0), ('1', 6.9),
... ('3', 6.9), ('4', 6.2), ('1', 5.4), ('4', 5.8)]
```

We've got a sequence of raw data values with a key (a short string) and a measurement for each key (a float value).

One way to produce usable groups from this data is to build a dictionary that maps a key to a list of members in this groups, as follows:

```
from collections import defaultdict
from collections.abc import Iterable, Callable, Iterator
from typing import Any, TypeVar, Protocol, cast
DT = TypeVar("DT")

class Comparable(Protocol):
    def __lt__(self, __other: Any) -> bool: ...
    def __gt__(self, __other: Any) -> bool: ...
    def __hash__(self) -> int: ...
KT = TypeVar("KT", bound=Comparable)

def partition(
    source: Iterable[DT],
    key: Callable[[DT], KT] = cast(Callable[[DT], KT], lambda x: x)
) -> Iterator[tuple[KT, Iterator[DT]]]:
    """Sorting deferred."""
```

```
    pd: dict[KT, list[DT]] = defaultdict(list)
    for item in source:
        pd[key(item)].append(item)
    for k in sorted(pd):
        yield k, iter(pd[k])
```

This will separate each item in the iterable into a group based on the key. The iterable source of data is described using a type variable of `DT`, representing the type of each data item. The `key()` function is used to extract a key value from each item. This function produces an object of some key type, `KT`, that is generally distinct from the original data item type, `DT`. When looking at the sample data, the type of each data item is a tuple. The keys are of type `str`. The callable function for extracting a key transforms a tuple into a string.

This key value extracted from each data item is used to append each item to a list in the `pd` dictionary. The `defaultdict` object is defined as mapping each key, `KT`, to a list of the data items, `list[DT]`.

The resulting value of this function matches the results of the `itertools.groupby()` function. It's an iterable sequence of the `(group key, iterator)` tuples. The group key value will be of the type produced by the key function. The iterator will provide a sequence of the original data items.

The following is the same feature defined with the `itertools.groupby()` function:

```
from itertools import groupby
from collections.abc import Iterable, Callable, Iterator

def partition_s(
    source: Iterable[DT],
    key: Callable[[DT], KT] = cast(Callable[[DT], KT], lambda x: x)
) -> Iterable[tuple[KT, Iterator[DT]]]:
    """Sort source data"""
    return groupby(sorted(source, key=key), key)
```

The important difference in the inputs to each function is that the `groupby()` function version requires data to be sorted by the key, whereas the `defaultdict` version doesn't require sorting. For very large sets of data, the sort can be expensive, measured in both time and storage.

Here's the core partitioning operation. This might be used prior to filtering out a group, or it might be used prior to computing statistics for each group:

```
>>> for key, group_iter in partition(data, key=lambda x: x[0]):
...     print(key, tuple(group_iter))
1 (('1', 4.0), ('1', 4.6), ('1', 5.8), ('1', 6.9), ('1', 5.4))
2 (('2', 8.3), ('2', 6.5), ('2', 6.8), ('2', 7.8), ('2', 9.2))
3 (('3', 9.3), ('3', 10.5), ('3', 8.1), ('3', 8.0), ('3', 6.9))
4 (('4', 6.1), ('4', 5.6), ('4', 3.8), ('4', 6.2), ('4', 5.8))
```

We can summarize this grouped data as follows:

```
from collections.abc import Iterable, Sequence

def summarize(
    key: KT,
    item_iter: Iterable[tuple[KT, float]]
) -> tuple[KT, float, float]:
    # mean = lambda seq: sum(seq) / len(seq)
    def mean(seq: Sequence[float]) -> float:
        return sum(seq) / len(seq)
    # var = lambda mean, seq: sum(...)
    def var(mean: float, seq: Sequence[float]) -> float:
        return sum((x - mean) ** 2 / (len(seq)-1) for x in seq)

    values = tuple(v for k, v in item_iter)
    m = mean(values)
    return key, m, var(m, values)
```

The results of the `partition()` functions will be a sequence of *(key, iterator)* two-tuples.

The `summarize()` function accepts the two-tuple and decomposes it into the key and the iterator over the original data items. In this function, the data items are defined as `tuple[KT, float]`, a key of some type, `KT`, and a numeric value. From each two-tuple in the `item_iter` iterator we want the value portion, and we use a generator expression to create a tuple of only the values.

We can also use the expression `map(snd, item_iter)` to pick the second item from each of the two-tuples. This requires a definition of `snd = lambda x: x[1]` or perhaps `snd = operator.itemgetter(1)`. The name `snd` is a short form of *second*.

We can use the following command to apply the `summarize()` function to each partition:

```
>>> from itertools import starmap
>>> partition1 = partition(data, key=lambda x: x[0])
>>> groups1 = starmap(summarize, partition1)
```

This uses the `starmap()` function from the `itertools` module. See *Chapter 8, The Itertools Module*. An alternative definition using the `partition_s()` function is as follows:

```
>>> partition2 = partition_s(data, key=lambda x: x[0])
>>> groups2 = starmap(summarize, partition2)
```

Both will provide us with summary values for each group. The resulting group statistics look as follows:

```
1 5.34 1.25
2 7.72 1.22
3 8.56 1.9
4 5.5 0.96
```

The variance can be used as part of a χ^2 (chi-squared) test to determine if the null hypothesis holds for this data. The null hypothesis asserts that there's nothing to see: the variance in the data is essentially random. We can also compare the data between the four groups

to see if the various means are consistent with the null hypothesis or if there is some statistically significant variation.

Avoiding problems with reduce()

There is a dark side to the `reduce()` function. We must avoid expressions like the following:

```
reduce(operator.add, list_of_strings, "")
```

This does work, because Python will apply the generic `add` operator between two operands, which are strings. However, it will compute a large number of intermediate string objects, a relatively costly operation. An alternative is the `"".join(list_of_strings)` expression. A little study with `timeit` reveals that the `string.join()` approach is more efficient than the generic `reduce()` version. We'll leave the data collection and analysis as an exercise for the reader.

In general, it's best to scrutinize `reduce()` operations where the function provided creates or modifies a collection of some kind. It's possible to have a superficially simple-looking expression that creates very large intermediate results. For example, we might write `reduce(accumulate_details, some_source, {})` without thinking of the way the `accumulate_details()` function updates a dictionary. We might be better off looking at ways to rewrite the underlying `accumulate_details()` function to accept a sequence instead of a single item.

Handling multiple types with singledispatch

We'll often have functions which have similar semantics but distinct implementations based on the type of data presented. We might have a function that works for either a subclass of `NamedTuple`, or `TypedDict`. The syntax for working with these objects is distinct, and we can't use a single, generic Python function.

We have the following choices for working with data of distinct types:

- Use the `match` statement with a `case` clause for each distinct type.

- Use the `@singledispatch` decorator to define a number of closely-related functions. This will create the necessary type-matching `match` statement for us.

A small example arises when working with US postal data and spreadsheets. It's common for a US postal ZIP code to be misinterpreted as an integer (or float) value. The town of Andover, MA, for example, has a postal code of 01810. A spreadsheet might misinterpret this as an integer, 1810, dropping the leading zero.

When working with US postal data, we often need a function to normalize ZIP codes as string values, restoring any dropped leading zero values. This function will have at least the following three cases:

- An integer value needs to be converted to a string and have leading zeroes restored.
- A float value, similarly, needs to be converted to a string and have the leading zeroes restored.
- A string value may be a five-digit ZIP code or a nine-digit ZIP code. Depending on the application, we might want to truncate the ZIP codes to ensure they are uniform.

While we can use a `match` statement to handle these three cases, we can also define several closely-related functions. The `@singledispatch` decorator lets us define a "default" function, used when no type matching is possible. We can then overload this function with additional definitions for each of the data types we want to process.

Here is the suite of definitions for a single `zip_format()` function. We'll start with the base definition, used when no other definition will work:

```
from functools import singledispatch
from typing import Any

@singledispatch
def zip_format(zip: Any) -> str:
    raise NotImplementedError(f"unsupported {type(zip)} for
    zip_format()")
```

The `@singledispatch` decorator will create a new decorator, using the name of the function,

zip_format. This new @zip_format decorator can then use be used to create alternative, overloaded definitions. These definitions imply a match statement to distinguish among the alternatives based on type matching rules.

Here are the alternative definitions:

```
@zip_format.register
def _(zip: int) -> str:
    return f"{zip:05d}"

@zip_format.register
def _(zip: float) -> str:
    return f"{zip:05.0f}"

@zip_format.register
def _(zip: str) -> str:
    if "-" in zip:
        zip, box = zip.split("-")
    return f"{zip:0>5s}"
```

Note that each alternative function uses a name, _, that will be ignored. The functions will all be combined into a single zip_format() function that will dispatch an appropriate implementation based on the type of the argument value.

It's also important to note that these functions do not all need to be defined in the same module. We can provide a module with foundational definitions. Additional modules can then import the base definitions and register their unique implementation functions. This permits expansion by adding alternative implementations at the module level.

Summary

In this chapter, we've looked at a number of functions in the functools module. This library module provides a number of functions that help us create sophisticated functions and classes.

We've looked at the @cache and @lru_cache decorators as ways to boost certain types of

applications with frequent re-calculations of the same values. These two decorators are of tremendous value for certain kinds of functions that take integer or string argument values. They can reduce processing by simply implementing memoization. The `@lru_cache` has an upper bound on the memory it will use; this is good for a domain with an unknown size.

We looked at the `@total_ordering` function as a decorator to help us build objects that support rich ordering comparisons. This is at the fringe of functional programming, but is very helpful when creating new kinds of numbers.

The `partial()` function creates a new function with the partial application of argument values. As an alternative, we can build a lambda with similar features. The use case for this is ambiguous.

We also looked at the `reduce()` function as a higher-order function. This generalizes reductions like the `sum()` function. We'll use this function in several examples in later chapters. This fits logically with the `filter()` and `map()` functions as an important higher-order function.

The `@singledispatch` decorator can help us to create a number of functions with similar semantics, but distinct data types for the argument values. This prevents the overhead of an explicit `match` statement. As the software evolves, we can add definitions to the collection of alternatives.

In the next chapter, we'll look at a collection of small topics. We'll examine the `toolz` package, which provides some alternatives to the built-in `itertools` and `functools` modules. This alternative has a number of new features. It also has some overlapping features that are considered from a different perspective, making them more useful for some applications.

We'll also see some additional use of the `operator` module. This module makes some Python operators available as functions, letting us simplify our own function definitions.

We'll also look at some techniques to design flexible decision-making and to allow expressions to be evaluated in a non-strict order.

Exercises

This chapter's exercises are based on code available from Packt Publishing on GitHub. See `https://github.com/PacktPublishing/Functional-Python-Programming-3rd-Editi` on.

In some cases, the reader will notice that the code provided on GitHub includes partial solutions to some of the exercises. These serve as hints, allowing the reader to explore alternative solutions.

In many cases, exercises will need unit test cases to confirm they actually solve the problem. These are often identical to the unit test cases already provided in the GitHub repository. The reader should replace the book's example function name with their own solution to confirm that it works.

Compare string.join() and reduce()

In the *Avoiding problems with reduce()* section of this chapter, we noted that we can combine a list of string values into a single string in the following two ways:

- `reduce(operator.add, list_of_strings, "")`
- `"".join(list_of_strings)`

One of these is considerably more efficient than the other. Use the `timeit` module to find out which is more efficient. The efficiency gain is dramatic, and it can be helpful to know what the ratio of time between these two approaches is.

It's also important to know how the two approaches scale with larger and larger collections of strings. To this end, build a small module that exercises the above two expressions with collections of strings. Use collections with sizes 100, 200, 300, ..., 900 as a way to see how the work scales with the number of strings being concatenated.

Extend the comma_fix() function

In the *Using the map() and reduce() functions to sanitize raw data* section, we defined a mapping, the `comma_fix()` function, that will convert data from a nearly correct string

format into a usable floating-point value. This will remove the comma character.

This function has a misleading name. It's really a string-to-float conversion that tolerates some punctuation. A better name might be `tolerant_str_to_float()`.

Define and test a tolerant string-to-decimal conversion function. This should remove dollar signs, as well as commas, and convert the remaining string to `decimal.Decimal`.

Define and test a tolerant string-to-int conversion function. This should parallel the `tolerant_str_to_float()` by removing only comma characters.

Revise the clean_sum() function

In the *Using the map() and reduce() functions to sanitize raw data* section, we defined a `clean_sum()` function to cleanse and sum a collection of raw string values. For a simple case like computing a mean, this involves a single pass over the data doing conversion and computation.

For a more complex operation, like variance or standard deviation, multiple passes can be burdensome because the string conversion is done repeatedly. This suggests the `clean_sum()` function is a poor design.

The first requirement is a function to compute the mean, variance, and standard deviation of string data:

$$\text{mean}(D) = \frac{\sum_{x \in D} x}{\text{len}(D)}$$

$$\text{var}(D) = \sum_{x \in D} \frac{(x - \text{mean}(D))^2}{\text{len}(D) - 1}$$

$$\text{stdev}(D) = \sqrt{\text{var}(D)}$$

One design alternative is to cache the intermediate numeric results of the `comma_fix()` function. Use the `@cache` decorator to define a `comma_fix()` function. (This function should be renamed to something a little more explicit, like `str_to_float()`.)

Create a very large collection of randomized numeric strings and see which alternative is faster.

Another design alternative is to materialize the sanitized intermediate values. Create a temporary sequence object with the purely numeric values, and then compute the various statistical measures on these purely numeric lists.

In *Chapter 7*, *Complex Stateless Objects*, we presented a way to use `sys.getallocatedblocks()` to understand how much memory was being used by Python. This procedure can be applied here to see which caching alternative uses the least memory.

Present the results to show which design alternative is best for performance and memory use.

11

The Toolz Package

The `toolz` package, offered by the **pytoolz** project on GitHub, contains a number of functional programming features. Specifically, these libraries offer iteration tools, higher-order function tools, and even some components to work with stateful dictionaries in an otherwise stateless function application.

There is some overlap between the `toolz` package and components of the standard library. The `toolz` project decomposes into three significant parts: `itertoolz`, `functoolz`, and `dicttoolz`. The `itertoolz` and `functoolz` modules are designed to mirror the standard library modules `itertools` and `functools`.

We'll look at the following list of topics in this chapter:

- We'll start with star-mapping, where a `f(*args)` is used to provide multiple arguments to a mapping
- We'll also look at some additional `functools.reduce()` topics using the `operator` module
- We'll look at the `toolz` package, which provides capabilities similar to the built-in

itertools and functools packages, but offers a higher level of functional purity
- We'll also look at the operator module and how it leads to some simplification and potential clarification when defining higher-order functions

We'll start with some more advanced use of itertools and functools.reduce(). These two topics will introduce the use cases for the toolz package.

The itertools star map function

The itertools.starmap() function is a variation of the map() higher-order function. The map() function applies a function against each item from a sequence. The starmap(f, S) function presumes each item, i, from the sequence, S, is a tuple, and uses f(*i). The number of items in each tuple must match the number of parameters in the given function.

Here's an example that uses a number of features of the starmap() function:

```
>>> from itertools import starmap, zip_longest
>>> d = starmap(pow, zip_longest([], range(4), fillvalue=60))
>>> list(d)
[1, 60, 3600, 216000]
```

The itertools.zip_longest() function will create a sequence of pairs, [(60, 0), (60, 1), (60, 2), (60, 3)]. It does this because we provided two sequences: the [] brackets and the range(4) parameter. The fillvalue parameter is used when the shorter sequence runs out of data.

When we use the starmap() function, each pair becomes the argument to the given function. In this case, we used the the built-in pow() function, which is the ** operator (we can also import this from the operator() module; the definition is in both places). This expression calculates values for [60**0, 60**1, 60**2, 60**3]. The value of the d variable is [1, 60, 3600, 216000].

The starmap() function expects a sequence of tuples. We have a tidy equivalence between the map(f, x, y) and starmap(f, zip(x, y)) functions.

Here's a continuation of the preceding example of the `itertools.starmap()` function:

```
>>> p = (3, 8, 29, 44)
>>> pi = sum(starmap(truediv, zip(p, d)))
>>> pi
3.1415925925925925
```

We've zipped together two sequences of four values. The value of the `d` variable was computed above using `starmap()`. The `p` variable refers to a simple list of literal items. We zipped these to make pairs of items. We used the `starmap()` function with the `operator.truediv()` function, which is the `/` operator. This will compute a sequence of fractions that we sum. The sum is an approximation of $\pi \approx \frac{3}{60^0} + \frac{8}{60^1} + \frac{29}{60^2} + \frac{44}{60^3}$.

Here's a slightly simpler version that uses the `map(f, x, y)` function instead of the `starmap(f, zip(x,y))` function:

```
>>> pi = sum(map(truediv, p, d))
>>> pi
3.1415925925925925
```

In this example, we effectively converted a base 60 fractional value to base 10. The sequence of values in the `d` variable are the appropriate denominators. A technique similar to the one explained earlier in this section can be used to convert other bases.

Some approximations involve potentially infinite sums (or products). These can be evaluated using similar techniques explained previously in this section. We can leverage the `count()` function in the `itertools` module to generate an arbitrary number of terms in an approximation. We can then use the `takewhile()` function to only accumulate values that contribute a useful level of precision to the answer. Looked at another way, `takewhile()` yields a stream of significant values, and stops consuming values from the stream when an insignificant value is found.

For our next example, we'll leverage the `fact()` function defined in *Chapter 6, Recursions and Reductions*. Look at the *Implementing manual tail-call optimization* section for the

relevant code.

We'll introduce a very similar function, the semifactorial, also called **double factorial**, denoted by the !! symbol. The definition of semifactorial is similar to the definition of factorial. The important difference is that it is the product of *alternate* numbers instead of all numbers. For example, take a look at the following formulas:

- $5!! = 5 \times 3 \times 1$
- $7!! = 7 \times 5 \times 3 \times 1$

Here's the essential function definition:

```
def semifact(n: int) -> int:
    match n:
        case 0 | 1:
            return 1
        case 2:
            return 2
        case _:
            return semifact(n-2)*n
```

Here's an example of computing a sum from a potentially infinite sequence of fractions using the `fact()` and `semifact()` functions:

```
>>> from Chapter06.ch06_ex1 import fact
>>> from itertools import count, takewhile
>>> num = map(fact, count())
>>> den = map(semifact, (2*n+1 for n in count()))

>>> terms = takewhile(
... lambda t: t > 1E-10, map(truediv, num, den))

>>> round(float(2*sum(terms)), 8)
3.14159265
```

The `num` variable is a potentially infinite sequence of numerators, based on the `fact()` function. The `count()` function returns ascending values, starting from zero and continuing

indefinitely. The `den` variable is also a potentially infinite sequence of denominators, based on the semifactorial function. This `den` computation also uses `count()` to create a potentially infinite series of values.

To create terms, we used the `map()` function to apply the `operator.truediv()` function, the `/` operator, to each pair of values. We wrapped this in a `takewhile()` function so that we only take terms from the `map()` output while the value is greater than some relatively small value, in this case, 10^{-10}.

This is a series expansion based on this definition:

$$4 \arctan(1) = \pi = 2 \sum_{0 \le n < \infty} \frac{n!}{(2n+1)!!}$$

An interesting variation of the series expansion theme is to replace the `operator.truediv()` function with the `fractions.Fraction()` function. This will create exact rational values that don't suffer from the limitations of floating-point approximations. We've left the implementation as an exercise for the reader.

All the built-in Python operators are available in the `operator` module. This includes all of the bit-fiddling operators as well as the comparison operators. In some cases, a generator expression may be more succinct or expressive than a rather complicated-looking `starmap()` function with a function that represents an operator.

The `operator` module offers functions that can be more terse than a lambda. We can use the `operator.add` method instead of the `add=lambda a, b: a+b` form. If we have expressions more complex than a single operator, then the lambda object is the only way to write them.

Reducing with operator module functions

We'll look at one more way that we can use the `operator` module definitions: we can use them with the built-in `functools.reduce()` function. The `sum()` function, for example, can be implemented as follows:

```
sum = functools.partial(functools.reduce, operator.add)
```

This creates a partially evaluated version of the `reduce()` function with the first argument supplied. In this case, it's the + operator, implemented via the `operator.add()` function.

If we have a requirement for a similar function that computes a product, we can define it like this:

```
prod = functools.partial(functools.reduce, operator.mul)
```

This follows the pattern shown in the previous example. We have a partially evaluated `reduce()` function with the first argument of the * operator, as implemented by the `operator.mul()` function.

It's not clear whether we can do similar things with too many of the other operators. We might be able to find a use for the `operator.concat()` function.

The `and()` and `or()` functions are the bit-wise & and | operators. These are designed to create integer results.

If we want to perform a reduce using the proper Boolean operations, we should use the `all()` and `any()` functions instead of trying to create something with the `reduce()` function.

Once we have a `prod()` function, this means that the factorial can be defined as follows:

```
fact = lambda n: 1 if n < 2 else n * prod(range(1, n))
```

This has the advantage of being succinct: it provides a single-line definition of factorial. It also has the advantage of not relying on recursion and avoids any problem with stack limitations.

It's not clear that this has any dramatic advantages over the many alternatives we have in

Python. However, the concept of building a complex function from primitive pieces such as the `partial()` and `reduce()` functions and the `operator` module is very elegant. This is an important design strategy for writing functional programs.

Some of the designs can be simplified by using features of the `toolz` package. We'll look at some of the `toolz` package in the next section.

Using the toolz package

The `toolz` package comprises functions that are similar to some of the functions in the built-in `itertools` and `functools` modules. The `toolz` package adds some functions to perform sophisticated processing on dictionary objects. This package has a narrow focus on iterables, dictionaries, and functions. This overlaps nicely with the data structures available in JSON and CSV documents. The idea of processing iterables that come from files or databases allows a Python program to deal with vast amounts of data without the complication of filling memory with the entire collection of objects.

We'll look at a few example functions from the various subsections of the `toolz` package. There are over sixty individual functions in the current release. Additionally, there is a `cytoolz` implementation written in Cython that provides higher performance than the pure Python `toolz` package.

Some itertoolz functions

We'll look at a common data analysis problem of cleaning and organizing data from a number of datasets. In *Chapter 9*, *Itertools for Combinatorics – Permutations and Combinations*, we mentioned several datasets available at `https://www.tylervigen.com`.

Each correlation includes a table of the relevant data. The tables often look like the following example:

	2000	2001	2002	...
Per capita consumption of cheese (US)	29.8	30.1	30.5	...
Number of people who died by becoming tangled in their bedsheets	327	456	509	...

There are generally three rows of data in each example of a spurious correlation. Each row has 10 columns of data, a title, and an empty column that acts as a handy delimiter. A small parsing function, using Beautiful Soup, can extract the essential data from the HTML. This extract isn't immediately useful; more transformations are required.

Here's the core function for extracting the relevant text from HTML:

```
from bs4 import BeautifulSoup  # type: ignore[import]
import urllib.request
from collections.abc import Iterator

def html_data_iter(url: str) -> Iterator[str]:
    with urllib.request.urlopen(url) as page:
        soup = BeautifulSoup(page.read(), 'html.parser')
        data = soup.html.body.table.table
        for subtable in data.table:
            for c in subtable.children:
                yield c.text
```

This `html_data_iter()` function uses `urllib` to read the HTML pages. It creates a `BeautifulSoup` instance from the raw data. The `soup.html.body.table.table` expression provides a navigation path into the HTML structure. This digs down into nested `<table>` tags to locate the data of interest. Within the nested table, there will be other sub-tables that contain rows and columns. Because the various structures can be somewhat inconsistent, it seems best to extract the text and impose a meaningful structure on the text separately.

This `html_data_iter()` function is used like this to acquire data from an HTML page:

```
>>> s7 =
html_data_iter("http://www.tylervigen.com/view_correlation?id=7")
```

The result of this expression is a sequence of strings of text. Many examples have 37 individual strings. These strings can can be divided into 3 rows of 12 strings and a fourth row with a single string value. We can understand these rows as follows:

- The first row has an empty string, ten values of years, plus one more zero-length string.
- The second row has the first data series title, ten values, and an extra zero-length string.
- The third row, like the second, has the second data series title, ten values, and an extra string.
- A fourth row has a single string with the correlation value between the two series.

This requires some reorganization to create a set of sample values we can work with.

We can use `toolz.itertoolz.partition` to divide the sequence of values into groups of 12. If we interleave the three collections using `toolz.itertoolz.interleave`, it will create a sequence with a value from each of the three rows: year, series one, and series two. If this is partitioned into groups of three, each year and the two sample values will be a small three-tuple. We'll quietly drop the additional row with the correlation value.

This isn't the ideal form of the data, but it gets us started on creating useful objects. In the long run, the `toolz` framework encourages us to create dictionaries to contain the sample data. We'll get to the dictionaries later. For now, we'll start with rearranging the source data of the first 36 strings into 3 groups of 12 strings, and then 12 groups of 3 strings. This initial restructuring looks like this:

```
>>> from toolz.itertoolz import partition, interleave

>>> data_iter = partition(3, interleave(partition(12, s7)))
```

```
>>> data = list(data_iter)

>>> from pprint import pprint
>>> pprint(data)
[('',
  'Per capita consumption of cheese (US)Pounds (USDA)',
  'Number of people who died by becoming tangled in their bedsheets
Deaths (US) '
  '(CDC)'),
 ('2000', '29.8', '327'),
 ('2001', '30.1', '456'),
 ('2002', '30.5', '509'),
 ('2003', '30.6', '497'),
 ('2004', '31.3', '596'),
 ('2005', '31.7', '573'),
 ('2006', '32.6', '661'),
 ('2007', '33.1', '741'),
 ('2008', '32.7', '809'),
 ('2009', '32.8', '717'),
 ('', '', '')]
```

The first row, awkwardly, doesn't have a title for the year column. Because this is the very first item in the sequence, we can use a pair of `itertoolz` functions to drop the initial string, which is always `""`, and replace it with something more useful, `"year"`. The resulting sequence will then have empty cells only at the end of each row, allowing us to use `partitionby()` to decompose the long series of strings into four separate rows. The following function definition can be used to break the source data on empty strings into parallel sequences:

```
from toolz.itertoolz import cons, drop  # type: ignore[import]
from toolz.recipes import partitionby  # type: ignore[import]

ROW_COUNT = 0

def row_counter(item: str) -> int:
```

```
    global ROW_COUNT
    rc = ROW_COUNT
    if item == "": ROW_COUNT += 1
    return rc
```

The `row_counter()` function uses a global variable, `ROW_COUNT`, to maintain a stateful count of end-of-row strings. A slightly better design would use a callable object to encapsulate the state information into a class definition. We've left this variant as an exercise for the reader. Using an instance variable in a class with a `__call__()` method has numerous advantages over a global; redesigning this function is helpful because it shows how to limit side effects to the state of objects. We can also use class-level variables and a `@classmethod` to achieve the same kind of isolation.

The following snippet shows how this function is used to partition the input:

```
>>> year_fixup = cons("year", drop(1, s7))
>>> year, series_1, series_2, extra = list(partitionby(row_counter,
year_fixup))
>>> data = list(zip(year, series_1, series_2))

>>> from pprint import pprint
>>> pprint(data)
[('year',
  'Per capita consumption of cheese (US)Pounds (USDA)',
  'Number of people who died by becoming tangled in their bedsheets
Deaths (US) '
  '(CDC)'),
 ('2000', '29.8', '327'),
 ('2001', '30.1', '456'),
 ('2002', '30.5', '509'),
 ('2003', '30.6', '497'),
 ('2004', '31.3', '596'),
 ('2005', '31.7', '573'),
 ('2006', '32.6', '661'),
 ('2007', '33.1', '741'),
```

```
('2008', '32.7', '809'),
('2009', '32.8', '717'),
('', '', '')]
```

The `row_counter()` function is called with each individual string, of which only a few are end-of-row. This allows each row to be partitioned into a separate sequence by the `partitionby()` function. The resulting three sequences are then combined via `zip()` to create a sequence of three-tuples.

This result is identical to the previous example. This variant, however, doesn't depend on there being precisely three rows of 12 values. This variation depends on being able to detect a cell that's at the end of each row. This offers flexibility.

A more useful form for the result is a dictionary for each sample with keys for `year`, `series_1`, and `series_2`. We can transform the sequence of three-tuples into a sequence of dictionaries with a generator expression. The following example builds a sequence of dictionaries:

```
from toolz.itertoolz import cons, drop
from toolz.recipes import partitionby

def make_samples(source: list[str]) -> list[dict[str, float]]:
    # Drop the first "" and prepend "year"
    year_fixup = cons("year", drop(1, source))
    # Restructure to 12 groups of 3
    year, series_1, series_2, extra = list(partitionby(row_counter,
    year_fixup))
    # Drop the first and the (empty) last
    samples = [
        {"year": int(year), "series_1": float(series_1), "series_2":
        float(series_2)}
        for year, series_1, series_2 in drop(1, zip(year, series_1,
        series_2))
        if year
    ]
```

```
    return samples
```

This `make_samples()` function creates a sequence of dictionaries. This, in turn, lets us then use other tools to extract sequences that can be used to compute the coefficient of correlation among the two series. The essential patterns for some of the `itertoolz` functions are similar to the built-in `itertools`.

In some cases, function names conflict with each other, and the semantics are different. For example, `itertoolz.count()` and `itertools.count()` have radically different definitions. The `itertoolz` function is similar to `len()`, while the standard library `itertools` function is a variation of `enumerate()`.

It can help to have the reference documentation for both libraries open when designing an application. This can help you to pick and choose the most useful option between the `itertoolz` package and the standard library `itertools` package.

Note that completely free mixing and matching between these two packages isn't easy. The general approach is to choose the one that offers the right mix of features and use it consistently.

Some dicttoolz functions

One of the ideas behind the `dicttoolz` module of `toolz` is to make dictionary state changes into functions that have side effects. This allows a higher-order function like `map()` to apply a number of updates to a dictionary as part of a larger expression. It makes it slightly easier to manage caches of values, for example, or to accumulate summaries.

For example, the `get_in()` function uses a sequence of key values to navigate down into deeply nested dictionary objects. When working with complex JSON documents, using `get_in(["k1", "k2"])` can be easier than writing a `["k1"]["k2"]` expression.

In the previous examples, we created a sequence of sample dictionaries, named `samples`.

We can extract the various series values from each dictionary and use this to compute a correlation coefficient, as shown here:

```
>>> from toolz.dicttoolz import get_in
>>> from Chapter04.ch04_ex4 import corr

>>> samples = make_samples(s7)
>>> s_1 = [get_in(['series_1'], s) for s in samples]
>>> s_2 = [get_in(['series_2'], s) for s in samples]
>>> round(corr(s_1, s_2), 6)
0.947091
```

In this example, our relatively flat document means we could use `s['series_1']` instead of `get_in(['series_1'], s)`. There's no dramatic advantage to the `get_in()` function. Using `get_in()` does, however, permit future flexibility in situations where the sample's structure needs to become more deeply nested to reflect a shift in the problem domain.

The data, `s7`, is described in *Some itertoolz functions*. It comes from the Spurious Correlations website.

We can set a path `field = ["domain", "example", "series_1"]` and then use this path in a `get_in(path, document)` expression. This isolates the path through the data structure and makes changes easier to manage. This path to the relevant data can even become a configuration parameter if data structures change frequently.

Some functoolz functions

The `functoolz` module of `toolz` has a number of functions that can help with functional design. One idea behind these functions is to provide some names that match the **Clojure** language, permitting easier transitioning between the two languages.

For example, the `@functoolz.memoize` decorator is essentially the same as the standard library `functools.cache`. The word "memoize" matches the **Clojure** language, which some programmers find helpful.

One significant feature of the `@functoolz` module is the ability to compose multiple func-

tions. This is perhaps the most flexible way to approach functional composition in Python.

Consider the earlier example of using the expression `partition(3, interleave(partition(12, s7)))` to restructure source data from a sequence of 37 values to 12 three-tuples. The final string is quietly dropped.

This is in effect a composition of three functions. We can look at it as the following abstract formula:

$$\Big(p(3) \circ i \circ p(12)\Big)(s_7)$$

In the above, $p(3)$ is `partition(3, x)`, i is `interleave(y)`, and $p(12)$ is `partition(12, z)`. This sequence of functions is applied to the source data sequence, s_7.

We can more directly implement the abstraction using `functoolz.compose()`. Before we can look at the `functoolz.compose()` solution, we need to look at the `curry()` function. In *Chapter 10, The Functools Module*, we looked at the `functools.partial()` function. This is similar to the concept behind the `functoolz.curry()` function, with a small difference. When a curried function is evaluated with incomplete arguments, it returns a new curried function with more argument values supplied. When a curried function is evaluated with all of the arguments required, it computes a result:

```
>>> from toolz.functoolz import curry
>>> def some_model(a: float, b: float, x: float) -> float:
...     return x**a * b

>>> curried_model = curry(some_model)
>>> cm_a = curried_model(1.0134)
>>> cm_ab = cm_a(0.7724)
>>> expected = cm_ab(1500)
>>> round(expected, 2)
1277.89
```

The initial evaluation of `curry(some_model)` created a curried function, which we assigned to the `curried_model` variable. This function needs three argument values. When we evaluated `curried_model(1.0134)`, we provided one of the three. The result of this evaluation

is a new curried function with a value for the `a` parameter. The evaluation of `cm_a(0.7724)` provided the second of the three parameter values; this resulted in a new function with values for both the `a` and `b` parameters. We've provided the parameters incrementally to show how a curried function can either act as a higher-order function and return another curried function, or—if all the parameters have values—compute the expected result.

We'll revisit currying again in *Chapter 13, The PyMonad Library*. This will provide another perspective on this idea of using a function and argument values to create a new function.

It's common to see expressions like `model = curry(some_model, 1.0134, 0.7724)` to bind two parameters. Then the expression `model(1500)` will provide a result because all three parameters have values.

The following example shows how to compose a larger function from three separate functions:

```
>>> from toolz.itertoolz import interleave, partition, drop
>>> from toolz.functoolz import compose, curry
>>> steps = [
...     curry(partition, 3),
...     interleave,
...     curry(partition, 12),
... ]
>>> xform = compose(*steps)
>>> data = list(xform(s7))

>>> from pprint import pprint
>>> pprint(data)  # doctest+ ELLIPSIS
[('',
  'Per capita consumption of cheese (US) Pounds (USDA)',
  'Number of people who died by becoming tangled in their bedsheets
Deaths (US) '
  '(CDC)'),
 ('2000', '29.8', '327'),
```

```
...
 ('2009', '32.8', '717'),
 ('', '', '')]
```

Because the `partition()` function requires two parameters, we used the `curry()` function to bind one parameter value. The `interleave()` function, on the other hand, doesn't require multiple parameters, and there's no real need to curry this. While there's no harm done by currying this function, there's no compelling reason to curry it.

The overall `functoolz.compose()` function combines the three individual steps into a single function, which we've assigned to the variable `xform`. The `s7` sequence of strings is provided to the composite function. This applies the functions in right-to-left order, following conventional mathematical rules. The expression $(f \circ g \circ h)(x)$ means $f(g(h(x)))$; the right-most function in the composition is applied first.

There is a `functoolz.compose_left()` function that doesn't follow the mathematical convention. Additionally, there's a `functoolz.pipe()` function that many people find easier to visualize.

Here's an example of using the `functoolz.pipe()` function:

```
>>> from toolz.itertoolz import interleave, partition, drop
>>> from toolz.functoolz import pipe, curry

>>> data_iter = pipe(s7, curry(partition, 12), interleave,
curry(partition, 3))
>>> data = list(data_iter)

>>> from pprint import pprint
>>> pprint(data)  # doctext: +ELLIPSIS
[('',
  'Per capita consumption of cheese (US Pounds (USDA)',
  'Number of people who died by becoming tangled in their bedsheets
Deaths (US) '
  '(CDC)'),
```

```
 ('2000', '29.8', '327'),
...
 ('2009', '32.8', '717'),
 ('', '', '')]
```

This shows the processing steps in the pipeline in left-to-right order. First, `partition(12, s7)` is evaluated. The results of this are presented to `interleave()`. The interleaved results are presented to `curry(partition(3))`. This pipeline concept can be a very flexible way to transform very large volumes of data using the `toolz.itertoolz` library.

In this section, we've seen a number of the functions in the `toolz` package. These functions provide extensive and sophisticated functional programming support. They complement functions that are part of the standard `itertools` and `functools` libraries. It's common to use functions from both libraries to build applications.

Summary

We started with a quick look at some `itertools` and `functools` component features that overlap with components of the `toolz` package. A great many design decisions involve making choices. It's important to know what's built in via Python's Standard Library. This makes it easier to see what the benefits of reaching out to another package might be.

The central topic of this chapter was a look at the `toolz` package. This complements the built-in `itertools` and `functools` modules. The `toolz` package extends the essential concepts using terminology that's somewhat more accessible to folks with experience in other languages. It also provides a helpful focus on the data structures used by JSON and CSV.

In the following chapters, we'll look at how we can build higher-order functions using decorators. These higher-order functions can lead to slightly simpler and clearer syntax. We can use decorators to define an isolated aspect that we need to incorporate into a number of other functions or classes.

Exercises

This chapter's exercises are based on code available from Packt Publishing on GitHub. See `https://github.com/PacktPublishing/Functional-Python-Programming-3rd-Editi` on.

In some cases, the reader will notice that the code provided on GitHub includes partial solutions to some of the exercises. These serve as hints, allowing the reader to explore alternative solutions.

In many cases, exercises will need unit test cases to confirm they actually solve the problem. These are often identical to the unit test cases already provided in the GitHub repository. The reader should replace the book's example function name with their own solution to confirm that it works.

Replace true division with a fraction

In the *The itertools star map function* section, we computed a sum of fractions computed using the `/` true division operator, available from the `operator` module as the `operator.truediv()` function.

An interesting variation of the series expansion theme is to replace the `operator.truediv()` function—which creates `float` objects—with the `fractions.Fraction()` function, which will create `Fraction` objects. Doing this will create exact rational values that don't suffer from the limitations of floating-point approximations.

Change this operator and be sure the summation still approximates π.

Color file parsing

In *Chapter 3*, *Functions, Iterators, and Generators*, the `Crayola.GPL` file was presented without showing the details of the parser. In *Chapter 8*, *The Itertools Module*, a parser was presented that applied a sequence of transformations to the source file. This can be rewritten to use `toolz.functoolz.pipe()`.

First, write and test the new parser.

Compare the two parses. In particular, look for possible extensions and changes to the parsing. What if a file had multiple named color sets? Would it be possible to skip over the irrelevant ones while looking for the relevant collection of colors to parse and extract?

Anscombe's quartet parsing

The Git repository for this book includes a file, `Anscombe.txt`, that contains four series of *(x, y)* pairs. Each series has the same well-known mean and standard deviation. Four distinct models are required to compute an expected y value for a given x value, since each series is surprisingly different.

The data is in a table that starts like the following example:

Anscombe's quartet							
I		II		III		IV	
x	y	x	y	x	y	x	y
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
etc.							

The first row is a title. The second row has series names. The third row has the two column names for each series. The remaining rows all have x and y values for each series.

This needs to be decomposed into four separate sequences. Each sequence should have two-element dictionaries with keys of `"x"` and `"y"`.

The foundation of the parsing is the `csv` module. This will transform each row into a sequence of strings. Each sequence, however, has eight samples from four distinct series in it.

The remaining parsing to decompose the four series can be done either with `toolz.itertoolz` or `itertools`. Write this parser to decompose the Anscombe datasets from each other. Be sure to convert the values from strings to float values so descriptive statistics can be computed for each series.

Waypoint computations

The Git repository for this book includes a file, `Winter 2012-2013.kml`, that contains a series of waypoints for a long trip. In *Chapter 4*, *Working with Collections*, the foundational `row_iter_kml()` function is described. This emits a series of `list[str]` objects for each waypoint along the journey.

To be useful, the waypoints must be processed in pairs. The `toolz.itertoolz.sliding_window()` function is one way to decompose a simple sequence into pairs. The `itertools.pairwise()` function is another candidate.

In *Chapter 7*, *Complex Stateless Objects*, a `distance()` function is presented that computes a close-enough distance between two waypoints. Note that the function was designed to work with complex `NamedTuple` objects. Redesign and reimplement this distance function to work with points represented as dictionaries with keys of "latitude" and "longitude."

The foundation of the source data parsing is the `row_iter_kml()` function, which depends on the underlying `xml.etree` module. This transformed each waypoint into a sequence of strings.

Redesign the source data parsing to use the `toolz` package. The general processing can use `tools.functoolz.pipe` to transform source strings into more useful resulting dictionaries. Be sure to convert latitude and longitude values to properly signed float values.

After the redesign, compare and contrast the two implementations. Which seems more clear and concise? Use the `timeit` module to compare the performance to see if either offers specific performance advantages.

Waypoint geofence

The *Waypoint computations* exercise consumed a file with a number of waypoints. The waypoints were connected to form a journey from start to finish.

It's also sensible to examine the waypoints as isolated location samples with a latitude and longitude. Given the points, a simple boundary can be computed from the greatest and least latitude as well as the greatest and least longitude.

Superficially, this describes a rectangle. Pragmatically, the closer to the North Pole, the closer together the longitude positions become. The area is actually a kind of trapezoid, narrower closer to the pole.

A parsing pipeline similar to the one described in the *Waypoint computations* exercise is required. The waypoints, however, do not have to be combined into pairs. Locate the extrema on each axis to define a box around the overall voyage.

There are several ways to bracket the voyage, as described below:

- Given the extreme edges of the voyage, it's possible to define four points for the four corners of the bounding trapezoid. These four points can be used to locate a midpoint for the journey.
- Given two sequences of latitudes and longitudes, a mean latitude and a mean longitude can be computed.
- Given two sequences of latitudes and longitudes, a median latitude and a median longitude can be computed.

Once the boundaries and center alternatives are known, the equirectangular distance computation (from *Chapter 7, Complex Stateless Objects*) can be used to locate the point on the journey closest to the center.

Callable object for the row_counter() function

In the *Some itertoolz functions* section of this chapter, a `row_counter()` function was defined. It used a global variable to maintain a count of source data items that ended an input row.

A better design is a callable object with an internal state. Consider the following class definition as a base class for your solution:

```
class CountEndingItems:
    def __init__(self, ending_test_function: Callable[[Any], bool])
    -> None:
        ...
    def __call__(self, row: Any) -> int:
```

```
        ...
    def reset(self) -> None:
        ...
```

The idea is to create a callable object, `row_test = CountEndingItems(lambda item: item == "")`. This callable object can then be used with `toolz.itertoolz.partition_by()` as a way to partition the input based on a count of rows that match some given condition.

Finish this class definition. Use it with the `toolz.itertoolz.partition_by()` solution for partitioning input. Contrast the use of a global variable with a stateful, callable object.

12

Decorator Design Techniques

Python offers us many ways to create higher-order functions. In *Chapter 5*, *Higher-Order Functions*, we looked at two techniques: defining a function that accepts a function as an argument, and defining a subclass of `Callable`, which is either initialized with a function or called with a function as an argument.

One of the benefits of decorating functions is that it can create **composite functions**. These are single functions that embody functionality from several sources. It's often helpful to have the decoration syntax as a way to express complex processing.

We can also use decorators to identify classes or functions, often building a **registry**—a collection of related definitions. We may not necessarily create a composite function when building a registry.

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

- Using a decorator to build a function based on another function

- The `wraps()` function in the `functools` module; this can help us build decorators
- The `update_wrapper()` function, which may be helpful in the rare case when we want access to the original function as well as the wrapped function

Decorators as higher-order functions

The core idea of a decorator is to transform some original function into a new function. Used like this, a decorator creates a composite function based on the decorator and the original function being decorated.

A decorator can be used in one of the two following ways:

- As a prefix that creates a new function with the same name as the base function, as follows:

```
@decorator
def base_function() -> None:
    pass
```

- As an explicit operation that returns a new function, possibly with a new name:

```
def base_function() -> None:
    pass

base_function = decorator(base_function)
```

These are two different syntaxes for the same operation. The prefix notation has the advantages of being tidy and succinct. The prefix location is also more visible to some readers. The suffix notation is explicit and slightly more flexible.

While the prefix notation is common, there is one reason for using the suffix notation: we may not want the resulting function to replace the original function. We may want to execute the following command, which allows us to use both the decorated and the undecorated functions:

```
new_function = decorator(base_function)
```

This will build a new function, named `new_function()`, from the original function. When using the `@decorator` syntax, the original function is no longer available for use. Indeed, once the name is reassigned to a new function object, the original object may have no remaining references, and the memory it once occupied may be eligible for reclamation.

A decorator is a function that accepts a function as an argument and returns a function as the result. This basic description is clearly a built-in feature of the language. Superficially, it may seem like we can update or adjust the internal code structure of a function.

Python doesn't work by adjusting the internals of a function. Rather than messing about with the byte codes, Python uses a cleaner approach of defining a new function that wraps the original function. It's easier to process the argument values or the result and leave the original function's core processing alone.

We have two phases of higher-order functions involved in defining a decorator; they are as follows:

- At definition time, a decorator function applies a wrapper to a base function and returns the new, wrapped function. The decoration process can do some one-time-only evaluation as part of building the decorated function. Complex default values can be computed, for example.
- At evaluation time, the wrapping function can (and usually does) evaluate the base function. The wrapping function can pre-process the argument values or post-process the return value (or do both). It's also possible that the wrapping function may avoid calling the base function. In the case of managing a cache, for example, the primary reason for wrapping is to avoid expensive calls to the base function.

Here's an example of a decorator:

```
from collections.abc import Callable
from functools import wraps

def nullable(function: Callable[[float], float]) -> Callable[[float |
None], float | None]:
    @wraps(function)
    def null_wrapper(value: float | None) -> float | None:
        return None if value is None else function(value)
    return null_wrapper
```

We almost always want to use the `@wraps` decorator when creating our own decorators to ensure that the decorated function retains the attributes of the original function. Copying the `__name__` and `__doc__` attributes, for example, ensures that the resulting decorated function has the name and docstring of the original function.

The resulting composite function, defined as the `null_wrapper()` function in the definition of the decorator, is also a type of higher-order function that combines the original function, the `function` callable object, in an expression that preserves the `None` values. Within the resulting `null_wrapper()` function, the original `function` callable object is not an explicit argument; it is a free variable that will get its value from the context in which the `null_wrapper()` function is defined.

The `@nullable` decorator's return value is the newly minted function. It will be assigned to the original function's name. It's important that decorators only return functions and they don't attempt to process data. Decorators use meta-programming: code that creates more code. The resulting `null_wrapper()` function is the function intended to process the application's data.

The `typing` module makes it particularly easy to describe the types of null-aware function and null-aware result, using the `Optional` type definitions or the `|` type operator. The definition `float | None` or `Optional[float]` means `Union[float, None]`; either a `None` object or a `float` object match

the type hint's description.

As an example, we'll assume we have a scaling function that converts input data from nautical miles to statute miles. This might be used with geolocation data that did calculations in nautical miles. The essential conversion from nautical miles, n, to statute miles, s, is a multiplication: $s = 1.15078 \times n$.

We can apply our `@nullable` decorator to create a composite function as follows:

```
import math

@nullable
def st_miles(nm: float) -> float:
    return 1.15078 * nm
```

This will create a function, `st_miles()`, which is a null-aware version of a small mathematical operation. The decoration process returns a version of the `null_wrapper()` function that invokes the original `st_miles()` function. This result will be named `st_miles()` and will have the composite behavior of both the wrapper and the original base function.

We can use this composite `st_miles()` function as follows:

```
>>> some_data = [8.7, 86.9, None, 43.4, 60]
>>> scaled = map(st_miles, some_data)
>>> list(scaled)
[10.011785999999999, 100.002782, None, 49.94385199999999,
69.04679999999999]
```

We've applied the function to a collection of data values. The `None` value politely leads to a `None` result. There was no exception processing involved.

As a second example, here's how we can create a null-aware rounding function using the same decorator:

```
@nullable
def nround4(x: float) -> float:
    return round(x, 4)
```

This function is a partial application of the `round()` function, wrapped to be null-aware. We can use this `nround4()` function to create a better test case for our `st_miles()` function as follows:

```
>>> some_data = [8.7, 86.9, None, 43.4, 60]
>>> scaled = map(st_miles, some_data)
>>> [nround4(v) for v in scaled]
[10.0118, 100.0028, None, 49.9439, 69.0468]
```

This rounded result will be independent of any platform considerations. It's very handy for `doctest` testing.

As an alternative implementation, we could also create these null-aware functions using the following code:

```
st_miles_2: Callable[[float | None], float | None] = (
    nullable(lambda nm: nm * 1.15078)
)
nround4_2: Callable[[float | None], float | None] = (
    nullable(lambda x: round(x, 4))
)
```

We didn't use the `@nullable` decorator in front of the function definition `def` statement. Instead, we applied the `nullable()` function to another function defined as a lambda form. These expressions have the same effect as a decorator in front of a function definition.

Note how it is challenging to apply type hints to lambda forms. The variable `nround4_2` is given a type hint of `Callable` with an argument list of `float | None` and a return type of `float | None`. The use of

the `Callable` hint is appropriate only for positional arguments. In cases where there will be keyword arguments or other complexities, see `https://mypy.readthedocs.io/en/stable/additional_features.html?highlight=callable}extended-callable-types`.

The `@nullable` decorator makes an assumption that the decorated function is unary. We would need to revisit this design to create a more general-purpose null-aware decorator that works with arbitrary collections of arguments.

In *Chapter 13*, *The PyMonad Library*, we'll look at an alternative approach to this problem of tolerating the `None` values. The PyMonad library defines a `Maybe` class of objects, which may have a proper value or may be the `None` value.

Using the functools update_wrapper() function

The `@wraps` decorator applies the `update_wrapper()` function to preserve a few attributes of a wrapped function. In general, this does everything we need by default. This function copies a specific list of attributes from the original function to the resulting function created by a decorator.

The `update_wrapper()` function relies on a global variable defined in the `functools` module to determine what attributes to preserve. The `WRAPPER_ASSIGNMENTS` variable defines the attributes that are copied by default. The default value is this list of attributes to copy:

```
('__module__', '__name__', '__qualname__', '__doc__',
'__annotations__')
```

It's difficult to make meaningful modifications to this list. The internals of the `def` statement aren't open to simple modification or change. This detail is mostly interesting as a piece of reference information.

If we're going to create callable objects, then we may have a class that provides some additional attributes as part of the definition. This could lead to a situation where a

decorator must copy these additional attributes from the original wrapped callable object to the wrapping function being created. However, it seems simpler to make these kinds of changes through object-oriented class design, rather than exploit tricky decorator techniques.

Cross-cutting concerns

One general principle behind decorators is to allow us to build a composite function from the decorator and the original function to which the decorator is applied. The idea is to have a library of common decorators that can provide implementations for common concerns.

We often call these *cross-cutting* concerns because they apply across several functions. These are the sorts of things that we would like to design once through a decorator and have them applied in relevant classes throughout an application or a framework.

Concerns that are often centralized as decorator definitions include the following:

- Logging
- Auditing
- Security
- Handling incomplete data

A logging decorator, for example, may write standardized messages to the application's log file. An audit decorator may write details surrounding a database update. A security decorator may check some runtime context to be sure that the login user has the necessary permissions.

Our example of a *null-aware* wrapper for a function is a cross-cutting concern. In this case, we'd like to have a number of functions handle the `None` values by returning the `None` values instead of raising an exception. In applications where data is incomplete, we may need to process rows in a simple, uniform way without having to write lots of distracting `if` statements to handle missing values.

Composite design

The common mathematical notation for a composite function looks as follows:

$$f \circ g(x) = f(g(x))$$

The idea is that we can define a new function, $f \circ g(x)$, that combines two other functions, $f(y)$ and $g(x)$.

Python's multiple-line definition of a composition function can be done through the following code:

```
@f_deco
def g(x):
    something
```

The resulting function can be essentially equivalent to $f \circ g(x)$. The `@f_deco` decorator must define and return the composite function by merging an internal definition of $f(y)$ with the provided base function, $g(x)$.

The implementation details show that Python actually provides a slightly more complex kind of composition. The structure of a wrapper makes it helpful to think of Python decorator composition as follows:

$$w \circ g(x) = (w_\beta \circ g \circ w_\alpha)(x) = w_\beta\Big(g(w_\alpha(x))\Big)$$

A decorator applied to some application function, $g(x)$, will include a wrapper function, $w(y)$, that has two parts. One portion of the wrapper, $w_\alpha(y)$, applies to the *arguments* of the base function; the other portion, $w_\beta(z)$, applies to the *result* of the base function.

Here's a slightly more concrete idea, shown as a `@stringify` decorator definition:

```
def stringify(argument_function: Callable[[int, int], int]) ->
Callable[[str], str]:
    @wraps(argument_function)
    def two_part_wrapper(text: str) -> str:
        # The "before" part
        arg1, arg2 = map(int, text.split(","))
        int_result = argument_function(arg1, arg2)
        # The "after" part
        return str(int_result)
    return two_part_wrapper
```

This decorator inserts conversions from string to integer, and integer back to string. Concealing the details of string processing may be helpful when working with CSV files, where the content is always string data.

We can apply this decorator to a function:

```
>>> @stringify
... def the_model(m: int, s: int) -> int:
...     return m * 45 + s * 3
...
>>> the_model("5,6")
'243'
```

This shows the two places to inject additional processing before as well as after the original function. This emphasizes an important distinction between the abstract concept of functional composition and the Python implementation: it's possible that a decorator can create either $f(g(x))$, or $g(f(x))$, or a more complex $f_\beta\Big(g(f_\alpha(x))\Big)$. The syntax of decoration doesn't describe which kind of composition will be created.

The real value of decorators stems from the way any Python statement can be used in the wrapping function. A decorator can use `if` or `for` statements to transform a function into something used conditionally or iteratively. In the next section, the examples will leverage the `try:` statement to perform an operation with a standard recovery from bad data. There

are many things that can be done within this general framework.

A great deal of functional programming follows the essential $f \circ g(x)$ design pattern. Defining a composite from two smaller functions can help to summarize complex processing. In other cases, it can be more informative to keep the two functions separate.

It's easy to create composites of the common higher-order functions, such as `map()`, `filter()`, and `functools.reduce()`. Because these functions are relatively simple, a composite function is often easy to describe, and can help to make the code more expressive.

For example, an application may include `map(f, map(g, x))`. It may be more clear to create a composite function and use a `map(f_g, x)` expression to describe applying a composite to a collection. We can use `f_g = lambda x: f(g(x))` to help explain a complex application as a composition of simpler functions. To make sure the type hints are correct, we'll almost always want to define individual functions with the `def` statement.

It's important to note that there's no real performance advantage to either technique. The `map()` function is lazy: with two `map()` functions, one item will be taken from the source collection, `x`, processed by the `g()` function, and then processed by the `f()` function. With a single `map()` function, an item will be taken from the source collection, `x`, and then processed by the `f_g()` composite function; the memory use is the same.

In *Chapter 13*, *The PyMonad Library*, we'll look at an alternative approach to this problem of creating composite functions from individual curried functions.

Preprocessing bad data

One cross-cutting concern in some exploratory data analysis applications is how to handle numeric values that are missing or cannot be parsed. We often have a mixture of `float`, `int`, `datetime.datetime`, and `decimal.Decimal` currency values that we'd like to process with some consistency.

In other contexts, we have *not applicable* or *not available* placeholders instead of data values; these shouldn't interfere with the main thread of the calculation. It's often handy to allow the *not applicable* values to pass through an expression without raising an exception. We'll

focus on three bad-data conversion functions: `bd_int()`, `bd_float()`, and `bd_decimal()`. We've left `bd_datetime()` as an exercise for the reader.

The composite feature we're adding will be defined first. Then we'll use this to wrap a built-in conversion function. Here's a simple bad-data decorator:

```
from collections.abc import Callable
import decimal
from typing import Any, Union, TypeVar, TypeAlias

Number: TypeAlias = Union[decimal.Decimal, float]
NumT = TypeVar("NumT", bound=Number)

def bad_data(
         function: Callable[[str], NumT]
) -> Callable[[str], NumT]:
    @wraps(function)
    def wrap_bad_data(source: str, **kwargs: Any) -> NumT:
        try:
            return function(source, **kwargs)
        except (ValueError, decimal.InvalidOperation):
            cleaned = source.replace(",", "")
            return function(cleaned, **kwargs)
    return wrap_bad_data
```

The decorator, `@bad_data`, wraps a given conversion function, with the parameter name `function`, to try a second conversion in the event the first conversion fails. The `ValueError` and `decimal.InvalidOperation` exceptions are generally indicators of data that has an invalid format: bad data. The second conversion will be attempted after `","` characters are removed. This wrapper passes the `*args` and `**kwargs` parameters into the wrapped function. This ensures that the wrapped functions can have additional argument values provided.

The type variable `NumT` is bound to the original return type of the base function being wrapped, the value of the `function` parameter. The decorator is defined to return a function with the same type, `NumT`. This type has an upper bound of the union of `float` and `Decimal`

types. This boundary permits objects that are a subclass of `float` or `Decimal`.

The type hints for complex decorator design are evolving rapidly. In particular, PEP 612 (`https://peps.python.org/pep-0612/`) defines some new constructs that can allow even more flexible type hints. For decorators that do not make any type changes, we can use generic parameter variables like `ParamSpec` to capture the actual parameters of the function being decorated. This lets us write generic decorators without having to wrestle with the details of the type hints of the functions being decorated. We'll note where PEP 612's `ParamSpec` and `Concatenate` will come in useful. Be sure to see the PEP 612 examples when designing generic decorators.

We can use this wrapper to create bad-data-sensitive functions as follows:

```
from decimal import Decimal

bd_int = bad_data(int)
bd_float = bad_data(float)
bd_decimal = bad_data(Decimal)
```

This will create a suite of functions that can do conversions of good data as well as a limited amount of data cleansing to handle specific kinds of bad data.

It can be difficult to write type hints for some kinds of callable objects. For example, the `int()` function has optional keyword arguments, with their own complex type hints. Our decorator summarizes these keyword arguments as `**kwargs: Any`. Ideally, a `ParamSpec` can be used to capture the details of the parameters for the function being wrapped. See PEP 612 (`https://peps.python.org/pep-0612/`) for guidance on creating complex type signatures for callable objects.

The following are some examples of using the `bd_int()` function:

```
>>> bd_int("13")
13
>>> bd_int("1,371")
1371
>>> bd_int("1,371", base=16)
4977
```

We've applied the `bd_int()` function to a string that converted neatly and a string with the specific type of punctuation that we'll tolerate. We've also shown that we can provide additional parameters to each of these conversion functions.

We may like to have a more flexible decorator. One feature that we may like to add is the ability to handle a variety of data scrubbing alternatives. Simple `","` removal isn't always what we need. We may also need to remove `$` or `°` symbols, too. We'll look at more sophisticated, parameterized decorators in the next section.

Adding a parameter to a decorator

A common requirement is to customize a decorator with additional parameters. Rather than simply creating a composite $f \circ g(x)$, we can do something a bit more complex. With parameterized decorators, we can create $\Big(f(c) \circ g\Big)(x)$. We've applied a parameter, c, as part of creating the wrapper, $f(c)$. This parameterized composite function, $f(c) \circ g$, can then be applied to the actual data, x.

In Python syntax, we can write it as follows:

```
@deco(arg)
def func(x):
    base function processing...
```

There are two steps to this. The first step applies the parameter to an abstract decorator to create a concrete decorator. Then the concrete decorator, the parameterized `deco(arg)` function, is applied to the base function definition to create the decorated function.

The effect is as follows:

```
concrete_deco = deco(arg)

def func(x):
    base function processing...

func = concrete_deco(func)
```

The parameterized decorator worked by doing the following three things:

1. Applied the abstract decorator, `deco()`, to its argument, `arg`, to create a concrete decorator, `concrete_deco()`.
2. Defined the base function, `func()`.
3. Applied the concrete decorator, `concrete_deco()`, to the base function to create the decorated version of the function; in effect, it's `deco(arg)(func)`.

A decorator with arguments involves indirect construction of the final function. We seem to have moved beyond merely higher-order functions into something even more abstract: higher-order functions that create higher-order functions.

We can expand our bad-data-aware decorator to create a slightly more flexible conversion. We'll define a `@bad_char_remove` decorator that can accept parameters of characters to remove. The following is a parameterized decorator:

```
from collections.abc import Callable
import decimal
from typing import Any, TypeVar

T = TypeVar('T')

def bad_char_remove(
    *bad_chars: str
) -> Callable[[Callable[[str], T]], Callable[[str], T]]:
    def cr_decorator(
            function: Callable[[str], T]
```

```
    ) -> Callable[[str], T]:
        def clean_list(text: str, *, to_replace: tuple[str, ...]) ->
        str:
            if to_replace:
                return clean_list(
                    text.replace(to_replace[0], ""),
                    to_replace=to_replace[1:]
                )
            return text

        @wraps(function)
        def wrap_char_remove(text: str, **kwargs: Any) -> T:
            try:
                return function(text, **kwargs)
            except (ValueError, decimal.InvalidOperation):
                cleaned = clean_list(text, to_replace=bad_chars)
                return function(cleaned, **kwargs)
        return wrap_char_remove
    return cr_decorator
```

A parameterized decorator has two internal function definitions:

- The concrete decorator; in this example, the `cr_decorator()` function. This will have the free variable, `bad_chars`, bound to the function being built. The concrete decorator is then returned; it will later be applied to a base function. When applied, the decorator will return a new function wrapped inside the `wrap_char_remove()` function. This new `wrap_char_remove()` function has type hints with a type variable, `T`, that claim the wrapped function's type will be preserved by the new `wrap_char_remove()` function.
- The decorating wrapper, the `wrap_char_remove()` function in this example, will replace the original function with a wrapped version. Because of the `@wraps` decorator, the `__name__` (and other attributes) of the new function will be replaced with the name of the base function being wrapped.

The overall decorator, the `@bad_char_remove` function in this example, has the job of

binding the parameter, named `bad_chars`, to a function and returning the concrete decorator. The type hint clarifies the return value is a `Callable` object that transforms a `Callable` function into another `Callable` function. The language rules will then apply the concrete decorator to the following function definition.

The internal `clean_list()` function is used by the `@bad_char_remove` decorator to remove all characters in a given argument value. This is defined as a recursion to keep it very short. It can be optimized into an iteration if necessary. We've left that optimization as an exercise for the reader.

We can use the `@bad_char_remove` decorator to create conversion functions as follows:

```
from decimal import Decimal
from typing import Any

@bad_char_remove("$", ",")
def currency(text: str, **kw: Any) -> Decimal:
    return Decimal(text, **kw)
```

We've used our `@bad_char_remove` decorator to wrap a base `currency()` function. The essential feature of the `currency()` function is a reference to the `decimal.Decimal` constructor.

This `currency()` function will now handle some variant data formats:

```
>>> currency("13")
Decimal('13')
>>> currency("$3.14")
Decimal('3.14')
>>> currency("$1,701.00")
Decimal('1701.00')
```

We can now process input data using a relatively simple `map(currency, row)` expression to convert source data from strings to usable `Decimal` values. The `try:/except:` error-handling has been isolated to a function that we've used to build a composite conversion

function.

We can use a similar design to create null-tolerant functions. These functions would use a similar `try:/except:` wrapper, but would return the `None` values. This design variant is left as an exercise for the reader.

This decorator is limited to conversion functions that apply to a single string, and have a type hint like `Callable[[str], T]`. For generic decorators, it helps to follow the examples in PEP-612 and use the `ParamSpec` and `Concatenate` type hints to broaden the domain of the decorators. Because we're interested in applying the internal `clean_list()` function to the first argument value, we can look at the conversion function as `Callable[Concatenate[str, P], T]`. We would define the first parameter as a string, and use a `ParamSpec`, `P`, to represent all other parameters of the conversion function.

Implementing more complex decorators

To create more complex compositions, Python allows the following kinds of function definitions:

```
@f_wrap
@g_wrap
def h(x):
    return something...
```

Python permits stacking decorators that modify the results of other decorators. This has a meaning somewhat like $f \circ g \circ h(x)$. However, the resulting name will be merely $h(x)$, concealing the stack of decorations. Because of this potential confusion, we need to be cautious when creating functions that involve deeply nested decorators. If our intent is simply to handle some cross-cutting concerns, then each decorator should be designed to handle a separate concern while avoiding confusion.

While many things can be done with decoration, it's essential to ask if using a decorator creates clear, succinct, expressive programming. When working with cross-cutting concerns, the features of the decorator are often essentially distinct from the function being

decorated. This can be a wonderful simplification. Adding logging, debugging, or security checks through decoration is a widely followed practice.

One important consequence of an overly complex design is the difficulty in providing appropriate type hints. When the type hints devolve to simply using `Callable[..., Any]`, the design may have become too difficult to reason about clearly.

Complicated design considerations

In the case of our data cleanup, the simplistic removal of stray characters may not be sufficient. When working with the geolocation data, we may have a wide variety of input formats that include simple degrees (`37.549016197`), degrees and minutes (`37° 32.94097′`), and degrees-minutes-seconds (`37° 32′ 56.46″`). Of course, there can be even more subtle cleaning problems: some devices will create an output with the Unicode `U+00BA` character, `º`, the "masculine ordinal indicator," instead of the similar-looking degree character, `°`, which is `U+00B0`.

For this reason, it is often necessary to provide a separate cleansing function that's bundled with the conversion function. This function will handle the more sophisticated conversions required by inputs that are as wildly inconsistent in format as latitudes and longitudes are.

How can we implement this? We have a number of choices. Simple higher-order functions are a good choice. A decorator, on the other hand, doesn't work out terribly well. We'll look at a decorator-based design to see some limitations to what makes sense in a decorator.

The requirements have the following two orthogonal design considerations:

- The output conversion from `string` to `int`, `float` or `Decimal`, summarized as `Callable[str, T]`
- The input cleaning; removing stray characters, reformatting coordinates; summarized as `Callable[str, str]`

Ideally, one of these aspects could be considered as the **essential** function that gets wrapped,

and the other aspect is something that's included via a decoration. The choice of essence versus wrapper isn't always clear.

Considering the previous examples, it appears that this should be seen as a three-part composite:

- The output conversion from `string` to `int`, `float`, or `decimal`
- The input cleansing: either a simple replace or a more complex multiple-character replacement
- An overall processing function that first attempts the conversion, then does any cleansing as a response to an exception, and then attempts the conversion again

The third part—attempting the conversion and retrying—is the actual wrapper that also forms a part of the composite function. As we noted previously, a wrapper contains an argument phase and a return-value phase, which we can call $w_\alpha(x)$ and $w_\beta(x)$, respectively.

We want to use this wrapper to create a composite of two additional functions. We have two choices for the design. We could include the cleansing function as an argument to the decorator on the conversion, as follows:

```
@cleanse_before(cleanser)
def convert(text: str) -> int:
    # code to convert the text, trusting it was clean
    return # an int value
```

This first design claims that the conversion function is central, and the cleansing is an ancillary detail that will modify the behavior but preserve the original intent of the conversion.

Or, we could include the conversion function as an argument to the decorator for a cleansing function as follows:

```
@then_convert(converter)
def cleanse(text: str) -> str:
    # code to clean the text
    return # the str value for later conversion
```

This second design claims that the cleansing is central and the conversion is an ancillary detail. This is a bit confusing because the cleansing type is generally `Callable[[str], str]`, while the conversion's type of `Callable[[str], some other type]` is what is required for the overall wrapped function.

While both of these approaches can create a usable composite function, the first version has an important advantage: the type signature of the `conversion()` function is also the type signature of the resulting composite function. This highlights a general design pattern for decorators: the type annotations—the signatures—of the function being decorated are the easiest to preserve.

When confronted with several choices for defining a composite function, it is generally easiest to preserve the type hints for the function being decorated. This helps identify the concept that's central.

Consequently, the `@cleanse_before(cleaner)` style decorator is preferred. The decorator definition looks like the following example:

```
from collections.abc import Callable
from typing import Any, TypeVar

# Defined Earlier:
# T = TypeVar('T')

def cleanse_before(
    cleanse_function: Callable[[str], Any]
) -> Callable[[Callable[[str], T]], Callable[[str], T]]:
    def concrete_decorator(converter: Callable[[str], T]) ->
    Callable[[str], T]:
        @wraps(converter)
        def cc_wrapper(text: str, **kwargs: Any) -> T:
            try:
                return converter(text, **kwargs)
            except (ValueError, decimal.InvalidOperation):
                cleaned = cleanse_function(text)
```

```
                return converter(cleaned, **kwargs)
        return cc_wrapper
    return concrete_decorator
```

We've defined the following multi-layer decorator:

- At the heart is the `cc_wrapper()` function that applies the `converter()` function. If this fails, then it uses the given `cleanse_function()` function and then tries the `converter()` function again.
- The `cc_wrapper()` function is built around the given `cleanse_function()` and a `converter()` function by the `concrete_decorator()` decorator. The `converter()` function is the function being decorated.
- The top-most layer is the `concrete_decorator()` function. This decorator has the `cleanse_function()` function as a free variable.
- The concrete decorator is created when the decorator interface, `cleanse_before()`, is evaluated. The interface is customized by providing the `cleanse_function` as an argument value.

The type hints emphasize the role of the `@cleanse_before` decorator. It expects some `Callable` function, named `cleanse_function`, and it creates a function, shown as `Callable[[str], T]`, which will transform a function into a wrapped function. This is a helpful reminder of how parameterized decorators work.

We can now build a slightly more flexible cleanse and convert function, `to_int()`, as follows:

```
def drop_punct2(text: str) -> str:
    return text.replace(",", "").replace("$", "")

@cleanse_before(drop_punct2)
def to_int(text: str, base: int = 10) -> int:
    return int(text, base)
```

The integer conversion is decorated with a cleansing function. In this case, the cleansing function removes `$` and `,` characters. The integer conversion is wrapped by this cleansing.

The `to_int()` function defined previously leverages the built-in `int()` function. An alternative definition that avoids the `def` statement would be the following:

```
to_int2 = cleanse_before(drop_punct2)(int)
```

This uses `drop_punct2()` to wrap the built-in `int()` conversion function. Using the **mypy** tool's `reveal_type()` function shows that the type signature for `to_int()` matches the type signature for the built-in `int()`. It can be argued that this style is less readable than using a decorator.

We can use this enhanced integer conversion as follows:

```
>>> to_int("1,701")
1701
>>> to_int("42")
42
```

The type hints for the underlying `int()` function have been rewritten (and simplified) for the decorated function, `to_int()`. This is a consequence of trying to use decorators to wrap built-in functions.

Because of the complexity of defining parameterized decorators, it appears that this is the edge of the envelope. The decorator model doesn't seem to be ideal for this kind of design. It seems like a definition of a composite function would be more clear than the machinery required to build decorators.

The alternative is to duplicate a few lines of code that will be the same for all of the conversion functions. We could use:

```
def to_int_flat(text: str, base: int = 10) -> int:
    try:
        return int(text, base)
    except (ValueError, decimal.InvalidOperation):
        cleaned = drop_punct2(text)
        return int(cleaned, base)
```

Each of the data type conversions will repeat the `try-except` block. The use of a decorator isolates this design feature in a way that can be applied to any number of conversion functions without explicitly restating the code. Later changes to the design when using this alternative may require editing a number of similar functions instead of changing one decorator.

Generally, decorators work well when we have a number of relatively simple and fixed aspects that we want to include with a given function (or a class). Decorators are also important when these additional aspects can be looked at as infrastructure or as support, and not something essential to the meaning of the application code.

For something that involves multiple orthogonal design aspects, we may want to result to a callable class definition with various kinds of plugin strategy objects. This might have a simpler class definition than the equivalent decorator. Another alternative to decorators is to look closely at creating higher-order functions. In some cases, partial functions with various combinations of parameters may be simpler than a decorator.

The typical examples for cross-cutting concerns include logging or security testing. These features can be considered as the kind of background processing that isn't specific to the problem domain. When we have processing that is as ubiquitous as the air that surrounds us, then a decorator might be an appropriate design technique.

Summary

In this chapter, we've looked at two kinds of decorators: simple decorators with no arguments and parameterized decorators. We've seen how decorators involve an indirect

composition between functions: the decorator wraps a function (defined inside the decorator) around another function.

Using the `functools.wraps()` decorator ensures that our decorators will properly copy attributes from the function being wrapped. This should be a piece of every decorator we write.

In the next chapter, we'll look at the PyMonad library to express a functional programming concept directly in Python. We don't require monads generally because Python is an imperative programming language under the hood.

Exercises

This chapter's exercises are based on code available from Packt Publishing on GitHub. See `https://github.com/PacktPublishing/Functional-Python-Programming-3rd-Edition`.

In some cases, the reader will notice that the code provided on GitHub includes partial solutions to some of the exercises. These serve as hints, allowing the reader to explore alternative solutions.

In many cases, exercises will need unit test cases to confirm they actually solve the problem. These are often identical to the unit test cases already provided in the GitHub repository. The reader should replace the book's example function name with their own solution to confirm that it works.

Datetime conversions

In the *Preprocessing bad data* section of this chapter, we introduced the concept of data conversion functions that included special *not applicable* or *not available* data values. These are often called null values; because of this, a database may have a universal `NULL` literal. We'll call them "bad data" because that's how we often discover them. When examining data for the first time, we find bad data that might represent missing, or not applicable, values.

This kind of data can have any of these possible processing paths:

- The bad data is silently ignored; it's excluded from counts and averages. To make this work out, we'll often want to replace bad values with a consistent object. The `None` object is a good replacement value.
- The bad data stops the processing, raising an exception. This is quite easy to implement, since Python tends to do this automatically. In some cases, we want to retry the conversion using alternative rules. We'll focus on this approach for this exercise.
- Bad data is replaced with interpolated or imputed values. This often means keeping two versions of a collection of data: the original with bad data present, and a more useful version with replacement values. This isn't a simple computation.

The idea of our core `bad_data()` function is to try a conversion, replace known bad punctuation, and then try again. We might, for example, strip "," and "`$`" from numeric values.

Earlier in this chapter, we described three bad-data conversion functions: `bd_int()`, `bd_float()`, and `bd_decimal()`. Each of these performed a relatively direct conversion-or-replacement algorithm. We left the `bd_datetime()` function as an exercise for the reader. In this case, the alternative date formats can lead to a bit more complexity.

We'll assume that dates must be in one of three formats: "yyyy-mon-dd", "yyyy-mm-dd", or "mon-dd" without a year. In the first and third formats, the month name is spelled out. In the second format, the month name is numeric. These are handled by the `datetime.strptime()` function using format strings like `"%Y-%b-%d"`, `"%b-%d"`, and `"%Y-%m-%d"`.

Write a `bd_datetime()` function to try multiple data format conversions, looking for one that produces a valid date. In the case of a missing year, the `datetime.replace()` method can be used to build a final date result with the current year.

Once the basic implementation is complete, create appropriate test cases with a mix of valid and invalid dates.

Be sure to make the design flexible enough that adding another format can be done without

too much struggle.

Optimize a decorator

In the *Adding a parameter to a decorator* section of this chapter, we defined a decorator to replace "bad" characters in a given field and retry an attempted conversion.

This decorator had an internal function, `clean_list()`, that provided a recursive definition for removing bad characters from a string.

Here's the Python function definition:

```
    def clean_list(text: str, *, to_replace: tuple[str, ...]) -> str:
        ...
```

This recursion has two cases:

- When the `to_replace` argument value is empty, there's nothing to replace, and the value of the `text` parameter is returned unchanged.
- Otherwise, split the `to_replace` string to separate the first character from the remaining characters. Remove any occurrence of the first character from the value of the `text` parameter and apply this function again using the remaining characters of the `to_replace` string.

Looking back at *Chapter 6, Recursions and Reductions*, we recall that this kind of tail-call recursion can be transformed into a `for` statement. Rewrite the `clean_list()` function to eliminate the recursion.

None-tolerant functions

In the *Adding a parameter to a decorator* section of this chapter, we saw a design pattern of using a `try:/except:` wrapper to uncover numbers with spurious punctuation marks. A similar technique can be used to detect `None` values and pass them through a function, unprocessed.

Write a decorator that can be used for `Callable[[float], float]` functions that will handle `None` values gracefully.

If the none-tolerant decorator is called `@none_tolerant`, here is a test case:

```
@none_tolerant
def x2(x: float) -> float:
    return 2 * x

def test_x2() -> None:
    assert x2(42.) == 84.0
    assert x2(None) == None
    assert list(map(x2, [1, 2, None, 3])) == [2, 3, None, 6]
```

Logging

A common requirement for debugging is a consistent collection of logging messages. It can become tedious to include a `logger.debug()` line in a number of closely-related functions. If the functions have a consistent set of type definitions, it can be helpful to define a decorator that can be applied to a number of related functions.

As example functions, we'll define a collection of "models" that compute an expected result from sample values. We'll start with a dataclass to define each sample as having an identifier, an observed value, and a time-stamp. It looks like this:

```
from dataclasses import dataclass
@dataclass(frozen=True)
class Sample:
    id: int
    observation: float
    date_time: datetime.datetime
```

We have three models to compute an expected value, e, from the observed value in the sample, s_o:

- $e = 0.7412 \times s_o$

- $e = 0.9 \times s_o - 90$
- $e = 0.7724 \times s_o^{1.0134}$

First, define these three functions with appropriate test cases.

Second, define a `@logging` decorator to use `logger.info()` to record the sample value and the computed expectation.

Third, add the `@logging` decorator in front of each function definition.

Create an overall application that uses `logging.basicConfig()` to set the logging level to `logging.INFO` to ensure that the informational messages are seen. (The default logging level only shows warnings and errors.)

This permits creating a consistent logging setup for the three "model" functions. This reflects a complete separation between the logging aspect of the application and the computation of expected values from sample values. Is this separation clear and helpful? Are there circumstances where this separation might not be desirable?

The actual measurements are given here. One of the models is more accurate than the others:

Sample Number	Observed	Actual
1	1000	883
2	1500	1242
3	1500	1217
4	1600	1306
5	1750	1534
6	2000	1805
7	2000	1720

Dry-run check

Applications that can modify the file system require extensive unit testing as well as integration testing. To mitigate risk even further, these applications will often have a

"dry-run" mode where file system modifications are logged but not carried out; files are not moved, directories are not deleted, and so on.

The idea here is to write small functions for file system state changes. Each function can then be decorated with a `@dry_run_check` decorator. This decorator can examine a global variable, `DRY_RUN`. The decorator writes a log message. If the `DRY_RUN` value is `True`, nothing else is done. If the `DRY_RUN` value is `False`, the base function is evaluated to make the underlying state changes, such as removing files, or removing directories.

First, define a number of functions to copy a directory. The following state changes need separate functions:

- Create a new, empty, directory.
- Copy a file from somewhere in the source directory to the target directory. We can use an expression like `offset = source_path.relative_to(source_dir)` to compute the relative location of a file in the source directory. We can use `target_dir / offset` to compute the new location in a target directory. The `pathlib.Path` objects provide all of the features required.

The `pathlib.Path.glob()` method provides a useful view of a directory's content. This can be used by an overall function that calls the other two functions to create subdirectories and copy files into them.

Second, define a decorator to block the action if this is a dry run. Apply the decorator to the directory creation function and the file copy function. Note that these two function signatures are different. One function uses a single path, the other function uses two paths.

Third, create a suitable unit test to confirm that dry-run mode goes through the motions, but doesn't alter the underlying file system. The `pytest.tmp_path` fixture provides a temporary working directory; using this prevents endlessly having to drop and recreate output directories while debugging.

13

The PyMonad Library

A monad allows us to impose an order on an expression evaluation in an otherwise lenient language. We can use a monad to insist that an expression such as `a + b + c` is evaluated in left-to-right order. This can interfere with the compiler's ability to optimize expression evaluation. This is necessary, however, when we want files to have their content read or written in a specific order: a monad is a way to assure that the `read()` and `write()` functions are evaluated in a particular order.

Languages that are lenient and have optimizing compilers benefit from monads imposing order on the evaluation of expressions. Python, for the most part, is strict and does not optimize, meaning there are few practical requirements for monads in Python.

While the PyMonad package contains a variety of monads and other functional tools, much of the package was designed to help folks understand functional programming using Python syntax. We'll focus on a few features to help clarify this point of view.

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

- Downloading and installing PyMonad

- The idea of currying and how this applies to functional composition
- The PyMonad star operator for creating composite functions
- Functors and techniques for currying data items with more generalized functions
- The `bind()` operation, using the Python `>>` operator, to create ordered monads
- We'll also explain how to build a Markov chain simulation using PyMonad techniques

What's important is that Python doesn't require the use of monads. In many cases, the reader will be able to rewrite the example using pure Python constructs. Doing this kind of rewrite can help solidify one's understanding of functional programming.

Downloading and installing

The PyMonad package is available on the **Python Package Index** (**PyPI**). In order to add PyMonad to your environment, you'll need to use the `python -m pip pymonad` command to install it.

This book used version 2.4.0 to test all of the examples. Visit `https://pypi.python.org/pypi/PyMonad` for more information.

Once the PyMonad package is installed, you can confirm it using the following commands:

```
>>> import pymonad
>>> help(pymonad)
```

This will display the module's docstring and confirm that things really are properly installed.

The overall project name, PyMonad, uses mixed case. The installed Python package name that we import, `pymonad`, is all lower case.

Functional composition and currying

Some functional languages work by transforming a multi-argument function syntax into a collection of single argument functions. This process is called **currying**: it's named after logician Haskell Curry, who developed the theory from earlier concepts. We've looked

at currying in depth in *Chapter 11, The Toolz Package*. We'll revisit it from the PyMonad perspective here.

Currying is a technique for transforming a multi-argument function into higher-order single argument functions. In a simple case, consider a function $f(x, y) \longrightarrow z$; given two arguments x and y; this will return some resulting value, z. We can curry the function $f(x, y)$ into into two functions: $f_{c1}(x) \longrightarrow f_{c2}(y)$ and $f_{c2}(y) \longrightarrow z$. Given the first argument value, x, evaluating the function $f_{c1}(x)$ returns a new one-argument function, $f_{c2}(y)$. This second function can be given the second argument value, y, and it returns the desired result, z.

We can evaluate a curried function in Python with concrete argument values as follows: `f_c1(2)(3)`. We apply the curried function to the first argument value of 2, creating a new function. Then, we apply that new function to the second argument value of 3.

Let's look at a concrete example in Python. For example, we have a function like the following one:

```
from pymonad.tools import curry  # type: ignore[import]

@curry(4)  # type: ignore[misc]
def systolic_bp(
    bmi: float, age: float, gender_male: float, treatment: float
) -> float:
    return (
        68.15 + 0.58 * bmi + 0.65 * age + 0.94 * gender_male + 6.44 *
        treatment
    )
```

This is a simple, multiple-regression-based model for systolic blood pressure. This predicts blood pressure from body mass index (BMI), age, gender (a value of 1 means male), and history of previous treatment (a value of 1 means previously treated). For more information on the model and how it's derived, visit `http://sphweb.bumc.bu.edu/otlt/MPH-Modules/BS/BS704_Multivariable/BS704_Multivariable7.html`.

We can use the `systolic_bp()` function with all four arguments, as follows:

```
>>> systolic_bp(25, 50, 1, 0)
116.09

>>> systolic_bp(25, 50, 0, 1)
121.59
```

A male person with a BMI of 25, age 50, and no previous treatment is predicted to have a blood pressure near 116. The second example shows a similar woman with a history of treatment who will likely have a blood pressure of 121.

Because we've used the `@curry` decorator, we can create intermediate results that are similar to partially applied functions. Take a look at the following command snippet that creates a new function, `treated()`:

```
>>> treated = systolic_bp(25, 50, 0)
>>> treated(0)
115.15
>>> treated(1)
121.59
```

In the preceding case, we evaluated the `systolic_bp(25, 50, 0)` expression to create a curried function and assigned this to the `treated` variable. This built a new function, `treated`, with values for some of the parameters. The BMI, age, and gender values don't typically change for a given patient. We can now apply the new `treated()` function to the remaining argument value to get different blood pressure expectations based on patient history.

Here's an example of creating some additional curried functions:

```
>>> g_t = systolic_bp(25, 50)
>>> g_t(1, 0)
116.09
```

```
>>> g_t(0, 1)
121.59
```

This is a gender-based treatment function based on our initial model. We must provide both the needed gender and treatment argument values to get a final value from the model.

This is similar in some respects to the `functools.partial()` function. The important difference is that currying creates a function that can work in a variety of ways. The `functools.partial()` function creates a more specialized function that can only be used with the given set of bound values. For more information, see *Chapter 10, The Functools Module*.

Using curried higher-order functions

An important application of currying shows up when we use it on higher-order functions. We can, for example, curry the `reduce` function, as follows:

```
>>> from pymonad.tools import curry
>>> from functools import reduce

>>> creduce = curry(2, reduce)
```

The `creduce()` function is a curried function; we can now use it to create functions by providing some of the required argument values. In the next example, we will use `operator.add` as one of the two argument values to reduce. We can create a new function, and assign this to `my_sum`.

We can create and use this new `my_sum()` function as shown in the following example:

```
>>> from operator import add

>>> my_sum = creduce(add)
>>> my_sum([1,2,3])
6
```

We can also use our curried `creduce()` function with other binary operators to create other reductions. The following shows how to create a reduction function that finds the maximum value in a sequence:

```
>>> my_max = creduce(lambda x,y: x if x > y else y)
>>> my_max([2,5,3])
5
```

We defined our own version of the default `max()` function using a lambda object that picks the larger of two values. We could use the built-in `max()` function for this. More usefully, we could use more sophisticated comparisons among items to locate a local maxima. For geofencing applications, we might have a maximum east-west function separate from a maximum north-south function.

We can't easily create the more general form of the `max()` function using the PyMonad `curry()` function. This implementation is focused on positional parameters. Trying to use the `key=` keyword parameter adds too much complexity to make the technique work toward our overall goals of succinct and expressive functional programs.

The built-in reductions including the `max()`, `min()`, and `sorted()` functions all rely on an optional `key=` keyword parameter paradigm. Creating curried versions means we need variants of these that accept a function as the first argument in the same way as the `filter()`, `map()`, and `reduce()` functions do. We could also create our own library of more consistent higher-order curried functions. These functions would rely exclusively on positional parameters, and follow the pattern of providing the function first and the values last.

Functional composition with PyMonad

One of the significant benefits of using curried functions is the ability to combine them through functional composition. We looked at functional composition in *Chapter 5, Higher-Order Functions*, and *Chapter 12, Decorator Design Techniques*.

When we've created a curried function, we can more easily perform function composition

to create a new, more complex curried function. In this case, the PyMonad package defines the * operator for composing two functions. To explain how this works, we'll define two curried functions that we can compose. First, we'll define a function that computes the product, and then we'll define a function that computes a specialized range of values.

Here's our first function, which computes the product:

```
import operator

prod = creduce(operator.mul)
```

This is based on our curried `creduce()` function that was defined previously. It uses the `operator.mul()` function to compute a *times-reduction* of an iterable: we can call a product a times-reduce of a sequence.

Here's our second curried function that will produce a range of even or odd values:

```
from collections.abc import Iterable

@curry(1)  # type: ignore[misc]
def alt_range(n: int) -> Iterable[int]:
    if n == 0:
        return range(1, 2)  # Only the value [1]
    elif n % 2 == 0:
        return range(2, n+1, 2)  # Even
    else:
        return range(1, n+1, 2)  # Odd
```

The result of the `alt_range()` function will be even values or odd values. It will have only odd values up to (and including) `n`, if `n` is odd. If `n` is even, it will have only even values up to `n`. The sequences are important for implementing the semifactorial or double factorial function, $n!!$.

Here's how we can combine the `prod()` and `alt_range()` functions to compute a result:

```
>>> prod(alt_range(9))
945
```

One very interesting use of curried functions is the idea of creating a composition of those functions that can be applied to argument values. The PyMonad package provides operators for this, but they can be confusing. What seems better is making use of the `Compose` subclass of `Monad`.

We can use `Compose` to implement functional composition in a direct way. The following example shows how we can compose our `alt_range()` and `prod()` functions to compute the semifactorial:

```
>>> from pymonad.reader import Compose
>>> semi_fact = Compose(alt_range).then(prod)
>>> semi_fact(9)
945
```

We've built a `Compose` monad from the `alt_range()` function composed with the `prod()` function. The resulting function can be applied to an argument value to compute a result from the composition of the two functions.

Using curried functions can help to clarify a complex computation by eliding some of the argument-passing details.

Note that the `then()` method imposes a strict ordering: first, compute the range. Once that is done, use the result to compute the final product.

Functors – making everything a function

The idea of a **functor** is a functional representation of a piece of simple data. A functor version of the number 3.14 is a function of zero arguments that returns this value. Consider the following example:

```
>>> pi = lambda: 3.14
>>> pi()
3.14
```

We created a zero-argument lambda object that returns a Python float object.

When we apply a curried function to a functor, we're creating a new curried functor. This generalizes the idea of applying a function to an argument to get a value by using functions to represent the arguments, the values, and the functions themselves.

Once everything in our program is a function, then all processing becomes a variation on the theme of functional composition. To recover the underlying Python object, we can use the `value` attribute of a functor object to get a Python-friendly, simple type that we can use in uncurried code.

Since this kind of programming is based on functional composition, no calculation needs to be done until we actually demand a value using the `value` attribute. Instead of performing a lot of intermediate calculations, our program defines intermediate complex objects that can produce a value when requested. In principle, this composition can be optimized by a clever compiler or runtime system.

In order to work politely with functions that have multiple arguments, PyMonad offers a `to_arguments()` method. This is a handy way to clarify the argument value being provided to a curried function. We'll see an example of this below, after introducing the `Maybe` and `Just` monads.

We can wrap a Python object with a subclass of the `Maybe` monad. The `Maybe` monad is interesting, because it gives us a way to deal gracefully with missing data. The approach we used in *Chapter 12*, *Decorator Design Techniques*, was to decorate built-in functions to make them `None`-aware. The approach taken by the PyMonad library is to decorate the data to distinguish something that's *just an object* from *nothing*.

There are two subclasses of the `Maybe` monad:

- `Nothing`
- `Just(` *some Python object* `)`

We use `Nothing` similarly to the Python value of `None`. This is how we represent missing data. We use `Just()` to wrap all other Python objects. These are also functors, offering function-like representations of constant values.

We can use a curried function with these `Maybe` objects to tolerate missing data gracefully. Here's a short example:

```
>>> from pymonad.maybe import Maybe, Just, Nothing

>>> x1 = Maybe.apply(systolic_bp).to_arguments(Just(25), Just(50),
Just(1), Just(0))
>>> x1.value
116.09

>>> x2 = Maybe.apply(systolic_bp).to_arguments(Just(25), Just(50),
Just(1), Nothing)
>>> x2
Nothing
>>> x2.value is None
True
```

This shows us how a monad can provide an answer instead of raising a `TypeError` exception. This can be very handy when working with large, complex datasets in which data could be missing or invalid.

We must use the `value` attribute to extract the simple Python value for uncurried Python code.

Using the lazy ListMonad() monad

The `ListMonad()` monad can be confusing at first. It's extremely lazy, unlike Python's built-in list type. When we evaluate the a `list(range(10))` expression, the `list()` function

will evaluate the `range()` object to create a list with 10 items. The PyMonad `ListMonad()` monad, however, is too lazy to even do this evaluation.

Here's the comparison:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> from pymonad.list import ListMonad
>>> ListMonad(range(10))
[range(0, 10)]
```

The `ListMonad()` monad did not evaluate the `range()` object's iterable sequence of values; it preserved it without being evaluated. A `ListMonad()` monad is useful for collecting functions without evaluating them.

We can evaluate the `ListMonad()` monad later as required:

```
>>> from pymonad.list import ListMonad

>>> x = ListMonad(range(10))
>>> x
[range(0, 10)]
>>> x[0]
range(0, 10)
>>> list(x[0])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

We created a lazy `ListMonad()` object which contained a `range()` object. Then we extracted and evaluated a `range()` object at position 0 in that list.

A `ListMonad()` object won't evaluate a generator function. It treats any iterable argument as a single iterator object. We can, later, apply the function being contained by the monad.

Here's a curried version of the `range()` function. This has a lower bound of 1 instead of 0. It's handy for some mathematical work because it allows us to avoid the complexity of the

positional arguments in the built-in `range()` function:

```
from collections.abc import Iterator
from pymonad.tools import curry

@curry(1)  # type: ignore[misc]
def range1n(n: int) -> range:
    if n == 0: return range(1, 2)  # Only the value 1
    return range(1, n+1)
```

We wrapped the built-in `range()` function to make it curryable by the PyMonad package.

Since a `ListMonad` object is a functor, we can map functions to the `ListMonad` object. The function is applied to each item in the `ListMonad` object.

Here's an example:

```
>>> from pymonad.reader import Compose
>>> from pymonad.list import ListMonad

>>> fact = Compose(range1n).then(prod)
>>> seq1 = ListMonad(*range(20))

>>> f1 = seq1.map(fact)
>>> f1[:10]
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880]
```

We defined a composite function, `fact()`, which was built from the `prod()` and `range1n()` functions shown previously. This is the factorial function. We created a `ListMonad()` functor, `seq1`, which is a sequence of 20 values. We mapped the `fact()` function to the `seq1` functor, which created a sequence of factorial values, `f1`. Finally, we extracted the first 10 of these values.

Here's another little function that we'll use to extend this example:

```
from pymonad.tools import curry

@curry(1)  # type: ignore[misc]
def n21(n: int) -> int:
    return 2*n+1
```

This little `n21()` function does a simple computation. It's curried, however, so we can apply it to a functor such as a `ListMonad()` object. Here's the next part of the preceding example:

```
>>> semi_fact = Compose(alt_range).then(prod)
>>> f2 = seq1.map(n21).then(semi_fact)
>>> f2[:10]
[1, 3, 15, 105, 945, 10395, 135135, 2027025, 34459425, 654729075]
```

We've defined a composite function from the `prod()` and `alt_range()` functions shown previously. The value of the `f2` object is built by mapping our small `n21()` function applied to the `seq1` sequence. This creates a new sequence. We then applied the `semi_fact()` function to each object in this new sequence to create a sequence of values that are parallels to the `f1` sequence of values.

We can now map the `/` operator, `operator.truediv`, to these two parallel sequences of values, `f1` and `f2`:

```
>>> import operator
>>> 2 * sum(map(operator.truediv, f1, f2))
3.1415919276751456
```

The built-in `map()` function will apply the given operator to both functors, yielding a sequence of fractions that we can add.

We defined a fairly complex calculation using a few functional composition techniques and a functor class definition. This is based on a computation for the arctangent. Here's the

full definition for this calculation:

$$\pi = 2 \sum_{0 \leq n < \infty} \frac{n!}{(2n+1)!!}$$

Ideally, we prefer not to use a fixed-size `ListMonad` object with only twenty values. We'd prefer to have a lazy and potentially infinite sequence of integer values, allowing us an approximation of arbitrary accuracy. We could then use curried versions of the `sum()` and `takewhile()` functions to find the sum of values in the sequence until the values are too small to contribute to the result.

This rewrite to use the `takewhile()` function is left as an exercise for the reader.

Monad bind() function

The name of the PyMonad library comes from the functional programming concept of a monad, a function that has a strict order. The underlying assumption behind much functional programming is that functional evaluation is liberal: it can be optimized or rearranged as necessary. A monad provides an exception that imposes a strict left-to-right order.

Python, as we have seen, is already strict. It doesn't *require* monads. We can, however, still apply the concept in places where it can help clarify a complex algorithm. We'll look at an example, below, of using a monad-based approach to designing a simulation based on Markov chains.

The technology for imposing strict evaluation is a binding between a monad and a function that will return a monad. A *flat* expression will become nested bindings that can't be reordered by an optimizing compiler. The `then()` method of a monad imposes this strict ordering.

In other languages, such as Haskell, a monad is crucial for file input and output where strict ordering is required. Python's imperative mode is much like a Haskell `do` block, which has an implicit Haskell `>>=` operator to force the statements to be evaluated in order. PyMonad

uses the `then()` method for this binding.

Implementing simulation with monads

Monads are expected to pass through a kind of *pipeline*: a monad will be passed as an argument to a function and a similar monad will be returned as the value of the function. The functions must be designed to accept and return similar structures.

We'll look at a monad-based pipeline that can be used for simulation of a process. This kind of simulation is sometimes called a **Monte Carlo** simulation. In this case, the simulation will create a **Markov chain**.

A Markov chain is a model for a series of potential events. The probability of each event depends only on the state attained in the previous event. Each state of the overall system had a set of probabilities that define the events and related state changes. It fits well with games that involve random chance, like dice or cards. It also fits well with industrial processes where small random effects can "ripple through" the system, leading to effects that may not appear to be—directly—related to tiny initial problems.

Our example involves some rules for a fairly complex simulation. We can visualize the following state changes shown in *Figure 13.1* as creating a chain of events that ends with either a Pass or Fail event. The number of events has a lower bound of 1.

The state transition probabilities are stated as fractions, $\frac{n}{36}$, because this particular Markov chain generator comes from a game that uses two dice. There are 36 possible outcomes from a roll of the dice. When considering the sum of the two dice, the 10 distinct values have probabilities ranging from $\frac{1}{36}$ for the values 2 and 12, to $\frac{6}{36}$ for the value 7.

Because this is based on a game, the actual algorithm is somewhat simpler than the diagram of the state transitions. The trick in simplifying the algorithm description is combining a number of similar behaviors into a single state defined by a parameter, p.

The algorithm's use of an internal state. For designers new to functional programming, this is a bit of a problem. The solution that we've shown in other examples is to expose the state as a parameter to a function.

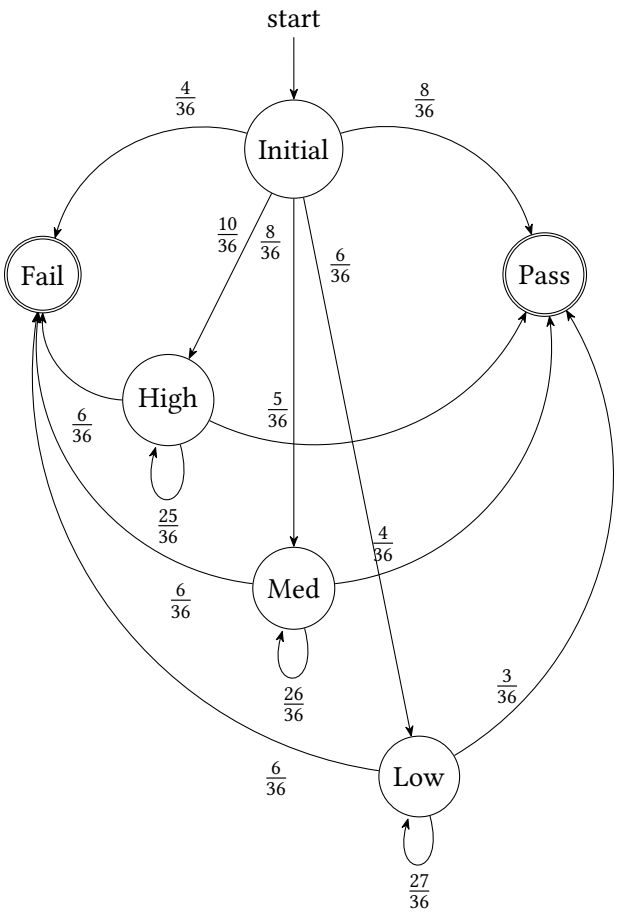


Figure 13.1: Markov chain generator

We'll start with a presentation of the algorithm with explicit state.

Algorithm 9 Markov chain generator

Ensure: Outcome is one of **Pass** or **Fail**

1: $d_1 \leftarrow$ **randint**$(1,6)$ + **randint**$(1,6)$ ▷ Roll two dice.
2: **if** $d_1 \in \{2,3,12\}$ **then** ▷ $P = \frac{4}{36}$
3: **return Fail**
4: **else if** $d_1 \in \{7,11\}$ **then** ▷ $P = \frac{8}{36}$
5: **return Pass**
6: **else** ▷ $P = \frac{24}{36}$
7: $p \leftarrow d_1$ ▷ The point to match.
8: **end if**
9: **repeat**
10: $d_n \leftarrow$ **randint**$(1,6)$ + **randint**$(1,6)$ ▷ Roll two dice.
11: **if** $d_n = 7$ **then** ▷ $P = \frac{6}{36}$
12: **return Fail**
13: **else if** $d_n = p$ **then** ▷ $P \in \{\frac{3}{36}, \frac{4}{36}, \frac{5}{36}\}$
14: **return Pass**
15: **else** ▷ $P \in \{\frac{27}{36}, \frac{26}{36}, \frac{25}{36}\}$
16: No outcome yet.
17: **end if**
18: **until** An outcome.

The algorithm can be seen as requiring a state change. Alternatively, we can look at this as a sequence of operations to append to the Markov chain, rather than a state change. There's one function that must be used first to create an initial outcome or establish the value of p. Another, recursive function is used after that to iterate until an outcome is determined. In this way, this pairs-of-functions approach fits the monad design pattern nicely.

To build Markov chains, we'll need a source of random numbers:

```
import random

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

from collections.abc import Callable
from typing import TypeAlias

DiceT: TypeAlias = Callable[[], tuple[int, int]]
```

The preceding function will generate a pair of dice for us. We also included a type hint, `DiceT`, that can be used to describe any similar function that returns a tuple with two integers. The type hint will be used in later functions as a shorthand for any similar random number generator.

Here's our expectations from the overall chain generator based on the game algorithm:

```
from pymonad.maybe import Maybe, Just

def game_chain(dice: DiceT) -> Maybe:
    outcome = (
        Just(("", 0, []))
        .then(initial_roll(dice))
        .then(point_roll(dice))
    )
    return outcome
```

We create an initial monad, `Just(("", 0, []))`, to define the essential type we're going to work with. A game will produce a three-tuple with the outcome text, the point value, and a sequence of rolls. At the start of each game, a default three-tuple establishes the three-tuple type.

We pass this monad to two other functions. This will create a resulting monad, `outcome`, with the results of the game. We use the `then()` method to connect the functions in the specific order they must be executed. In a language with an optimizing compiler, this will

prevent the expression from being rearranged.

We will get the value of the monad at the end using the `value` attribute. Since the monad objects are lazy, this request is what triggers the evaluation of the various monads to create the required output.

Each resulting sequence of three-tuples is a Markov chain we can analyze to determine the overall statistical properties. We're often interested in the expected lengths of the chains. This can be difficult to predict from the initial model or the algorithm.

The `initial_roll()` function has the `rng()` function curried as the first argument. The monad will become the second argument to this function. The `initial_roll()` function can roll the dice and apply the *come out* rule to determine if we have a pass, a fail, or a point.

The `point_roll()` function also has the `rng()` function curried as the first argument. The monad will become the second argument. The `point_roll()` function can then roll the dice to see if the game is resolved. If the game is unresolved, this function will operate recursively to continue looking for a resolution.

The `initial_roll()` function looks like this:

```
from pymonad.tools import curry
from pymonad.maybe import Maybe, Just

@curry(2)  # type: ignore[misc]
def initial_roll(dice: DiceT, status: Maybe) -> Maybe:
    d = dice()
    if sum(d) in (7, 11):
        return Just(("pass", sum(d), [d]))
    elif sum(d) in (2, 3, 12):
        return Just(("fail", sum(d), [d]))
    else:
        return Just(("point", sum(d), [d]))
```

The dice are rolled once to determine if the initial outcome is pass, fail, or establish the

point. We return an appropriate monad value that includes the outcome, a point value, and the roll of the dice that led to this state. The point values for an immediate pass and immediate fail aren't really meaningful. We could sensibly return a 0 value here, since no point was really established.

For developers using tools like **pylint**, the `status` argument isn't used. This creates a warning that needs to be silenced. Adding a `# pylint: disable=unused-argument` comment will silence the warning.

The `point_roll()` function looks like this:

```
from pymonad.tools import curry
from pymonad.maybe import Maybe, Just

@curry(2)  # type: ignore[misc]
def point_roll(dice: DiceT, status: Maybe) -> Maybe:
    prev, point, so_far = status
    if prev != "point":
        # won or lost on a previous throw
        return Just(status)

    d = dice()
    if sum(d) == 7:
        return Just(("fail", point, so_far+[d]))
    elif sum(d) == point:
        return Just(("pass", point, so_far+[d]))
    else:
        return (
            Just(("point", point, so_far+[d]))
            .then(point_roll(dice))
        )
```

We decomposed the status monad into the three individual values of the tuple. We could have used small lambda objects to extract the first, second, and third values. We could also have used the `operator.itemgetter()` function to extract the tuple's items. Instead, we

used multiple assignment.

If a point was not established, the previous state will be *pass* or *fail*. The game was resolved during the `initial_roll()` function, and this function simply returns the status monad.

If a point was established, the state will be *point*. The dice is rolled and rules applied to this new roll. If roll is 7, the game is a lost and a final monad is returned. If the roll is the point, the game is won and the appropriate monad is returned. Otherwise, a slightly revised monad is passed to the `point_roll()` function. The revised status monad includes this roll in the history of rolls.

A typical output looks like this:

```
>>> game_chain()
('fail', 5, [(2, 3), (1, 3), (1, 5), (1, 6)])
```

The final monad has a string that shows the outcome. It has the point that was established and the sequence of dice rolls leading to the final outcome.

We can use simulation to examine different outcomes to gather statistics on this complex, stateful process. This kind of Markov-chain model can reflect a number of odd edge cases that lead to surprising distributions of results.

A great deal of clever Monte Carlo simulation can be built with a few simple, functional programming design techniques. The monad, in particular, can help to structure these kinds of calculations when there are complex orders or internal states.

Additional PyMonad features

One of the other features of PyMonad is the confusingly named **monoid**. This comes directly from mathematics and it refers to a group of data elements that have an operator and an identity element, and the group is closed with respect to that operator. Here's an example of what this means: when we think of natural numbers, the add operator, and an identity element 0, this is a proper monoid. For positive integers, with an operator `*`, and

an identity value of 1, we also have a monoid; strings using + as an operator and an empty string as an identity element also qualify.

PyMonad includes a number of predefined monoid classes. We can extend this to add our own monoid class. The intent is to limit a compiler to certain kinds of optimization. We can also use the monoid class to create data structures which accumulate a complex value, perhaps including a history of previous operations.

The `pymonad.list` is an example of a monoid. The identity element is an empty list, defined by `ListMonad()`. The addition operation defines list concatenation. The monoid is an aspect of the overall `ListMonad()` class.

Much of this package helps provide deeper insights into functional programming. To paraphrase the documentation, this is an easy way to learn about functional programming in, perhaps, a slightly more forgiving environment. Rather than learning an entire language and toolset to compile and run functional programs, we can just experiment with interactive Python.

Pragmatically, we don't need too many of these features because Python is already stateful and offers strict evaluation of expressions. There's no practical reason to introduce stateful objects in Python, or strictly ordered evaluation. We can write useful programs in Python by mixing functional concepts with Python's imperative implementation. For that reason, we won't delve more deeply into PyMonad.

Summary

In this chapter, we looked at how we can use the PyMonad library to express some functional programming concepts directly in Python. The module contains many important functional programming techniques.

We looked at the idea of currying, a function that allows combinations of arguments to be applied to create new functions. Currying a function also allows us to use functional composition to create more complex functions from simpler pieces. We looked at functors that wrap simple data objects to make them into functions that can also be used with

functional composition.

Monads are a way to impose a strict evaluation order when working with an optimizing compiler and lazy evaluation rules. In Python, we don't have a good use case for monads because Python is an imperative programming language under the hood. In some cases, imperative Python may be more expressive and succinct than a monad construction.

In the next chapter, we'll look at the multiprocessing and multithreading techniques that are available to us. These packages become particularly helpful in a functional programming context. When we eliminate a complex shared state and design around non-strict processing, we can leverage parallelism to improve the performance.

Exercises

This chapter's exercises are based on code available from Packt Publishing on GitHub. See `https://github.com/PacktPublishing/Functional-Python-Programming-3rd-Editi` on.

In some cases, the reader will notice that the code provided on GitHub includes partial solutions to some of the exercises. These serve as hints, allowing the reader to explore alternative solutions.

In many cases, exercises will need unit test cases to confirm they actually solve the problem. They are often identical to the unit test cases provided in the GitHub repository. The reader should replace the book's example function name with their own solution to confirm that it works.

Revise the arctangent series

In *Using the lazy ListMonad() monad*, we showed a computation for π that involved summing fractions from a series that used factorials, $n!$, and double factorials, $(2n + 1)!!$.

The examples use a sequence `seq1 = ListMonad(*range(20))` with only 20 values. This choice of 20 was arbitrary, and intended only to keep the intermediate results small enough to visualize.

A better choice is to use use curried versions of the `sum()` and `takewhile()` functions to find the sum of values in the sequence until the values are too small to contribute to the result.

Rewrite the approximation to compute π to an accuracy of 10^{-15}. This is close to the limit of what 64-bit floating-point values can represent.

Statistical computations

Given a list of values, `v`, we can create a useful monad with `Just(v)`. We can use built-in functions like `sum()` and `len()` with the `Just.map()` method to compute the values required for mean, variance, and standard deviation.

$$\text{mean}(D) = \frac{\sum_{x \in D} x}{\text{count}(D)}$$

$$\text{var}(D) = \sum_{x \in D} \frac{(x - \text{mean}(D))^2}{\text{count}(D) - 1}$$

$$\text{stdev}(D) = \sqrt{\text{var}(D)}$$

After implementing these functions using PyMonad, compare these definitions with more conventional Python language techniques. Does the presence of a monad structure help with these relatively simple computations?

Data validation

The PyMonad library includes an `Either` class of monads. This is similar to the `Maybe` class of monads. The `Maybe` monad can have just a value, or nothing, providing a `None`-like object. An `Either` monad has two subclasses, `Left` and `Right`. If we use `Right` instances for valid data, we can use `Left` instances for error messages that identify invalid data.

The above concept suggests that a `try:/except:` statement can be used. If no Python exception is raised, the result is a `Right(v)`. If an exception is raised, a `Left` can be returned with the exception's error message.

This permits a `Compose` or `Pipe` to process data, emitting all of the erroneous rows as `Left`

monads. This can lead to a helpful data validation application because it spots all of the problems with the data.

First, define a simple validation rule, like "the values must be multiples of 3 or 5." This means they must convert to integer values and the integer modulo 3 is zero or the integer modulo 5 is zero. Second, write the validation function that returns either a `Right` or `Left` object.

While a `pymonad.io.IO` object can be used to parse a file, we'll start with applying the validation function to a list and examining the results. Apply the validation function to a sequence of values, saving the resulting sequence of `Either` objects.

An `Either` object has an `.either()` method which can process either `Left` or `Right` instances. For example, `e.either(lambda x: True, lambda x: False)` will return `True` if the value of the `e` monad is a `Left` instance.

Multiple models

A given process has several alternative models that compute an expected value from an observed sample value.

Each model computes an expected value, e, from the observed value in the sample, s_o:

- $e = 0.7412 \times s_o$
- $e = 0.9 \times s_o - 90$
- $e = 0.7724 \times s_o^{1.0134}$

First, we need to implement each of these models as a curried function. This will let us compute predicted values using any of these models.

Given a model function, we then need to create a comparison function. We can use a general PyMonad `Composition` or `Pipe` to compute a predicted value using one of the models and compare the predicted value with an observed value.

The results of this comparison function can be used as part of a χ^2 (chi-squared) test to discern how well the model fits the observations. The actual chi-squared metric is the

subject of *Chapter 16, A Chi-Squared Case Study*.

For now, create the curried model functions, and the `Composition` or `Pipe` to compare the model's prediction with actual results.

For actual values and observed values, see the *Logging* exercise in *Chapter 12, Decorator Design Techniques*.

14

The Multiprocessing, Threading, and Concurrent.Futures Modules

When we eliminate the complexities of shared state and design around pure functions with non-strict processing, we can leverage concurrency and parallelism to improve performance. In this chapter, we'll look at some of the multiprocessing and multithreading techniques that are available to us. Python library packages become particularly helpful when applied to algorithms designed from a functional viewpoint.

The central idea here is to distribute a functional program across several threads within a process or across several processes in a CPU. If we've created a sensible functional design, we can avoid complex interactions among application components; we have functions that accept argument values and produce results. This is an ideal structure for a process or a thread.

In this chapter, we'll focus on several topics:

- The general idea of functional programming and concurrency.
- What concurrency really means when we consider cores, CPUs, and OS-level concurrency and parallelism. It's important to note that concurrency won't magically make a bad algorithm faster.
- Using the built-in `multiprocessing` and `concurrent.futures` modules. These modules allow a number of parallel execution techniques. The external `dask` package can do much of this as well.

We'll focus on process-level parallelism more than multithreading. Using process parallelism allows us to completely ignore Python's **Global Interpreter Lock** (**GIL**).

For more information on Python's GIL, see `https://docs.python.org/3/glossary.html#term-global-interpreter-lock`. Also see `https://peps.python.org/pep-0684/` for a proposal to alter the way the GIL operates. Additionally, see `https://github.com/colesbury/nogil` for a project that proposes a way to remove the GIL entirely.

The GIL is very much part of Python 3.10, meaning some kinds of compute-intensive multithreading won't show significant speedups.

We'll focus on concurrency throughout the chapter. Concurrent work is interleaved, distinct from parallel work, which requires multiple cores or multiple processors. We don't want to dig too deeply into the nuanced distinctions between concurrency and parallelism. Our focus is on leveraging a functional approach, more than exploring all of the ways work can be accomplished in a modern CPU with a multi-processing OS.

Functional programming and concurrency

The most effective concurrent processing occurs when there are no dependencies among the tasks being performed. The biggest difficulty in developing concurrent (or parallel) programming is the complications arising from coordinating updates to shared resources,

where tasks depend on a common resource.

When following functional design patterns, we tend to avoid stateful programs. A functional design should minimize or eliminate concurrent updates to shared objects. If we can design software where lazy, non-strict evaluation is central, we can also design software where concurrent evaluation is helpful. In some cases, parts of an application can have an **embarrassingly parallel** design, where most of the work can be done concurrently with few or no interactions among computations. Mappings and filterings, in particular, benefit from parallel processing; reductions typically can't be done in parallel.

The frameworks we'll focus on all make use of an essential `map()` function to allocate work to multiple workers in a pool. This fits nicely with the higher-order functional design we've been looking at throughout this book. If we've built our application with a particular focus on the `map()` function, then partitioning the work into processes or threads should not involve a breaking change.

What concurrency really means

In a small computer, with a single processor and a single core, all evaluations are serialized through the one and only core of the processor. The OS will interleave multiple processes and multiple threads through clever time-slicing arrangements to make it appear as if things are happening concurrently.

On a computer with multiple CPUs or multiple cores in a single CPU, there can be some actual parallel processing of CPU instructions. All other concurrency is simulated through time slicing at the OS level. A macOS X laptop can have 200 concurrent processes that share the CPU; this is many more processes than the number of available cores. From this, we can see that OS time slicing is responsible for most of the apparently concurrent behavior of the system as a whole.

The boundary conditions

Let's consider a hypothetical algorithm that has a complexity described by $\mathbf{O}(n^2)$. This generally means two nested `for` statements, each of which is processing n items. Let's

assume the inner `for` statement's body involves 1,000 Python operation codes. When processing 10,000 objects, this could execute 100 billion Python operations. We can call this the essential processing budget. We can try to allocate as many processes and threads as we feel might be helpful, but the processing budget can't change.

The individual CPython bytecodes—the internal implementation of Python statements and expressions—don't all share a single, uniform execution time. However, a long-term average on a macOS X laptop shows that we can expect about 60 MB of bytecode operations to be executed per second. This means that our 100 billion bytecode operation could take about 1,666 seconds, or 28 minutes.

If we have a dual-processor, four-core computer, then we might cut the elapsed time to 25% of the original total: about 7 minutes. This presumes that we can partition the work into four (or more) independent OS processes.

The important consideration here is that the overall budget of 100 billion bytecodes can't be changed. Concurrency won't magically reduce the workload. It can only change the schedule to perhaps reduce the elapsed time to execute all those bytecodes.

Switching to a better algorithm with a complexity of $\mathbf{O}(n \log n)$ can reduce the workload dramatically. We need to measure the actual speedup to determine the impact; the following example includes a number of assumptions. Instead of doing $10,000^2$ iterations, we may only do $10,000 \log 10,000 \approx 132,877$ iterations, dropping from 100 billion operations to a number on the order of 133 thousand operations. This could be as small as $\frac{1}{700}$ of the original time. Concurrency can't provide the kind of dramatic improvements that algorithm change will have.

Sharing resources with process or threads

The OS assures us there is little or no interaction between processes. When creating an application where multiple processes must interact, a common OS resource must be explicitly shared. This can be a common file, a shared-memory object, or a semaphore with a shared state between the processes. Processes are inherently independent; interaction

among them is the exception, not the rule.

Multiple threads, in contrast, are part of a single process; all threads of a process generally share resources, with one special case. Thread-local memory can be freely used without interference from other threads. Outside thread-local memory, operations that write to memory can set the internal state of the process in a potentially unpredictable order. One thread can overwrite the results of another thread. A technique for mutually exclusive access—often a form of locking—*must* be used to avoid problems. As noted previously, the overall sequence of instruction from concurrent threads and processes are generally interleaved among the cores in an unpredictable order. With this concurrency comes the possibility of destructive updates to shared variables and the need for mutually exclusive access.

The existence of concurrent object updates can create havoc when trying to design multithreaded applications. Locking is one way to avoid concurrent writes to shared objects. Avoiding shared objects in general is another viable design technique. The second technique—avoiding writes to shared objects—is often also applicable to functional programming.

In CPython, the GIL is used to ensure that OS thread scheduling will not interfere with the internals of maintaining Python data structures. In effect, the GIL changes the granularity of scheduling from machine instructions to groups of Python virtual machine operations.

Pragmatically, the performance impact of the GIL on a wide variety of application types is often negligible. For the most part, compute-intensive applications tend to see the largest impact from GIL scheduling. I/O-intensive applications see little impact because the threads spend more time waiting for I/O to complete. A far greater impact on performance comes from the fundamental inherent complexity of the algorithm being implemented.

Where benefits will accrue

A program that does a great deal of calculation and relatively little I/O will not see much benefit from concurrent processing on a single core. If a calculation has a budget of

28 minutes of computation, then interleaving the operations in different ways won't have a dramatic impact. Using eight cores for parallel computation may cut the time by approximately one-eighth. The actual time savings depend on the OS and language overheads, which are difficult to predict.

When a calculation involves a great deal of I/O, then interleaving CPU processing while waiting for I/O requests to complete can *dramatically* improve performance. The idea is to do computations on some pieces of data while waiting for the OS to complete the I/O of other pieces of data. Because I/O generally involves a great deal of waiting, an eight-core processor can interleave the work from dozens (or hundreds) of concurrent I/O requests.

Concurrency is a core principle behind Linux. If we couldn't do computation while waiting for I/O, then our computer would freeze while waiting for each network request to finish. A website download would involve waiting for the initial HTML and then waiting for each individual graphic to arrive. All the while, the keyboard, mouse, and display would not work.

Here are two approaches to designing applications that interleave computation and I/O:

- We can create a pipeline of processing stages. An individual item must move through all of the stages where it is read, filtered, computed, aggregated, and written. The idea of multiple concurrent stages means there will be distinct data objects in each stage. Time slicing among the stages will allow computation and I/O to be interleaved.
- We can create a pool of concurrent workers, each of which performs all of the processing for a data item. The data items are assigned to workers in the pool and the results are collected from the workers.

The differences between these approaches aren't crisp. It's common to create a hybrid mixture where one stage of a pipeline involves a pool of workers to make that stage as fast as the other stages. There are some formalisms that make it somewhat easier to design concurrent programs. The **Communicating Sequential Processes** (**CSP**) paradigm can help design message-passing applications. Packages such as `pycsp` can be used to add CSP formalisms to Python.

I/O-intensive programs often gain the most dramatic benefits from concurrent processing. The idea is to interleave I/O and processing. CPU-intensive programs will see smaller benefits from concurrent processing.

Using multiprocessing pools and tasks

Python's `multiprocessing` package introduces the concept of a `Pool` object. A `Pool` object contains a number of worker processes and expects these processes to be executed concurrently. This package allows OS scheduling and time slicing to interleave execution of multiple processes. The intention is to keep the overall system as busy as possible.

To make the most of this capability, we need to decompose our application into components for which non-strict, concurrent execution is beneficial. The overall application must be built from discrete tasks that can be processed in an indefinite order.

An application that gathers data from the internet through web scraping, for example, is often optimized through concurrent processing. A number of individual processes can be waiting for the data to download, while others are performing the scraping operation on data that's been received. We can create a `Pool` object of several identical workers, which implement the website scraping. Each worker is assigned tasks in the form of URLs to be analyzed. Multiple workers waiting for downloads have little processing overhead. The workers with completely downloaded pages, on the other hand, can perform the real work of extracting data from the content.

An application that analyzes multiple log files is also a good candidate for concurrent processing. We can create a `Pool` object of analytical workers. We can assign each log file to a worker; this allows reading and analysis to proceed concurrently among the various workers in the `Pool` object. Each individual worker will be performing both I/O and computation. However, some workers can be analyzing while other workers are waiting for I/O to complete.

Because the benefits depend on difficult-to-predict timing for input and output operations,

multiprocessing always involves experimentation. Changing the pool size and measuring elapsed time is an essential part of implementing concurrent applications.

Processing many large files

Here is an example of a multiprocessing application. We'll parse **Common Log Format** (**CLF**) lines in web log files. This is the generally used format for web server access logs. The lines tend to be long, but look like the following when wrapped to the book's margins:

```
99.49.32.197 - - [01/Jun/2012:22:17:54 -0400] "GET /favicon.ico\\
HTTP/1.1" 200 894 "-" "Mozilla/5.0 (Windows NT 6.0)\\
AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.52\\
Safari/536.5"
```

We often have large numbers of files that we'd like to analyze. The presence of many independent files means that concurrency will have some benefit for our scraping process. Some workers will be waiting for data, while others can be doing the compute-intensive portion of the work.

We'll decompose the analysis into two broad areas of functionality. The first phase of processing is the essential parsing of the log files to gather the relevant pieces of information. We'll further decompose the parsing phase into four stages. They are as follows:

1. All the lines from multiple source log files are read.
2. Then, we create simple `NamedTuple` objects from the lines of log entries in a collection of files.
3. The details of more complex fields such as dates and URLs are parsed separately.
4. Uninteresting paths from the logs are rejected, leaving the interesting paths for further processing.

Once past the parsing phase, we can perform a large number of analyses. For our purposes in demonstrating the `multiprocessing` module, we'll look at a simple analysis to count occurrences of specific paths.

The first portion is reading from the source files. Python's use of file iterators will translate

into lower-level OS requests for the buffering of data. Each OS request means that the process must wait for the data to become available.

Clearly, we want to interleave the other operations so that they are not all waiting for I/O to complete. The operations can be imagined to form a spectrum, from processing individual rows to processing whole files. We'll look at interleaving whole files first, as this is relatively simple to implement.

The functional design for parsing Apache CLF files can look as follows:

```
data = path_filter(
    access_detail_iter(
        access_iter(
            local_gzip(filename))))
```

This function decomposes the larger parsing problem into a number of functions. The `local_gzip()` function reads rows from locally cached GZIP files. The `access_iter()` function creates a `NamedTuple` object for each row in the access log. The `access_detail_iter()` function expands on some of the more difficult-to-parse fields. Finally, the `path_filter()` function discards some paths and file extensions that aren't of much analytical value.

It can help to visualize this kind of design as a shell-like pipeline of processing, as shown here:

```
(local_gzip(filename) | access_iter
    | access_detail_iter | path_filter) >data
```

This uses borrows the shell notation of a pipe (|) to pass data from process to process. Python doesn't have this operator, directly.

Pragmatically, we can use the `toolz` module to define this pipeline:

```
from toolz.functoolz import pipe

data = pipe(filename,
    local_gzip,
    access_iter,
    access_detail_iter,
    path_filter
)
```

For more on the `toolz` module, see *Chapter 11, The Toolz Package*.

We'll focus on designing these four functions that process data in stages. The idea is to interleave intensive processing with waiting for I/O to finish.

Parsing log files – gathering the rows

Here is the first stage in parsing a large number of files: reading each file and producing a simple sequence of lines. As the log files are saved in the `.gzip` format, we need to open each file with the `gzip.open()` function.

The following `local_gzip()` function reads lines from locally cached files:

```
from collections.abc import Iterator
import gzip
from pathlib import Path

import sys
def local_gzip(zip_path: Path) -> Iterator[str]:
    with gzip.open(zip_path, "rb") as log_file:
        yield from (
            line.decode('us-ascii').rstrip()
            for line in log_file
        )
```

The function iterates through all lines of a file. We've created a composite function that encapsulates the details of opening a log file compressed with the `.gzip` format, breaking

a file into a sequence of lines, and stripping the newline (`\n`) characters.

Additionally, this function also encapsulates a non-standard encoding for the files. Instead of Unicode, encoded in a standard format like UTF-8 or UTF-16, the files are encoded in old US-ASCII. This is very similar to UTF-8. In order to be sure the log entries are read properly, the exact encoding is supplied.

This function is a close fit with the way the `multiprocessing` module works. We can create a worker pool and map tasks (such as `.gzip` file reading) to the pool of processes. If we do this, we can read these files in parallel; the open file objects will be part of separate processes, and the resource consumption and wait time will be managed by the OS.

An extension to this design can include a second function to transfer files from the web host using SFTP or a RESTful API if one is available. As the files are collected from the web server, they can be analyzed using the `local_gzip()` function.

The results of the `local_gzip()` function are used by the `access_iter()` function to create named tuples for each row in the source file that describes file access by the web server.

Parsing log lines into named tuples

Once we have access to all of the lines of each log file, we can extract details of the access that's described. We'll use a regular expression to decompose the line. From there, we can build a `NamedTuple` object.

Each individual access can be summarized as a subclass of `NamedTuple`, as follows:

```
from typing import NamedTuple, Optional, cast
import re

class Access(NamedTuple):
    host: str
    identity: str
    user: str
    time: str
    request: str
```

```
    status: str
    bytes: str
    referer: str
    user_agent: str

    @classmethod
    def create(cls: type, line: str) -> Optional["Access"]:
        format_pat = re.compile(
            r"(?P<host>[\d\.]+)\s+"
            r"(?P<identity>\S+)\s+"
            r"(?P<user>\S+)\s+"
            r"\[(?P<time>.+?)\]\s+"
            r'"(?P<request>.+?)"\s+'
            r"(?P<status>\d+)\s+"
            r"(?P<bytes>\S+)\s+"
            r'"(?P<referer>.*?)"\s+'
            r'"(?P<user_agent>.+?)"\s*'
            )
        if match := format_pat.match(line):
            return cast(Access, cls(**match.groupdict()))
        return None
```

The method for building an `Access` object, `create()`, from the source text contains a lengthy regular expression to parse lines in a CLF file. This is quite complex, but we can use a **railroad diagram** to help simplify it. The following image shows the various elements and how they're identified by the regular expression:

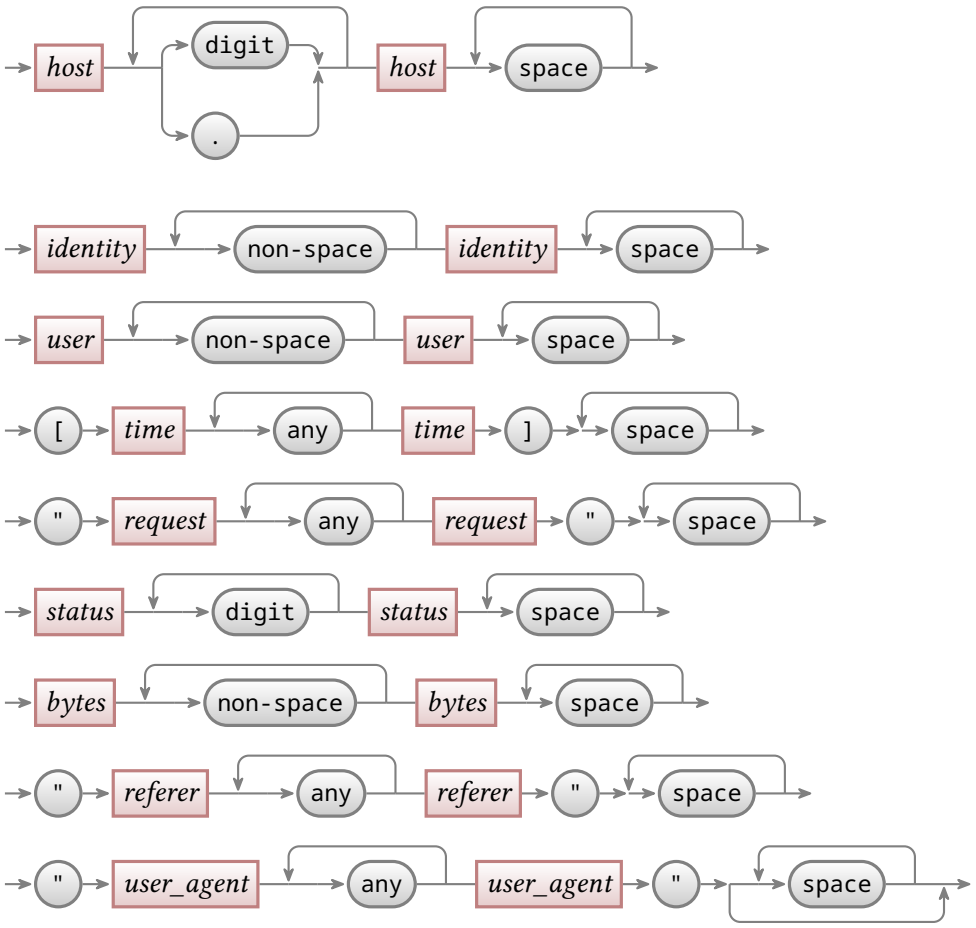


Figure 14.1: Regular expression diagram for parsing log files

This diagram shows the sequence of clauses in the regular expression. Each rectangular box represents a named capture group. For example, `(?P<host>[\d\.]+)` is a group named `host`. The ovals and circles are classes of characters (e.g., digit) or specific characters (e.g., `.`) that comprise the contents of the capture group.

We used this regular expression to break each row into a dictionary of nine individual data elements. The use of `[]` and `"` to delimit complex fields such as the `time`, `request`, `referer`, and `user_agent` parameters can be handled elegantly by transforming the text into a `NamedTuple` object.

We've taken pains to ensure that the `NamedTuple` field names match the regular expression group names in the `(?P<name>)` constructs for each portion of the record. By making sure the names match, we can very easily transform the parsed dictionary into a tuple for further processing. This means we've spelled *referrer* wrong to fit with the RFC documentation.

Here is the `access_iter()` function, which requires each file to be represented as an iterator over the lines of the file:

```
from collections.abc import Iterator

def access_iter(source_iter: Iterator[str]) -> Iterator[Access]:
    for line in source_iter:
        if access := Access.create(line):
            yield access
```

The output from the `local_gzip()` function is a sequence of strings. The outer sequence is based on the lines from individual log files. If the line matches the given pattern, it's a file access of some kind. We can create an `Access` instance from the dictionary of text parsed by the regular expression. Non-matching lines are quietly discarded.

The essential design pattern here is to build an immutable object from the results of a parsing function. In this case, the parsing function is a regular expression matcher. Other kinds of parsing can also fit this design pattern.

There are some alternative ways to do this. For example, here's a function that applies `map()` and `filter()`:

```
def access_iter_2(source_iter: Iterator[str]) -> Iterator[Access]:
    return filter(
        None,
        map(
            Access.create,
```

```
            source_iter
        )
    )
```

This `access_iter_2()` function transforms the output from the `local_gzip()` function into a sequence of `Access` instances. In this case, we apply the `Access.create()` function to the string iterator that results from reading a collection of files. The `filter()` function removes any `None` objects from the result of the `map()` function.

Our point here is to show that we have a number of functional styles for parsing files. In *Chapter 4*, *Working with Collections*, we looked at very simple parsing. Here, we're performing more complex parsing, using a variety of techniques.

Parsing additional fields of an Access object

The initial `Access` object created previously doesn't decompose some inner elements in the nine fields that comprise an access log line. We'll parse those items separately from the overall decomposition into high-level fields. Doing these parsing operations separately makes each stage of processing simpler. It also allows us to replace one small part of the overall process without breaking the general approach to analyzing logs.

The resulting object from the next stage of parsing will be a `NamedTuple` subclass, `AccessDetails`, which wraps the original `Access` tuple. It will have some additional fields for the details parsed separately:

```
from typing import NamedTuple, Optional
import datetime
import urllib.parse

class AccessDetails(NamedTuple):
    access: Access
    time: datetime.datetime
    method: str
    url: urllib.parse.ParseResult
```

```
    protocol: str
    referrer: urllib.parse.ParseResult
    agent: dict[str, str]

    @classmethod
    def create(cls: type, access: Access) -> "AccessDetails":
          meth, url, protocol = parse_request(access.request)
          return AccessDetails(
              access=access,
              time=parse_time(access.time),
              method=meth,
              url=urllib.parse.urlparse(url),
              protocol=protocol,
              referrer=urllib.parse.urlparse(access.referer),
              agent=parse_agent(access.user_agent)
          )
```

The `access` attribute is the original `Access` object, a collection of simple strings. The `time` attribute is the parsed `access.time` string. The `method`, `url`, and `protocol` attributes come from decomposing the `access.request` field. The `referrer` attribute is a parsed URL.

The `agent` attribute can also be broken down into fine-grained fields. The rules are quite complex, and we've decided that a dictionary mapping names to their associated values will be good enough.

Here are the three detail-level parsers for the fields to be decomposed:

```
from typing import Optional
import datetime
import re

def parse_request(request: str) -> tuple[str, str, str]:
    words = request.split()
    return words[0], ' '.join(words[1:-1]), words[-1]

def parse_time(ts: str) -> datetime.datetime:
```

```
    return datetime.datetime.strptime(
        ts, "%d/%b/%Y:%H:%M:%S %z"
    )

def parse_agent(user_agent: str) -> dict[str, str]:
    agent_pat = re.compile(
        r"(?P<product>\S*?)\s+"
        r"\((?P<system>.*?)\)\s*"
        r"(?P<platform_details_extensions>.*)"
    )

    if agent_match := agent_pat.match(user_agent):
        return agent_match.groupdict()
    return {}
```

We've written three parsers for the HTTP request, the time stamp, and the user agent information. The request value in a log is usually a three-word string such as `GET /some /path HTTP/1.1`. The `parse_request()` function extracts these three space-separated values. In the unlikely event that the path has spaces in it, we'll extract the first word and the last word as the method and protocol; all the remaining words are part of the path.

Time parsing is delegated to the `datetime` module. We've provided the proper format in the `parse_time()` function.

Parsing the user agent is challenging. There are many variations; we've chosen a common one for the `parse_agent()` function. If the user agent text matches the given regular expression, we'll use the attributes of the `AgentDetails` class. If the user agent information doesn't match the regular expression, we'll use the `None` value instead. The original text will be available in the `Access` object in either case.

We'll use these three parsers to build `AccessDetails` instances from the given `Access` objects. The main body of the `access_detail_iter()` function looks like this:

```
from collections.abc import Iterable, Iterator

def access_detail_iter(
    access_iter: Iterable[Access]
) -> Iterator[AccessDetails]:
    for access in access_iter:
        yield AccessDetails.create(access)
```

We've used a similar design pattern to the previous `access_iter()` function. A new object is built from the results of parsing some input object. The new `AccessDetails` object will wrap the previous `Access` object. This technique allows us to use immutable objects, yet still contains more refined information.

This function is essentially a mapping from an `Access` object to a sequence of `AccessDetails` objects. Here's an alternative design using the `map()` high-level function:

```
from collections.abc import Iterable, Iterator

def access_detail_iter_2(
    access_iter: Iterable[Access]
) -> Iterator[AccessDetails]:
    return map(AccessDetails.create, access_iter)
```

As we move forward, we'll see that this variation fits in nicely with the way the `multiprocessing` module works.

In an object-oriented programming environment, these additional parsers might be method functions or properties of a class definition. The advantage of an object-oriented design with lazy parsing methods is that items aren't parsed unless they're needed. This particular functional design parses everything, assuming that it's going to be used.

It's possible to create a lazy functional design. It can rely on the three parser functions to extract and parse the various elements from a given `Access` object as needed. Rather than using the `details.time` attribute, we'd use the `parse_time(access.time)` function. The

syntax is longer, but it ensures that the attribute is only parsed as needed. We could also make it a property that preserves the original syntax. We've left this as an exercise for the reader.

Filtering the access details

We'll look at several filters for the `AccessDetails` objects. The first is a collection of filters that reject a lot of overhead files that are rarely interesting. The second filter will be part of the analysis functions, which we'll look at later.

The `path_filter()` function is a combination of three functions:

- Exclude empty paths
- Exclude some specific filenames
- Exclude files that have a given extension

A flexible design can define each test as a separate first-class, filter-style function. For example, we might have a function such as the following to handle empty paths:

```
def non_empty_path(detail: AccessDetails) -> bool:
    path = detail.url.path.split('/')
    return any(path)
```

This function ensures that the path contains a name. We can write similar tests for the `non_excluded_names()` and `non_excluded_ext()` functions. Names like `favicon.ico` and `robots.txt` need to be excluded. Similarly, extensions like `.js` and `.css` need to be excluded as well. We've left these two additional filters as exercises for the reader.

The entire sequence of `filter()` functions will look like this:

```
def path_filter(
    access_details_iter: Iterable[AccessDetails]
) -> Iterable[AccessDetails]:
    non_empty = filter(non_empty_path, access_details_iter)
    nx_name = filter(non_excluded_names, non_empty)
```

```
    nx_ext = filter(non_excluded_ext, nx_name)
    yield from nx_ext
```

This style of stacked filters has the advantage of being slightly easier to expand when we add new filter criteria.

The use of generator functions (such as the `filter()` function) means that we aren't creating large intermediate objects. Each of the intermediate variables, `non_empty`, `nx_name`, and `nx_ext`, is a proper lazy generator function; no processing is done until the data is consumed by a client process.

While elegant, this suffers from inefficiency because each function will need to parse the path in the `AccessDetails` object. In order to make this more efficient, we could wrap a `path.split('/')` function with the `@cache` decorator. An alternative is to split the path on the `/` characters, and save the list in the `AccessDetails` object.

Analyzing the access details

We'll look at two analysis functions that we can use to filter and analyze the individual `AccessDetails` objects. The first function will filter the data and pass only specific paths. The second function will summarize the occurrences of each distinct path.

We'll define a small `book_in_path()` function and combine this with the built-in `filter()` function to apply the function to the details. Here is the composite `book_filter()` function:

```
from collections.abc import Iterable, Iterator

def book_filter(
    access_details_iter: Iterable[AccessDetails]
) -> Iterator[AccessDetails]:
    def book_in_path(detail: AccessDetails) -> bool:
        path = tuple(
            item
            for item in detail.url.path.split('/')
```

```
            if item
        )
        return path[0] == 'book' and len(path) > 1
    return filter(book_in_path, access_details_iter)
```

We've defined a rule, through the `book_in_path()` function, which we'll apply to each `AccessDetails` object. If the path has at least two components and the first component of the path is `'book'`, then we're interested in these objects. All other `AccessDetails` objects can be quietly rejected.

The `reduce_book_total()` function is the final reduction that we're interested in:

```
from collections import Counter

def reduce_book_total(
    access_details_iter: Iterable[AccessDetails]
) -> dict[str, int]:
    counts: Counter[str] = Counter(
        detail.url.path for detail in access_details_iter
    )
    return counts
```

This function will produce a `Counter()` object that shows the frequency of each path in an `AccessDetails` object. In order to focus on a particular set of paths, we'll use the `reduce_total(book_filter(details))` expression. This provides a summary of only items that are passed by the given filter.

Because `Counter` objects can be applied to a wide variety of types, a type hint is required to provide a narrow specification. In this case, the hint is `dict[str, int]` to show the **mypy** tool that string representations of paths will be counted.

The complete analysis process

Here is the composite `analysis()` function that digests a collection of log files:

```
def analysis(log_path: Path) -> dict[str, int]:
    """Count book chapters in a given log"""
    details = access_detail_iter(
        access_iter(
            local_gzip(log_path)))
    books = book_filter(path_filter(details))
    totals = reduce_book_total(books)
    return totals
```

The `analysis()` function uses the `local_gzip()` function to work with a single path. It applies a stack of parsing functions, `access_detail_iter()` and `access_iter()`, to create an iterable sequence of `AccessDetails` objects. It then applies a stack of filters to exclude paths that aren't interesting. Finally, it applies a reduction to a sequence of `AccessDetails` objects. The result is a `Counter` object that shows the frequency of access for certain paths.

A sample collection of saved `.gzip` format log files totals about 51 MB. Processing the files serially with this function takes over 140 seconds. Can we do better using concurrent processing?

Using a multiprocessing pool for concurrent processing

One elegant way to make use of the `multiprocessing` module is to create a processing `Pool` object and assign work to the various workers in that pool. We will depend on the OS to interleave execution among the various processes. If each of the processes has a mixture of I/O and computation, we should be able to ensure that our processor (and disk) are kept very busy. When processes are waiting for the I/O to complete, other processes can do their computations. When an I/O operation finishes, the process waiting for this will be ready to run and can compete with others for processing time.

The recipe for mapping work to a separate process looks like this:

```
def demo_mp(root: Path = SAMPLE_DATA, pool_size: int | None = None)
-> None:
    pool_size = (
        multiprocessing.cpu_count() if pool_size is None
        else pool_size
    )
    combined: Counter[str] = Counter()
    with multiprocessing.Pool(pool_size) as workers:
        file_iter = list(root.glob(LOG_PATTERN))
        results_iter = workers.imap_unordered(analysis, file_iter)
        for result in results_iter:
            combined.update(result)
    print(combined)
```

This function creates a `Pool` object with separate worker processes and assigns this `Pool` object to the `workers` variable. We then map the analytical function, `analysis`, to an iterable queue of work to be done using the pool of processes. Each process in the `workers` pool gets assigned items from the iterable queue. In this case, the queue is the result of the `root.glob(LOG_PATTERN)` attribute, which is a sequence of file names.

As each worker completes the `analysis()` function and returns a result, the parent process that created the `Pool` object can collect those results. This allows us to create several concurrently built `Counter` objects and to merge them into a single, composite result.

If we start p processes in the pool, our overall application will include $p + 1$ processes. There will be one parent process and p children. This often works out well because the parent process will have little to do after the subprocess pools are started. Generally, the workers will be assigned to separate CPUs (or cores) and the parent will share a CPU with one of the children in the `Pool` object.

The ordinary Linux parent/child process rules apply to the subprocesses created by this module. If the parent crashes without properly collecting the final status from the child processes, then zombie processes can be left

running. For this reason, a process `Pool` object is also a context manager. When we use a pool through the `with` statement, at the end of the context, the children are properly collected.

By default, a `Pool` object will have a number of workers based on the value of the `multiprocessing.cpu_count()` function. This number is often optimal, and simply using the `with multiprocessing.Pool() as workers:` attribute might be sufficient.

In some cases, it can help to have more workers than CPUs. This might be true when each worker has I/O-intensive processing. Having many worker processes waiting for I/O to complete can improve the overall runtime of an application.

If a given `Pool` object has p workers, this mapping can cut the processing time to almost $\frac{1}{p}$ of the time required to process all of the logs serially. Pragmatically, there is some overhead involved with communication between the parent and child processes in the `Pool` object. These overheads will limit the effectiveness of subdividing the work into very small concurrent pieces.

The multiprocessing `Pool` object has several map-like methods to allocate work to a pool. We'll look at `map()`, `imap()`, `imap_unordered()`, and `starmap()`. Each of these is a variation on the common theme of assigning a function to a pool of processes and mapping data items to that function. Additionally, there are two async variants: `map_async()` and `starmap_async()`. These functions differ in the details of allocating work and collecting results:

- The `map(function, iterable)` method allocates items from the iterable to each worker in the pool. The finished results are collected in the order they were allocated to the `Pool` object so that order is preserved.
- The `imap(function, iterable)` method is lazier than `map()`. By default, it sends each individual item from the iterable to the next available worker. This might involve more communication overhead. For this reason, a chunk size larger than 1 is suggested.

- The `imap_unordered(function, iterable)` method is similar to the `imap()` method, but the order of the results is not preserved. Allowing the mapping to be processed out of order means that, as each process finishes, the results are collected.
- The `starmap(function, iterable)` method is similar to the `itertools.starmap()` function. Each item in the iterable must be a tuple; the tuple is passed to the function using the * modifier so that each value of the tuple becomes a positional argument value. In effect, it's performing `function(*iterable[0])`, `function(*iterable[1])`, and so on.

The two `_async` variants don't simply return a result; they return an `AsyncResult` object. This object has some status information. We can, for example, see if the work has been completed in general, or if it has been completed without an exception. The most important method of an `AsyncResult` object is the `.get()` method, which interrogates the worker for the result.

This extra complexity works well when the duration of processing is highly variable. We can collect results from workers as the results become available. The behavior for the non-`_async` variants is to collect results in the order the work was started, preserving the order of the original source data for the map-like operation.

Here is the `map_async()` variant of the preceding mapping theme:

```
def demo_mp_async(root: Path = SAMPLE_DATA, pool_size: int | None =
None) -> None:
    pool_size = (
        multiprocessing.cpu_count() if pool_size is None
        else pool_size
    )
    combined: Counter[str] = Counter()
    with multiprocessing.Pool(pool_size) as workers:
        file_iter = root.glob(LOG_PATTERN)
        results = workers.map_async(analysis, file_iter)
        for result in results.get():
            combined.update(result)
```

```
    print(combined)
```

We've created a `Counter()` function that we'll use to consolidate the results from each worker in the pool. We created a pool of subprocesses based on the number of available CPUs, and used the `Pool` object as a context manager. We then mapped our `analysis()` function to each file in our file-matching pattern. The resulting `Counter` objects from the `analysis()` function are combined into a single resulting counter.

This version took about 68 seconds to analyze a batch of log files. The time to analyze the logs was cut dramatically using several concurrent processes. The single-process baseline time was 150 seconds. Other experiments need to be run with larger pool sizes to determine how many workers are required to make the system as busy as possible.

We've created a two-tiered map-reduce process with the `multiprocessing` module's `Pool.map_async()` function. The first tier was the `analysis()` function, which performed a map-reduce on a single log file. We then consolidated these reductions in a higher-level reduce operation.

Using apply() to make a single request

In addition to the map-like variants, a pool also has an `apply(function, *args, **kw)` method that we can use to pass one value to the worker pool. We can see that the various `map()` methods are really a `for` statement wrapped around the `apply()` method. We can, for example, use the following command to process a number of files:

```
list(
    workers.apply(analysis, f)
    for f in SAMPLE_DATA.glob(LOG_PATTERN)
)
```

It's not clear, for our purposes, that this is a significant improvement. Almost everything we need to do can be expressed as a `map()` function.

More complex multiprocessing architectures

The `multiprocessing` package supports a wide variety of architectures. We can create multiprocessing structures that span multiple servers and provide formal authentication techniques to create a necessary level of security. We can pass objects from process to process using queues and pipes. We can share memory between processes. We can also share lower-level locks between processes as a way to synchronize access to shared resources such as files.

Most of these architectures involve explicitly managing states among several working processes. Using locks and shared memory, in particular, is imperative in nature and doesn't fit in well with a functional programming approach.

We can, with some care, treat queues and pipes in a functional manner. Our objective is to decompose a design into producer and consumer functions. A **producer** can create objects and insert them into a queue. A **consumer** will take objects out of a queue and process them, perhaps putting intermediate results into another queue. This creates a network of concurrent processors and the workload is distributed among these various processes.

This design technique has some advantages when designing a complex application server. The various subprocesses can exist for the entire life of the server, handling individual requests concurrently.

Using the concurrent.futures module

In addition to the `multiprocessing` package, we can also make use of the `concurrent.futures` module. This also provides a way to map data to a concurrent pool of threads or processes. The module API is relatively simple and similar in many ways to the `multiprocessing.Pool()` function's interface.

Here is an example to show how similar they are:

```
def demo_cf_threads(root: Path = SAMPLE_DATA, pool_size: int = 4) ->
None:
```

```
    pattern = "*itmaybeahack.com*.gz"
    combined: Counter[str] = Counter()
    with futures.ProcessPoolExecutor(max_workers=pool_size)
            as workers:
        file_iter = root.glob(LOG_PATTERN)
        for result in workers.map(analysis, file_iter):
            combined.update(result)
    print(combined)
```

The most significant change between the preceding example and the previous examples is that we're using an instance of the `concurrent.futures.ProcessPoolExecutor` object instead of a `multiprocessing.Pool` object. The essential design pattern is to map the `analysis()` function to the list of filenames using the pool of available workers. The resulting `Counter` objects are consolidated to create a final result.

The performance of the `concurrent.futures` module is nearly identical to the `multiprocessing` module.

Using concurrent.futures thread pools

The `concurrent.futures` module offers a second kind of executor that we can use in our applications. Instead of creating a `concurrent.futures.ProcessPoolExecutor` object, we can use the `ThreadPoolExecutor` object. This will create a pool of threads within a single process.

The syntax for thread pools is almost identical to using a `ProcessPoolExecutor` object. The performance, however, can be remarkably different. CPU-intensive processing doesn't often show improvement in a multi-threaded environment because there's no computation while waiting for I/O to complete. Processing that is I/O-intensive can benefit from multi-threading.

Using sample log files and a small four-core laptop running macOS X, these are the kinds of results that indicate the difference between threads that share I/O resources and processes:

- Using the `concurrent.futures` thread pool, the elapsed time was 168 seconds

- Using a process pool, the elapsed time was 68 seconds

In both cases, the `Pool` object's size was `4`. The single-process and single-thread baseline time was 150 seconds; adding threads made processing run more slowly. This result is typical of programs doing a great deal of computation with relatively little waiting for input and output. The `multithreading` module is often more appropriate for the following kinds of applications:

- User interfaces where threads are idle for long periods of time, while waiting for the person to move the mouse or touch the screen
- Web servers where threads are idle while waiting for data to transfer from a large, fast server through a network to a (relatively) slow client
- Web clients that extract data from multiple web servers, especially where these clients must wait for data to percolate through a network

It's important to benchmark and measure performance.

Using the threading and queue modules

The Python `threading` package involves a number of constructs helpful for building imperative applications. This module is not focused on writing functional applications. We can make use of thread-safe queues in the `queue` module to pass objects from thread to thread.

A queue permits safe data sharing. Since the queue processing involves using OS services, it can also mean applications using queues may observe less interference from the GIL.

The `threading` module doesn't have a simple way of distributing work to various threads. The API isn't ideally suited to functional programming.

As with the more primitive features of the `multiprocessing` module, we can try to conceal the stateful and imperative nature of locks and queues. It seems easier, however, to make use of the `ThreadPoolExecutor` method in the `concurrent.futures` module. The `ThreadPoolExecutor.map()` method provides us with a very pleasant interface to concurrently process the elements of a collection.

The use of the `map()` function primitive to allocate work seems to fit nicely with our functional programming expectations. For this reason, it's best to focus on the `concurrent.futures` module as the most accessible way to write concurrent functional programs.

Using async functions

The `asyncio` module helps us work with `async` functions to—perhaps—better interleave processing and computation. It's important to understand that `async` processing leverages the `threading` model. This means that it can effectively interleave waiting for I/O with computation. It does not effectively interleave pure computation.

In order to make use of the `asyncio` module, we need to do the following four things:

1. Add the `async` keyword to our various parsing and filtering functions to make them **coroutines**.
2. Add `await` keywords to collect results from one coroutine before passing them to another coroutine.
3. Create an overall event loop to coordinate the `async`/`await` processing among the coroutines.
4. Create a thread pool to handle file reading.

The first three steps listed above don't involve deep complexity. The `asyncio` module helps us create tasks to parse each file, and then run the collection of tasks. The event loop ensures that coroutines will pause at the `await` statements to collect results. It also ensures coroutines with available data are eligible to process. The interleaving of the coroutines happens in a single thread. As noted previously, the number of bytecode operations is not magically made smaller by changing the order of execution.

The tricky part of this is dealing with input and output operations that are *not* part of the `asyncio` module. Specifically, reading and writing local files is not part of `asyncio`. Any time we attempt to read (or write) a file, the operating system request could **block** waiting for the operation to complete. Unless this blocking request is in a separate thread, it stops the event loop, and stops all of Python's cleverly interleaved coroutine process-

ing. See `https://docs.python.org/3/library/asyncio-eventloop.html#id14` for more information on using a thread pool.

To work with local files, we would need to use a `concurrent.futures.ThreadPoolExecutor` object to manage the file input and output operations. This will allocate the work to threads outside the main event loop. Consequently, a design for local file processing based on `async`/`await` will not be dramatically better than one using `concurrent.futures` directly.

For network servers and complex clients, the `asyncio` module can make the application very responsive to a user's inputs. The fine-grained switching among the coroutines within a thread works best when most of the coroutines are waiting for data.

Designing concurrent processing

From a functional programming perspective, we've seen three ways to use the `map()` function concept applied to data items concurrently. We can use any one of the following:

- `multiprocessing.Pool`
- `concurrent.futures.ProcessPoolExecutor`
- `concurrent.futures.ThreadPoolExecutor`

These are almost identical in the way we interact with them; all three of these process pools support variations of a `map()` method that applies a function to items of an iterable collection. This fits in elegantly with other functional programming techniques. The performance of each pool may be different because of the nature of concurrent threads versus concurrent processes.

As we stepped through the design, our log analysis application decomposed into two overall areas:

- The lower-level parsing: This is generic parsing that will be used by almost any log analysis application
- The higher-level analysis application: This is more specific filtering and reduction focused on our application's needs

The lower-level parsing can be decomposed into four stages:

1. Reading all the lines from multiple source log files. This was the `local_gzip()` mapping from file name to a sequence of lines.
2. Creating named tuples from the lines of log entries in a collection of files. This was the `access_iter()` mapping from text lines to `Access` objects.
3. Parsing the details of more complex fields such as dates and URLs. This was the `access_detail_iter()` mapping from `Access` objects to `AccessDetails` objects.
4. Rejecting uninteresting paths from the logs. We can also think of this as passing only the interesting paths. This was more of a filter than a map operation. This was a collection of filters bundled into the `path_filter()` function.

We defined an overall `analysis()` function that parsed and analyzed a given log file. It applied the higher-level filter and reduction to the results of the lower-level parsing. It can also work with a wildcard collection of files.

Given the number of mappings involved, we can see several ways to decompose this problem into work designed to use a pool of threads or processes. Each mapping is an opportunity for concurrent processing. Here are some of the mappings we can consider as design alternatives:

- Map the `analysis()` function to individual files. We used this as a consistent example throughout this chapter.
- Refactor the `local_gzip()` function out of the overall `analysis()` function. This refactoring permits mapping a revised `analysis()` function to the results of the `local_gzip()` function.
- Refactor the `access_iter(local_gzip(pattern))` function out of the overall `analysis()` function. This revised `analysis()` function can be applied via `map()` to the iterable sequence of the `Access` objects.
- Refactor the `access_detail_iter(access_iter(local_gzip(pattern)))` function into two separate iterables. This permits using `map()` to apply one function to create `AccessDetail` objects. A separate, higher-level filter and reduction against the iterable sequence of the `AccessDetail` objects can be a separate process.
- We can also refactor the lower-level parsing into a function to keep it separate from

the higher-level analysis. We can map the analysis filter and reduction against the output from the lower-level parsing.

All of these are relatively simple methods to restructure the example application. The benefit of using functional programming techniques is that each part of the overall process can be defined as a mapping, a filter, or a reduction. This makes it practical to consider different architectures to locate an optimal design.

In this case, however, we need to distribute the I/O processing to as many CPUs or cores as we have available. Most of these potential refactorings will perform all of the I/O in the parent process; these will only distribute the computation portions of the work to multiple concurrent processes with little resulting benefit. Because of these, we want to focus on the mappings, as these distribute the I/O to as many cores as possible.

It's often important to minimize the amount of data being passed from process to process. In this example, we provided just short filename strings to each worker process. The resulting `Counter` object was considerably smaller than the 10 MB of compressed detail data in each log file.

It's also essential to run benchmarking experiments to confirm the actual timing between computation, input, and output. This information is essential to uncover optimal allocation of resources, and a design that better balances computation against waiting for I/O to complete.

The following table contains some preliminary results:

Approach	**Duration**
`concurrent.futures/threadpool`	106.58s
`concurrent.futures/processpool`	40.81s
`multiprocessing/imap_unordered`	27.26s
`multiprocessing/map_async`	27.45s

We can see that a thread pool doesn't permit any useful serialization of the work. This is not unexpected, and provides a kind of worst-case benchmark.

The `concurrent.futures/processpool` row shows the time with 4 workers. This variant used the `map()` to parcel requests to the workers. The need to process the work and collect the results in a specific order may have caused relatively slow processing.

The `multiprocessing` modules used the default number of cores, which is 8 for the computer being used. The time was cut almost to $\frac{1}{4}$ the baseline time. In order to make better use of the available processors, it might make sense to further decompose the processing to create batches of lines for analysis, and have separate worker pools for analysis and file parsing. Because the workloads are very difficult to predict, a flexible, functional design allows the restructuring of the work, searching for a way to maximize CPU use.

Summary

In this chapter, we've looked at two ways to support the concurrent processing of multiple pieces of data:

- The `multiprocessing` module: Specifically, the `Pool` class and the various kinds of mappings available to a pool of workers.
- The `concurrent.futures` module: Specifically, the `ProcessPoolExecutor` and `ThreadPoolExecutor` classes. These classes also support a mapping that will distribute work among workers that are threads or processes.

We've also noted some alternatives that don't seem to fit in well with functional programming. There are numerous other features of the `multiprocessing` module, but they're not a good fit with functional design. Similarly, the `threading` and `queue` modules can be used to build multithreaded applications, but the features aren't a good fit with functional programs.

In the next chapter, we'll look at how we can apply functional programming techniques to build web service applications. The idea of HTTP can be summarized as `response = httpd(request)`. When the HTTP processing is stateless, this seems to be a perfect match for functional design.

Adding stateful cookies to this is analogous to providing a response value which is expected

as an argument to a later request. We can think of it as `response, cookie = httpd( request, cookie)`, where the cookie object is opaque to the client.

Exercises

This chapter's exercises are based on code available from Packt Publishing on GitHub. See `https://github.com/PacktPublishing/Functional-Python-Programming-3rd-Edition`.

In some cases, the reader will notice that the code provided on GitHub includes partial solutions to some of the exercises. These serve as hints, allowing the reader to explore alternative solutions.

In many cases, exercises will need unit test cases to confirm they actually solve the problem. These are often identical to the unit test cases already provided in the GitHub repository. The reader should replace the book's example function name with their own solution to confirm that it works.

Lazy parsing

In the *Parsing additional fields of an Access object* section, we looked at a function that did the initial decomposition of a **Common Log File** (**CLF**) line into an initial set of easy-to-separate fields.

We then applied three separate functions to parse the details of the timestamp, request, the time, and the user agent information. These three functions were applied eagerly, decomposing these three fields, even if they were never used for further analysis.

There are two commonly-used ways to implement *lazy* parsing of these fields:

- Rather than parse the text to create a `details.time` attribute, we can define a `parse_time()` method to parse the `access.time` value. The syntax is longer, but it ensures that the attribute is only parsed as needed.
- Once we have this function, we can make it into a property.

First, redefine a new `Access_Details` class to use three separate methods to parse the

complex fields.

Once this works, make these methods into properties to provide values as if they had been parsed eagerly. Make sure the new property method names match the original attribute names in the class shown earlier.

To compare the performance, we need to know how often these additional property parsing methods are used. Two simple assumptions are 100% of the time and 0% of the time. To compare the two designs, we'll need some statistical summary functions that work with the `Access_Details` objects.

Create a function that fetches the values of all attributes, to compute a number of histograms, for example. Create another that uses only the status value to compute a histogram of status only. Compare the performance of the two `Access_Details` class variants and the two analytic approaches to see which is faster. The expectation is that lazy parsing will be faster. The question is "how much faster?"

Filter access path details

In the *Filtering the access details* section of this chapter, we showed a function to exclude empty paths from further analysis.

We can write similar test functions for the `non_excluded_names()` and `non_excluded_ext()` functions. Names like `'favicon.ico'` and `'robots.txt'` need to be excluded. Similarly, extensions like `'.js'` and `'.css'` need to be excluded, also.

Write these two functions to complete the implementation of the `path_filter()` function. These require some unit test cases, as does the overall `path_filter()` function that exploits three separate path function filters.

All of these functions work with a decomposed path name. Is it sensible to try to write a single, complex function for all three operations? Does it make more sense to decompose the three separate rules and combine them through an overall path filtering function?

Add @cache decorators

The implementation of the `path_filter()` function applies three separate filters. Each filter function will parse the path in the `AccessDetails` object. In order to make this more efficient, it can help to wrap lower-level parsing, like a `path.split('/')` function, with the `@cache` decorator.

Write (or rewrite) these three filter functions to make use of the `@cache` decorator.

Be sure to compare performance of the filter functions with caching and without caching. This can be challenging because when we use a simple `@cache` decorator, the original, uncached function is no longer available.

If, on the other hand, we use something like `func_c = cache(func)`, we can preserve both the original (uncached) function and the counterpart with caching. See *Chapter 12*, *Decorator Design Techniques*, for more on how this works. Doing this lets us gather timing data for cached and uncached implementations.

Create sample data

The design shown uses a mapping from filenames to summary counts. Each file is processed concurrently by a pool of workers. In order to determine if this is optimal, it's essential to have a high volume of data to measure performance.

For a lightly-used website, the log files can average about 10 Mb per month. Write a Python script to generate synthetic log rows in batches averaging about 10 Mb per file. Using simplistic random strings isn't the best approach because the application design expects that the request path will have a recognizable pattern. This requires some care to generate synthetic data that fits the expected pattern.

The application to create synthetic data needs some unit test cases. The overall analysis application is the final acceptance test case: does the analysis application identify the data patterns built into the synthetic rows of log entries?

Change the pipeline structure

For a lightly-used website, the log files can average about 10 Mb per month. Using Python 3.10 on a MacBook Pro, each file takes about 16 seconds to process. A collection of six 10 Mb files has a worst-case performance of 96 seconds. On a computer with over six cores, the best case would be 16 seconds.

The design shown in this chapter allocates each file to a separate worker.

Is this the right level of granularity? It's impossible to know without exploring alternatives. This requires sample data files created by the previous exercise. Consider implementing alternative designs and comparing throughput. Here are some suggested alternatives:

- Create two pools of workers: one pool reads files and returns lines in blocks of 1,024. The second pool of workers comprises the bulk of the `analysis()` function. This second pool has workers to parse each line in a block to create an `Access` object, create an `AccessDetails` object, apply the filters, and summarize the results. This leads to two tiers of mapping to pass work from the parsing workers to the analysis workers.
- Decompose the 10 Mb log files into smaller sizes. Write an application to read a log file and write new files, each of which is limited to 4,096 individual log entries. Apply the analysis application to this larger collection of small files instead of the smaller collection of the original large log files.
- Decompose the `analysis()` function to use three separate pools of workers. One pool parses files and returns blocks of `Access` objects. Another pool transforms `Access` objects into `AccessDetails` objects. The third pool of workers applies filters and summarizes the `AccessDetails` objects.

Summarize the results of using distinct processing pipelines to analyze large volumes of data.

15

A Functional Approach to Web Services

We'll step away from the topic of exploratory data analysis to look at web servers and web services. A web server is, to an extent, a cascade of functions. We can apply a number of functional design patterns to the problem of presenting web content. Our goal is to look at ways in which we can approach **Representational State Transfer** (**REST**). We want to build RESTful web services using functional design patterns.

We don't need to invent yet another Python web framework. Nor do we want to select from among the available frameworks. There are many web frameworks available in Python, each with a distinct set of features and advantages.

The intent of this chapter is to present some principles that can be applied to most of the available frameworks. This will let us leverage functional design patterns for presenting web content.

When we look at extremely large or complex datasets, we might want a web service that

supports subsetting or searching. We might also want a website that can download subsets in a variety of formats. In this case, we might need to use functional designs to create RESTful web services to support these more sophisticated requirements.

Interactive web applications often rely on stateful sessions to make the site easier for people to use. A user's session information is updated with data provided through HTML forms, fetched from databases, or recalled from caches of previous interactions. Because the stateful data must be fetched as part of each transaction, it becomes more like an input parameter or result value. This can lead to functional-style programming even in the presence of cookies and database updates.

In this chapter, we'll look at several topics:

- The general idea of the HTTP request and response model.
- The **Web Server Gateway Interface** (**WSGI**) standard that Python applications use.
- Leveraging WSGI, where it's possible to define web services as functions. This fits with the HTTP idea of a stateless server.
- We'll also look at ways to authorize client applications to make use of a web service.

The HTTP request-response model

The HTTP protocol is nearly stateless: a user agent (or browser) makes a request and the server provides a response. For services that don't involve cookies, a client application can take a functional view of the protocol. We can build a client using the `http.client` or `urllib.request` module. An HTTP user agent can be implemented as a function like the following:

```
import urllib.request

def urllib_get(url: str) -> tuple[int, str]:
    with urllib.request.urlopen(url) as response:
        body_bytes = response.read()
```

```
        encoding = response.headers.get_content_charset("utf-8")
        return response.status, body_bytes.decode(encoding)
```

A program like **wget** or **curl** does this kind of processing using a URL supplied as a command-line argument. A browser does this in response to the user pointing and clicking; the URL is taken from the user's actions, often the action of clicking on linked text or images.

Note that a page's encoding is often described in two separate places in the response. The HTTP headers will often name the encoding in use. In this example, the default of `"utf-8"` is supplied in the rare case that the headers are incomplete. In addition, the HTML content can *also* provide encoding information. Specifically, a `<meta charset="utf-8">` tag can claim an encoding. Ideally, it's the same as the encoding noted in the headers. Alternatively, a `<meta http-equiv...>` tag can provide an encoding.

While HTTP processing is stateless, the practical considerations of **user experience** (**UX**) design lead to some implementation details that need to be stateful. For human users to feel comfortable, it's essential for the server to know what they've been doing and retain a transaction state. This is implemented by making the client software (browser or mobile application) track cookies. To make cookies work, a response header provides the cookie data, and subsequent requests must return the saved cookies to the server.

An HTTP response will include a status code. In some cases, this status code will require additional actions on the part of the user agent. Many status codes in the 300-399 range indicate that the requested resource has been moved. The application or browser is then required to save details from the `Location` header and request a new URL. The 401 status code indicates that authentication is required; the user agent must make another request using the `Authorization` header that contains credentials for access to the server. The `urllib` library implementation handles this stateful client processing. The `http.client` library is similar, but doesn't automatically follow 3xx redirect status codes.

Looking at the other side of the protocol, a static content server can be stateless. We can

use the `http.server` library for this, as follows:

```
from http.server import HTTPServer, SimpleHTTPRequestHandler
from typing import NoReturn

def server_demo() -> NoReturn:
    httpd = HTTPServer(
          ('localhost', 8080),
          SimpleHTTPRequestHandler
    )
    print(f"Serving on http://localhost:8080...")
    while True:
        httpd.handle_request()
    httpd.shutdown()
```

We created a `server` object, and assigned it to the `httpd` variable. We provided the address, `localhost`, and port number `8080`. As part of accepting the request, the HTTP protocol will allocate another port; this is used to create an instance of the handler class. Listening on one port but doing the work on other ports allows a server to process numerous requests concurrently.

In this example, we provided `SimpleHTTPRequestHandler` as the class to instantiate with each request. This class must implement a minimal interface, which will send headers and then send the body of the response to the client. This particular class will serve files from the local directory. If we wish to customize this, we can create a subclass that implements methods such as `do_GET()` and `do_POST()` to alter the behavior.

The `HTTPServer` class has a `serve_forever()` method that saves having to write an explicit `while` statement. We've shown the `while` statement here to clarify that the server must, generally, be crashed with an interrupt signal if we need to stop it.

This example uses port number 8080, one that doesn't require elevated privileges. Web servers generally use ports 80 and 443. These require elevated privileges. Generally, it's best to use a server like **NGINX** or Apache **httpd** to manage the privileged ports.

Injecting state through cookies

The addition of cookies changes the overall relationship between a client and server to become stateful. Interestingly, it involves no change to HTTP. The state information is communicated through headers on the request and the reply. The server will send cookies to the user agent in response headers. The user agent will save and reply with cookies in request headers.

The user agent or browser is required to retain a cache of cookie values, provided as part of a response, and include appropriate cookies in subsequent requests. The web server will look for cookies in the request header and provide updated cookies in the response header. The effect is to make the web *server* stateless; the state changes happen only in the *client*. Because a server sees cookies as additional arguments in a request and provides additional details in a response, this shapes our view of the function that responds to a request.

Cookies can contain anything that fits in 4,096 bytes. They are often encrypted to avoid exposing web server details to other applications running on the client computer. Transmitting large cookies can be slow, and should be avoided. The best practice is to keep session information in a database, and provide only a database key in a cookie. This makes the session persistent, and allows session processing to be handled by any available web server, allowing load-balancing among servers.

The concept of a **session** is a feature of the web application software, not HTTP. A session is commonly implemented via a cookie to retain session information. When an initial request is made, no cookie is available, and a new session cookie is created. Every subsequent request will include the cookie's value. A logged-in user will have additional details in their session cookie. A session can last as long as the server is willing to accept the cookie; a cookie could be valid forever, or expire after a few minutes.

A RESTful approach to web services does *not* rely on sessions or cookies. Each REST request is distinct. In many cases, an `Authorization` header is provided with each request to provide credentials for authentication and authorization. This generally means that a separate client-facing application must create a pleasing user experience, often involving

sessions. A common architecture is a front-end application, perhaps a mobile app or browser-based site to provide a view of the supporting RESTful web services.

We'll focus on RESTful web services in this chapter. The RESTful approach fits well with stateless functional design patterns.

One consequence of sessionless REST processes is each individual REST request is separately authenticated. This generally means the REST service must also use **Secure Socket Layer** (**SSL**) protocols. The `HTTPS` scheme is required to transmit credentials securely from client to server.

Considering a server with a functional design

One core idea behind HTTP is that the server's response is a function of the request. Conceptually, a web service should have a top-level implementation that can be summarized as follows:

```
response = httpd(request)
```

While this is the essence of HTTP, it lacks a number of important details. First, an HTTP request isn't a simple, monolithic data structure. It has some required parts and some optional parts. A request may have headers, a method (e.g., `GET`, `POST`, `PUT`, `PATCH`, etc.), a URL, and there may be attachments. The URL has several optional parts including a path, a query string, and a fragment identifier. The attachments may include input from HTML forms or uploaded files, or both.

Second, the response, similarly, has three parts to it. It has a status code, headers, and a response body. Our simplistic model of a `httpd()` function doesn't cover these additional details.

We'll need to expand on this simplistic view to more accurately decompose web processing into useful functions.

Looking more deeply into the functional view

Both HTTP responses and requests have headers that are separate from the body. The request can also have some attached form data or other uploads. Therefore, we can more usefully think of a web server like this:

```
headers, content = httpd(
    headers, request, [attachments, either forms or uploads]
)
```

The request headers may include cookie values, which can be seen as adding more arguments. Additionally, a web server is often dependent on the OS environment in which it's running. This OS environment data can be considered as yet more arguments being provided as part of the request.

The **Multipurpose Internet Mail Extension** (**MIME**) types define the kinds of content that a web service might return. MIME describes a large but reasonably well-defined spectrum of content. This can include plain text, HTML, JSON, XML, or any of the wide variety of non-text media that a website might serve.

There are some common features of HTTP request processing that we'd like to reuse. This idea of reusable elements is what leads to the creation of web service frameworks that fill a spectrum from simple to sophisticated. The ways that functional designs allow us to reuse functions indicate that the functional approach can help in building web services.

We'll look at functional design of web services by examining how we can create a pipeline of the various elements of a service response. We'll do this by nesting the functions for request processing so that inner elements are free from the generic overheads, which are provided by outer elements. This also allows the outer elements to act as filters: invalid requests can yield error responses, allowing the inner function to focus narrowly on the application processing.

Nesting the services

We can look at web request-handling as a number of layered contexts. The foundation might cover session management: examining the request to determine if this is another request in an existing session or a new session. Built on this foundation, another layer can provide tokens used for form processing that can detect **Cross-Site Request Forgeries** (**CSRF**). Another layer on top of these might handle user authentication within a session.

A conceptual view of the functions explained previously is something like this:

```
response = content(
    authentication(
        csrf(
            session(headers, request, forms)
        )
    )
)
```

The idea here is that each function can build on the results of the previous function. Each function either enriches the request or rejects it because it's invalid. The `session()` function, for example, can use headers to determine if this is an existing session or a new session. The `csrf()` function will examine form input to ensure that proper tokens were used. The CSRF handling requires a valid session. The `authentication()` function can return an error response for a session that lacks valid credentials; it can enrich the request with user information when valid credentials are present.

The `content()` function is free from worrying about sessions, forgeries, and non-authenticated users. It can focus on parsing the path to determine what kind of content should be provided. In a more complex application, the `content()` function may include a rather complex mapping from path elements to the functions that determine the appropriate content.

This nested function view suffers from a profound problem. The stack of functions is defined to be used in a specific order. The `csrf()` function must be done first to provide useful

information to the `authentication()` function. However, we can imagine a high-security scenario where authentication must be done before the CSRF tokens can be checked. We don't want to have to define unique functions for each possible web architecture.

While each context must have a distinct focus, it would be more helpful to have a single, unified view of request and response processing. This allows pieces to be built independently. A useful website would be a composition of a number of disparate functions.

With a standardized interface, we can combine functions to implement the required features. This will fit the functional programming objectives of having succinct and expressive programs that provide web content. The WSGI standard provides a uniform way to build complex services as a composition of parts.

The WSGI standard

The **Web Server Gateway Interface** (**WSGI**) defines a standard interface for creating a response to a web request. This is a common framework for most Python-based web servers. A great deal of information is present at the following link: `http://wsgi.readthedocs.org/en/latest/`.

Some important background on WSGI can be found at `https://www.python.org/dev/peps/pep-0333/`.

The Python library's `wsgiref` package includes a reference implementation of WSGI. Each WSGI **application** has the same interface, as shown here:

```
def some_app(environ, start_response):
    # compute the status, headers, and content of the response
    start_response(status, headers)
    return content
```

The `environ` parameter is a dictionary that contains all of the arguments of the request in a single, uniform structure. The headers, the request method, the path, and any attachments for forms or file uploads will all be in the environment dictionary. In addition to this, the

OS-level context is also provided, along with a few items that are part of WSGI request handling.

The `start_response` parameter is a function that must be used to send the status and headers of a response. The portion of a WSGI server that has the final responsibility for building the response will use the given `start_response()` function and will also build the response document as the return value.

The response returned from a WSGI application is a sequence of strings or string-like file wrappers that will be returned to the user agent. If an HTML template tool is used, then the sequence may have a single item. In some cases, such as using the **Jinja2** templates to build HTML content, the template can be rendered lazily as a sequence of text chunks. This allows a server to interleave template filling with downloading to the user agent.

The `wsgiref` package does not have a complete set of type definitions. This is not a problem in general. For example, within the `werkzeug` package, the `werkzeug.wsgi` module has useful type definitions. Because the `werkzeug` package is generally installed with Flask, it is very handy for our purposes.

The `werkzeug.wsgi` module includes a stubs file with a number of useful type hints. These hints are not part of the working application; they're only used by the **mypy** tool. We can study the following `werkzeug.wsgi` type hints for a WSGI application:

```
from sys import _OptExcInfo
from typing import Any, Callable, Dict, Iterable, Protocol

class StartResponse(Protocol):
    def __call__(
        self, status: str, headers: list[tuple[str, str]], exc_info:
        "_OptExcInfo" | None = ...
    ) -> Callable[[bytes], Any]: ...

WSGIEnvironment = Dict[str, Any]
WSGIApplication = Callable[[WSGIEnvironment, StartResponse],
Iterable[bytes]]
```

The `WSGIEnvironment` type hint defines a dictionary with no useful boundaries on the values. It's difficult to enumerate all of the possible types of values defined by the WSGI standard. Instead of an exhaustively complex definition, it seems better to use `Any`.

The `StartResponse` type hint is the signature for the `start_response()` function provided to a WSGI application. This is defined as a `Protocol` to show the presence of an optional third parameter with exception information.

An overall WSGI application, `WSGIApplication`, requires the environment and the `start_response()` function. The result is an iterable collection of bytes.

The idea behind these hints is to allow us to define an application as follows:

```
from typing import TYPE_CHECKING

if TYPE_CHECKING:
    from _typeshed.wsgi import (
        WSGIApplication, WSGIEnvironment, StartResponse
    )

def static_text_app(
    environ: "WSGIEnvironment",
    start_response: "StartResponse"
) -> Iterable[bytes]:
    ...
```

We've included a conditional `import` to provide the type hints only when running the **mypy** tool. Outside using the **mypy** tool, the type hints are provided as strings. This additional clarification can help explain the design of a complex collection of functions that respond to web requests.

Each WSGI application needs to be designed as a collection of functions. The collection can be viewed as nested functions or as a chain of transformations. Each application in the chain will either return an error or will hand the request to another application that will determine the final result.

Often, the URL path is used to determine which of many alternative applications will be used. This will lead to a tree of WSGI applications that may share common components.

Here's a very simple routing application that takes the first element of the URL path and uses this to locate another WSGI application that provides content:

```
from wsgiref.simple_server import demo_app

SCRIPT_MAP: dict[str, "WSGIApplication"] = {
    "demo": demo_app,
    "static": static_text_app,
    "index.html": welcome_app,
    "": welcome_app,
}

def routing(
        environ: "WSGIEnvironment",
        start_response: "StartResponse"
) -> Iterable[bytes]:
    top_level = wsgiref.util.shift_path_info(environ)
    if top_level:
        app = SCRIPT_MAP.get(top_level, welcome_app)
    else:
        app = welcome_app
    content = app(environ, start_response)
    return content
```

This application will use the `wsgiref.util.shift_path_info()` function to tweak the environment. The change is a *head/tail split* on the request path, available in the `environ['PATH_INFO']` dictionary. The head of the path, up to the first `"/"`, will be assigned to the `SCRIPT_NAME` item in the environment; the `PATH_INFO` item will be updated to have the tail of the path. The returned value will also be the head of the path, the same value as `environ['SCRIPT_NAME']`. In the case where there's no path to parse, the return value is `None` and no environment updates are made.

The `routing()` function uses the first item on the path to locate an application in the

`SCRIPT_MAP` dictionary. We use `welcome_app` as a default in case the requested path doesn't fit the mapping. This seems a little better than an HTTP `404 NOT FOUND` error.

This WSGI application is a function that chooses between a number of other WSGI functions. Note that the routing function doesn't return a function; it provides the modified environment to the selected WSGI application. This is the typical design pattern for handing off the work from one function to another.

From this, we can see how a framework could generalize the path-matching process, using regular expressions. We can imagine configuring the `routing()` function with a sequence of regular expressions and WSGI applications, instead of a mapping from a string to the WSGI application. The enhanced `routing()` function application would evaluate each regular expression looking for a match. In the case of a match, any `match.groups()` function could be used to update the environment before calling the requested application.

Raising exceptions during WSGI processing

One central feature of WSGI applications is that each stage along the chain is responsible for filtering the requests. The idea is to reject faulty requests as early in the processing as possible. When building a pipeline of independent WSGI applications, each stage has the following two essential choices:

- Evaluate the `start_response()` function to start a reply with an error status
- OR pass the request with an expanded environment to the next stage

Consider a WSGI application that provides small text files. A file may not exist, or a request may refer to a directory of files. We can define a WSGI application that provides static content as follows:

```
def headers(content: bytes) -> list[tuple[str, str]]:
    return [
        ("Content-Type", 'text/plain;charset="utf-8"'),
        ("Content-Length", str(len(content))),
    ]
```

```
def static_text_app(
    environ: "WSGIEnvironment",
    start_response: "StartResponse"
) -> Iterable[bytes]:
    log = environ['wsgi.errors']
    try:
        static_path = Path.cwd() / environ['PATH_INFO'][1:]
        with static_path.open() as static_file:
            print(f"{static_path=}", file=log)
            content = static_file.read().encode("utf-8")
            start_response('200 OK', headers(content))
            return [content]
    except IsADirectoryError as exc:
        return index_app(environ, start_response)
    except FileNotFoundError as exc:
        print(f"{static_path=} {exc=}", file=log)
        message = f"Not Found {environ['PATH_INFO']}".encode("utf-8")
        start_response('404 NOT FOUND', headers(message))
        return [message]
```

This application creates a `Path` object from the current working directory and an element of the path provided as part of the requested URL. The path information is part of the WSGI environment, in an item with the `'PATH_INFO'` key. Because of the way the path is parsed, it will have a leading "/", which we discard by using `environ['PATH_INFO'][1:]`.

This application tries to open the requested path as a text file. There are two common problems, both of which are handled as exceptions:

- If the file is a directory, we'll route the request to a different WSGI application, `index_app`, to present directory contents
- If the file is simply not found, we'll return an HTTP `404 NOT FOUND` response

Any other exceptions raised by this WSGI application will not be caught. The application that invoked this application should be designed with some generic error-response capability. If the application doesn't handle the exceptions, a generic WSGI failure response will be used.

Our processing involves a strict ordering of operations. We must read the entire file so that we can create a proper HTTP Content-Length header.

This small application shows the WSGI idea of either responding or passing the request onto another application that forms the response. This respond-now-or-forward design pattern enables the building of multi-stage pipelines. Each stage either rejects the request, handles it completely, or passes it on to some other application.

These pipelines are often called **middleware** because they are between a base server (like **NGINX**) and the final web application or RESTful API. The idea is to use middleware to perform a series of common filters or mappings for each request.

Pragmatic web applications

The intent of the WSGI standard is not to define a complete web framework; the intent is to define a minimum set of standards that allows flexible interoperability of web-related processing. This minimum fits well with functional programming concepts.

A web application framework is focused on the needs of developers. It should offer numerous simplifications to providing web services. The foundational interface must be compatible with WSGI, so that it can be used in a variety of contexts. The developer's view, however, will diverge from the minimal WSGI definitions.

Web servers such as **Apache httpd** or **NGINX** have adapters to provide a WSGI-compatible interface from the web server to Python applications. For more information on WSGI implementations, visit `https://wiki.python.org/moin/WSGIImplementations`.

Embedding our applications in a larger server allows us to have a tidy separation of concerns. We can use Apache **httpd** or **NGINX** to serve the static content, such as `.css`, `.js`, and image files. For HTML pages, though, a server like **NGINX** can use the `uwsgi` module to hand off requests to a pool of Python processes. This focuses Python on handling the interestingly complex HTML portions of the web content.

Downloading static content requires little customization. There's often no application-

specific processing. This is best handled in a separate service that can be optimized to perform this fixed task.

The processing for dynamic content (often the HTML content of a web page) is where the interesting Python-based work happens. This work can be segregated to servers that are optimized to run this more complex application-specific computation.

Separating the static content from the dynamic content to provide optimized downloads means that we must either create a separate media server, or define our website to have two sets of paths. For smaller sites, a separate `/media` path works out nicely. For larger sites, distinct media servers are required.

An important consequence of the WSGI definition is the `environ` dictionary is often updated with additional configuration parameters. In this way, some WSGI applications can serve as gateways to enrich the environment with information extracted from cookies, headers, configuration files, or databases.

Defining web services as functions

We'll look at a RESTful web service, which can slice and dice a source of data and provide downloads as JSON, XML, or CSV files.

The direct use of WSGI for this kind of application isn't optimal because we need to create a great deal of "boilerplate" processing for all the details of conventional website processing. A more effective approach is to use a more sophisticated web server like Flask, Django, Bottle, or any of the frameworks listed here: `https://wiki.python.org/moin/WebFrameworks`. These servers handle the conventional cases more completely, allowing us—as developers—to focus on the unique features of a page or site.

We'll use a simple dataset with four series of data pairs: the Anscombe Quartet. We looked at ways to read and parse this data in *Chapter 3, Functions, Iterators, and Generators*. It's a small set of data, but it can be used to show the principles of a RESTful web service.

We'll split our application into two tiers: a web tier, which will provide the visible RESTful web service, and a data service tier, which will manage the underlying data. We'll look at

the web tier first, as this provides a context in which the data service tier must operate.

A request must include these two pieces of information:

- The series of data that is desired. The idea is to slice up the pool of available information by filtering and extracting the desired subset.
- The output format that the user needs. This includes common serialization formats like HTML, CSV, JSON, and XML.

The series selection is commonly done through the request path. We can request `/anscombe/I` or `/anscombe/II` to pick specific series from the quartet. Path design is important, and this seems to be the right way to identify the data.

The following two underlying ideas help define paths:

- A URL defines a resource
- There's no good reason for the URL to ever change

In this case, the dataset selectors of `I` or `II` aren't dependent on publication dates or some organizational approval status, or other external factors. This design seems to create URLs that are timeless and absolute.

The output format, on the other hand, is not a first-class part of the URL. It is merely a serialization format, not the data itself. One choice is to name the format in the HTTP `Accept` header. In some cases, to make things easy to use from a browser, a query string can be used to specify the output format. One approach is to use the query to specify the serialization format. We might use `?form=json`, `?format=json`, or even `?output_serialization=json` at the end of the path to specify that the output serialization format should be JSON. The HTTP `Accept` header is preferred, but hard to experiment with using only a browser.

A browser-friendly URL we can use will look like this:

```
http://localhost:8080/anscombe/III?form=csv
```

This would request a download of the third series in CSV format.

The OpenAPI Specification provides a way to define the family of URLs and the expected results. This specification is helpful because it serves as a clear, formal contract for the web server's expected behavior. What's most helpful about the OpenAPI specification is having a concrete list of paths, parameters, and responses. A good specification will include examples, helping the process of writing an acceptance test suite for the server.

Generally, the OpenAPI specification is provided by the web server to help clients properly use the available services. A URL like `"/openapi.yml"` or `"/openapi.json"` is suggested as a way to provide needed information about a web application.

Flask application processing

We'll use the Flask framework because it provides an easy-to-extend web services process. It supports a function-based design, with a mapping from a request path to a view function that builds the response. The framework also makes use of decorators, providing a good fit with functional programming concepts.

In order to bind all of the configuration and URL routing together, an overall `Flask` instance is used as a container. Our application will be an instance of the `Flask` class. As a simplification, each view function is defined separately and bound into the `Flask` instance via a routing table that maps URLs to functions. This routing table is built via decorators.

The core of the application is this collection of view functions. Generally, each view function needs to do three things:

1. Validate the request.
2. Perform the requested state change or data access.
3. Prepare a response.

Ideally, the view function does nothing more than this.

Here's the initial `Flask` object that will contain the routes and their functions:

```
from flask import Flask

app = Flask(__name__)
```

We've created the `Flask` instance and assigned it to the `app` variable. As a handy default, we've used the module's name, `__name__`, as the name of the application. This is often sufficient. For complex applications, it may be better to provide a name that's not specifically tied to a Python module or package name.

Most applications will need to have configuration parameters provided. In this case, the source data is a configurable value that might change.

For larger applications, it's often necessary to locate an entire configuration file. For this small application, we'll provide the configuration value as a literal:

```
from pathlib import Path

app.config['FILE_PATH'] = Path.cwd() / "Anscombe.txt"
```

Most of the view functions should be relatively small, focused functions that make use of other layers of the application. For this application, the web presentation depends on a data service tier to acquire and format the data. This leads to functions with the following three steps:

1. Validate the various inputs. This includes validating items like the path, any query parameters, form input data, uploaded files, header values, and even cookie values.
2. If the method involves a state change like `POST`, `PUT`, `PATCH`, or `DELETE`, perform the state-changing operation. These will often return a "redirect" response to a path that will display the results of the change. If the method involves a `GET` request, gather the requested data.
3. Prepare the response.

What's important about step 2 is all of the data manipulation is separate from the RESTful

web application. The web presentation sits on a foundation of data access and manipulation. The web application is designed as a view or a presentation of the underlying structure.

We'll look at two URL paths for the web application. The first path will provide an index of the available series in the Anscombe collection. The `view` function can be defined as follows:

```
from flask import request, abort, make_response, Response

@app.route("/anscombe/")
def index_view() -> Response:
    # 1. Validate
    response_format = format()
    # 2. Get data
    data = get_series_map(app.config['FILE_PATH'])
    index_listofdicts = [{"Series": k} for k in data.keys()]
    # 3. Prepare Response
    try:
        content_bytes = serialize(response_format, index_listofdicts,
        document_tag="Index", row_tag="Series")
        response = make_response(content_bytes, 200, {"Content-Type":
        response_format})
        return response
    except KeyError:
        abort(404, f"Unknown {response_format=}")
```

This function has the Flask `@app.route` decorator. This shows what URLs should be processed by this `view` function. There are a fair number of options and alternatives available here. The `view` function will be evaluated when a request matches one of the available routes.

The `format()` function definition will be shown in a little while. It locates the user's desired format by looking in two places: the query string, after the `?` in the URL, and also in the `Accept` header. If the query string value is invalid, a 404 response will be created.

The `get_series_map()` function is an essential feature of the data service tier. This will

locate the Anscombe series data and create a mapping from `Series` name to the data of the series.

The index information is in the form of a list-of-dict structure. This structure can be converted to JSON, CSV, and HTML without too much complication. Creating XML is a bit more difficult. The difficulty arises because the Python list and dictionary objects don't have any specific class name, making it awkward to supply XML tags.

The data preparation is performed in two parts. First, the index information is serialized in the desired format. Second, a Flask `Response` object is built using the bytes, an HTTP status code of 200, and a specific value for the `Content-Type` header.

The `abort()` function stops process and returns an error response with the given code and reason information. For RESTful web services, it helps to add a small helper function to transform the result into JSON. The use of the `abort()` function during data validation and preparation makes it easy to end processing at the first problem with the request.

The `format()` function is defined as follows:

```
def format() -> str:
    if arg := request.args.get('form'):
        try:
            return {
                'xml': 'application/xml',
                'html': 'text/html',
                'json': 'application/json',
                'csv': 'text/csv',
            }[arg]
        except KeyError:
            abort(404, "Unknown ?form=")
    else:
        return request.accept_mimetypes.best or "text/html"
```

This function looks for input from two attributes of the `request` object:

- The `args` will have the argument values that are present after the "?" in the URL

- The `accept_mimetypes` will have the parsed values from the `Accept` header, allowing an application to locate a response that meets the client's expectations

The `request` object is a bit of thread-local storage with the details of the web request being made. It is used like a global variable, making some functions look a little awkward. The use of a global like `request` tends to obscure the actual parameters to this function. Using explicit parameters requires also providing the underlying type information, which is little more than visual clutter.

The `series_view()` function to provide series data is defined as follows:

```
@app.route("/anscombe/<series_id>")
def series_view(series_id: str, form: str | None = None) -> Response:
    # 1. Validate
    response_format = format()
    # 2. Get data (and validate some more)
    data = get_series_map(app.config['FILE_PATH'])
    try:
        dataset = anscombe_filter(series_id, data)._as_listofdicts()
    except KeyError:
        abort(404, "Unknown Series")
    # 3. Prepare Response
    try:
        content_bytes = serialize(response_format, dataset,
        document_tag="Series", row_tag="Pair")
        response = make_response(
            content_bytes, 200, {"Content-Type": response_format}
        )
        return response
    except KeyError:
        abort(404, f"Unknown {response_format=}")
```

This function has a similar structure to the previous `index_view()` function. The request is validated, the data acquired, and a response prepared. As with the previous function, the work is delegated to two other data access functions: `get_series_map()` and `anscombe_filter()`. These are separate from the web application, and could be part of a

command-line application.

Both of these functions depend on an underlying data access layer. We'll look at those functions in the next section.

The data access tier

The `get_series_map()` function is similar to the examples shown in the *Cleaning raw data with generator functions* section of *Chapter 3, Functions, Iterators, and Generators*. In this section, we'll include some important changes. We'll start with the following two `NamedTuple` definitions:

```
from Chapter03.ch03_ex4 import (
    series, head_split_fixed, row_iter)
from collections.abc import Callable, Iterable
from typing import NamedTuple, Any, cast

class Pair(NamedTuple):
    x: float
    y: float

    @classmethod
    def create(cls: type["Pair"], source: Iterable[str]) -> "Pair":
        return Pair(*map(float, source))

class Series(NamedTuple):
    series: str
    data: list[Pair]

    @classmethod
    def create(cls: type["Series"], name: str, source:
    Iterable[tuple[str, str]]) -> "Series":
        return Series(name, list(map(Pair.create, source)))

    def _as_listofdicts(self) -> list[dict[str, Any]]:
        return [p._asdict() for p in self.data]
```

We've defined a `Pair` named tuple and provided a `@classmethod` to build instances of a

`Pair`. This definition will automatically provide an `_asdict()` method that responds with a dictionary of the form `dict[str, Any]` containing the attribute names and values. This is helpful for serialization.

Similarly, we've defined a `Series` named tuple. The `create()` method can build a tuple from an iterable source of lists of values. The automatically provided `_asdict()` method can be helpful for serializing. For this application, however, we'll make use of the `_as_listofdicts` method to create a list of dictionaries that can be serialized.

The function to produce the mapping from series name to `Series` object has the following definition:

```
from pathlib import Path

def get_series_map(source_path: Path) -> dict[str, Series]:
    with source_path.open() as source:
        raw_data = list(head_split_fixed(row_iter(source)))
        series_iter = (
            Series.create(id_str, series(id_num, raw_data))
            for id_num, id_str in enumerate(
                ['I', 'II', 'III', 'IV'])
        )
        mapping = {
            series.series: series
            for series in series_iter
        }
    return mapping
```

The `get_series_map()` function opens the local data file, and applies the `row_iter()` function to each line of the file. This parses the line into a row of separate items. The `head_split_fixed()` function is used to remove the heading from the file. The result is a tuple-of-list structure, which is assigned the variable `raw_data`.

From the `raw_data` structure, the `Series.create()` method is used to transform a sequence of values from the file into a `Series` object composed of individual `Pair` instances. The final step is to use a dictionary comprehension to collect the individual `Series` instances

into a single mapping from series name to `Series` object.

Since the output from the `get_series_map()` function is a mapping, we can do something like the following example to pick a specific series by name:

```
>>> source = Path.cwd() / "Anscombe.txt"
>>> get_series_map(source)['I']
Series(series='I', data=[Pair(x=10.0, y=8.04), Pair(x=8.0, y=6.95),
...])
```

Given a key, for example, `'I'`, the series is a list of `Pair` objects that have the `x`, `y` values for each item in the series.

Applying a filter

In this application, we're using a very simple filter. The entire filter process is embodied in the following function:

```
def anscombe_filter(
    set_id: str, raw_data_map: dict[str, Series]
) -> Series:
    return raw_data_map[set_id]
```

We made this trivial expression into a function for three reasons:

- The functional notation is slightly more consistent with other parts of the Flask application, and a bit more flexible than the subscript expression
- We can easily expand the filtering to do more
- We can include separate unit tests for this function

While a simple lambda would work, it wouldn't be quite as convenient to test.

For error handling, we've done exactly nothing. We've focused on what's sometimes called the *happy path*: an ideal sequence of events. Any problems that arise in this function will raise an exception. The WSGI wrapper function should catch all exceptions and return an appropriate status message and error response content.

For example, it's possible that the `set_id` method will be wrong in some way. Rather than obsess over all the ways it could be wrong, we'll allow Python to raise an exception. Indeed, this function follows Admiral Grace Murray Hopper's advice that *it's better to seek forgiveness than to ask permission.* This advice is materialized in code by avoiding *permission-seeking*: there are no preparatory `if` statements that seek to qualify the arguments as valid. There is only *forgiveness* handling: an exception will be raised and handled by evaluating the Flask `abort()` function.

Serializing the results

Serialization is the conversion of Python data into a stream of bytes, suitable for transmission. Each format is best described by a simple function that serializes just that one format. A top-level generic serializer can then pick from a list of specific serializers.

The general type hint for a serializer is this:

```
from collections.abc import Callable
from typing import Any, TypeAlias

Serializer: TypeAlias = Callable[[list[dict[str, Any]]], bytes]
```

This definition avoids the specific `Series` definition. It uses a more general `list[dict[str, Any]]` type hint. This can be applied to the data of a `Series` as well as other items like the series labels.

A mapping from MIME types to serializer functions will lead to the following mapping object:

```
SERIALIZERS: dict[str, Serializer] = {
    'application/xml': serialize_xml,
    'text/html': serialize_html,
    'application/json': serialize_json,
    'text/csv': serialize_csv,
}
```

This variable will be defined after the four functions it references. We've provided it here to act as context, showing where the serialization design is headed.

The top-level `serialize()` function can be defined as follows:

```
def serialize(
    format: str | None,
    data: list[dict[str, Any]],
    **kwargs: str
) -> bytes:
    """Relies on global SERIALIZERS, set separately"""
    if format is None:
        format = "text/html"
    function = SERIALIZERS.get(
        format.lower(),
        serialize_html
    )
    return function(data, **kwargs)
```

The overall `serialize()` function locates a specific serializer in the `SERIALIZERS` dictionary. This specific function fits the the `Serializer` type hint. The function will transform a `Series` object into bytes that can be downloaded to a web client application.

The `serialize()` function doesn't do any data transformation. It maps a MIME type string to a function that does the hard work of transformation.

We'll look at some of the individual serializers below. It's relatively common for Python processing to create strings. We can then encode the strings into bytes. To avoid repeating the encoding operation, we'll define a decorator to compose the serialization with the bytes encoding. Here's the decorator we can use:

```
from collections.abc import Callable
from typing import TypeVar, ParamSpec
from functools import wraps

T = TypeVar("T")
```

```
P = ParamSpec("P")

def to_bytes(
    function: Callable[P, str]
) -> Callable[P, bytes]:
    @wraps(function)
    def decorated(*args: P.args, **kwargs: P.kwargs) -> bytes:
        text = function(*args, **kwargs)
        return text.encode("utf-8")
    return decorated
```

We've created a small decorator named `@to_bytes`. This will evaluate the given function and then encode the results using UTF-8 to get bytes. Note that the decorator changes the decorated function from having a return type of `str` to a return type of `bytes`. We used the `ParamSpec` hint to collect declared parameters for the decorated function. This ensures that tools like **mypy** can match the parameter specification for the decorated function with the base function.

We'll show how this is used with JSON and CSV serializers. The HTML and XML serialization involves a bit more programming, but no significant complexity.

Serializing data with JSON or CSV formats

The JSON and CSV serializers are similar because both rely on Python's libraries to serialize. The libraries are inherently imperative, so the function bodies are sequences of statements.

Here's the JSON serializer:

```
import json

@to_bytes
def serialize_json(data: list[dict[str, Any]], **kwargs: str) -> str:
    text = json.dumps(data, sort_keys=True)
    return text
```

We created a list-of-dicts structure and used the `json.dumps()` function to create a string

representation. The JSON module requires a materialized list object; we can't provide a lazy generator function. The `sort_keys=True` argument value is helpful for unit testing because the order is clearly stated and can be used to match expected results. However, it's not required for the application and represents a bit of overhead.

Here's the CSV serializer:

```
import csv
import io

@to_bytes
def serialize_csv(data: list[dict[str, Any]], **kwargs: str) -> str:
    buffer = io.StringIO()
    wtr = csv.DictWriter(buffer, sorted(data[0].keys()))
    wtr.writeheader()
    wtr.writerows(data)
    return buffer.getvalue()
```

The `csv` module's readers and writers are a mixture of imperative and functional elements. We must create the writer, and properly create headings in a strict sequence. A client of this function can use the `_fields` attribute of the `Pair` named tuple to determine the column headings for the writer.

The `writerows()` method of the writer will accept a lazy generator function. A client of this function can use the `_asdict()` method of a `NamedTuple` object to return a dictionary suitable for use with the CSV writer.

Serializing data with XML and HTML

Serialization into XML has a goal of creating a document that looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<Series>
<Pair><x>2</x><y>3</y></Pair>
<Pair><x>5</x><y>7</y></Pair>
</Series>
```

This XML document doesn't include a reference to formal **XML Schema Definition** (**XSD**). It is, however, designed to parallel the named tuple definitions shown above.

One way to produce a document like this is to create a template and fill in the fields. This can be done with packages like Jinja or Mako. There are a number of sophisticated template tools to create XML or HTML pages. A number of these include the ability to embed iteration over a sequence of objects—like a list of dicts—in the template, separate from the function that initializes serialization. Visit `https://wiki.python.org/moin/Templating` for a list of alternatives.

A more sophisticated serialization library could be helpful here. There are many to choose from. Visit `https://wiki.python.org/moin/PythonXml` for a list of alternatives.

Modern HTML is based on XML. Therefore, an HTML document can be built similarly to an XML document by filling the actual values into a template. HTML documents often have a great deal more overhead than XML documents. The additional complexity arises because in HTML, the document is expected to provide an entire web page with a great deal of context information.

We've omitted the details for creating HTML or XML, leaving them as exercises for the reader.

Tracking usage

RESTful APIs need to be used for secured connections. This means the server must use SSL, and the connection will be via HTTPS protocol. The idea is to manage the SSL certificates used by "front-end" or client applications. In many web service environments, mobile applications and JavaScript-based interactive front-ends will have certificates allowing access to the back-end.

In addition to SSL, another common practice is to require an **API key** as part of each transaction. An API key can be used to authenticate access. It may also be used to authorize specific features. Most importantly, it's essential for tracking actual usage. A consequence of tracking usage can be throttling requests if an API key is used too often in a given time

period.

The variations in business models are numerous. For example, use of the API key could be a billable event and charges will be incurred. For other businesses, traffic must reach some threshold before payments are required.

What's important is non-repudiation of the use of the API. When a transaction is executed to make a state change, the API key can be used to identify the application making the request. This, in turn, means creating API keys that can act as a user's authentication credentials. The key must be difficult to forge and relatively easy to verify.

One way to create API keys is to use a cryptographic random number to generate a difficult-to-predict key string. The `secrets` module can be used to generate unique API key values. Here's an example of generating a unique key that can be assigned to clients to track activity:

```
>>> import secrets
>>> secrets.token_urlsafe(24)
'NLHirCPVf-S7aSAiaAJo3JECYk9dSeyq'
```

A base 64 encoding is used on the random bytes to create a sequence of characters. Using a multiple of three for the length will avoid any trailing `=` signs in the base 64 encoding. We've used the URL-safe base 64 encoding, which won't include the `/` or `+` characters in the resulting string. This means the key can be used as part of a URL or can be provided in a header.

A more elaborate method of generating a token won't lead to more random data. The use of the `secrets` module assures that it is very difficult to counterfeit a key assigned to another user.

The `secrets` module is notoriously hard to use as part of unit and integration test. In order to produce high-quality, secure values, it avoids having an explicit seed like the `random` module does. Since reproducible unit test cases

can't depend on the `secrets` module having reproducible results, a mock object should be used when testing. One consequence of this is creating a design that facilitates testing.

As API keys are generated, they need to be sent to the users creating applications, and also kept in a database that's part of the API service.

If a request includes a key that's in the database, the associated user is responsible for the request. If the API request doesn't include a known key, the request can be rejected with a `401 UNAUTHORIZED` response.

This small database can be a text file that the server loads to map API keys to authorized privileges. The file can be read at startup and the modification time checked to see if the version cached in the server is still current. When a new key is available, the file is updated and the server will re-read the file.

See `https://swagger.io/docs/specification/2-0/authentication/api-keys/` for more information on API keys.

The essential check for a valid API key is so common that Flask provides a decorator to identify this function. Using `@app.before_app_request` marks a function that will be invoked *before* every view function. This function can establish the validity of the API key before allowing any processing.

This API key-checking is often bypassed for a few paths. If, for example, the service will download its OpenAPI specification, the path should be handled without regard to the presence of an `API-Key` header. This often means a special-case check to see if `request.path` is `openapi.json` or one of the other common names for the specification.

Similarly, a server may need to respond to requests based on the presence of CORS headers. See `https://www.w3.org/TR/cors/#http-cors-protocol` for more information. This can make the `before_app_request()` function even more complex by adding another group

of exceptions.

The good news is there are only two exceptions to requiring an `API-Key` header with every request. One is handling the OpenAPI specification and the other is the CORS preflight request. This is unlikely to change, and a few `if` statements are sufficient.

Summary

In this chapter, we looked at ways in which we can apply functional design to the problem of serving content with REST-based web services. We looked at how the WSGI standard leads to somewhat functional overall applications. We also looked at how we can embed a more functional design into a WSGI context by extracting elements from the request for use by our application functions.

For simple services, the problem often decomposes into three distinct operations: getting the data, searching or filtering, and then serializing the results. We tackled this with three functions: `raw_data()`, `anscombe_filter()`, and `serialize()`. We wrapped these functions in a simple WSGI-compatible application to divorce the web services from the *real* processing around extracting and filtering the data.

We also looked at the way that web services' functions can focus on the *happy path* and assume that all of the inputs are valid. If inputs are invalid, the ordinary Python exception handling will raise exceptions. The WSGI wrapper function will catch the errors and return appropriate status codes and error content.

We have not looked at more complex problems associated with uploading data or accepting data from forms to update a persistent data store. These are not significantly more complex than getting data and serializing the results.

For simple queries and data sharing, a small web service application can be helpful. We can apply functional design patterns and assure that the website code is succinct and expressive. For more complex web applications, we should consider using a framework that handles the details properly.

In the next chapter (available on GitHub at `https://github.com/PacktPublishing/Fun`

ctional-Python-Programming-3rd-Edition/blob/main/Bonus_Content/Chapter_16.pdf), we'll look at a more complete example of functional programming. This is a case study that applies some statistical measures to sample data to determine if the data are likely to be random, or potentially include some interesting relationship.

Exercises

This chapter's exercises are based on code available from Packt Publishing on GitHub. See https://github.com/PacktPublishing/Functional-Python-Programming-3rd-Edition.

In some cases, the reader will notice that the code provided on GitHub includes partial solutions to some of the exercises. These serve as hints, allowing the reader to explore alternative solutions.

In many cases, exercises will need unit test cases to confirm they actually solve the problem. These are often identical to the unit test cases already provided in the GitHub repository. The reader should replace the book's example function name with their own solution to confirm that it works.

WSGI application: welcome

In the *The WSGI standard* section of this chapter, a routing application was described. It showed three application routes, including paths starting with `/demo` and a special case for the path `/index.html`.

Creating applications via WSGI can be challenging. Build a function, `welcome_app()`, that displays an HTML page with some links for the demo app and the static download app.

A unit test for this application should use a mocked `StartResponse` function, and a mocked environment.

WSGI application: demo

In the *The WSGI standard* section of this chapter, a routing application was described. It showed three application routes, including paths starting with `/demo` and a special case for

the `/index.html` path.

Build a function, `demo_app()`, to do some potentially useful activity. The intent here is to have a path that responds to an HTTP `POST` request to do some work, creating an entry in a log file. The result must be a redirect (status 303, usually) to a URL that uses the `static_text_app()` to download the log file. This behavior is described as Post/Redirect/Get, and allows for a good user experience when navigating back to a previous page. See `https://www.geeksforgeeks.org/post-redirect-get-prg-design-pattern/` for more details on this design pattern.

Here are two examples of useful work that might be implemented by the demo application:

- A `GET` request can present an HTML page with a form. The submit button on the form can make a `POST` request to do a computation of some kind.
- A `POST` request can execute `doctest.testfile()` to run a unit test suite and collect the resulting log.

Serializing data with XML

In the *Serializing data with XML and HTML* section of this chapter, we described two additional features of the RESTful API built using Flask.

Extend the response in those examples to serialize the resulting data into XML in addition to CSV and JSON. One alternative to adding XML serialization is to download and install a library that will serialize `Series` and `Pair` objects. Another choice is to write a function that can work with a `list[dict[str, Any]]` object. Adding the XML serialization format also requires adding test cases to confirm the response has the expected format and content.

Serializing data with HTML

In the *Serializing data with XML and HTML* section of this chapter, we described two additional features of the RESTful API built using Flask.

Extend the response in those examples to serialize the resulting data into HTML in addition to CSV and JSON. HTML serialization can be more complex than XML serialization

because there is quite a bit of overhead in an HTML presentation of data. Rather than a representation of the `Pair` objects, it is common practice to include an entire HTML table structure that mirrors the CSV rows and columns. Adding the HTML serialization format also requires adding test cases to confirm the response has the expected format and content.

www.packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

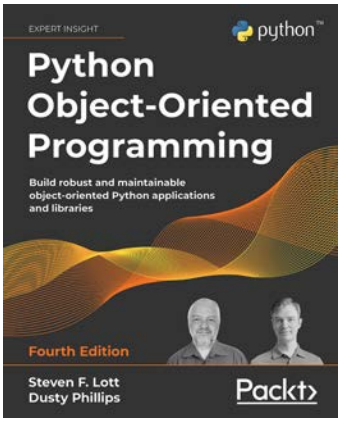
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

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

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free

Other Books You Might Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



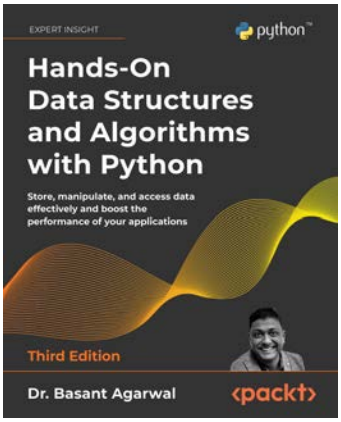
Python Object-Oriented Programming - Fourth Edition

Steven F. Lott , Dusty Phillips

ISBN: 978-1-80107-726-2

- Implement objects in Python by creating classes and defining methods
- Extend class functionality using inheritance
- Use exceptions to handle unusual situations cleanly
- Understand when to use object-oriented features, and more importantly, when not to use them
- Discover several widely used design patterns and how they are implemented in Python
- Uncover the simplicity of unit and integration testing and understand why they are so important

- Learn to statically type check your dynamic code
- Understand concurrency with asyncio and how it speeds up programs



Hands-On Data Structures and Algorithms with Python - Third Edition

Dr. Basant Agarwal

ISBN: 978-1-80107-344-8

- Understand common data structures and algorithms using examples, diagrams, and exercises
- Explore how more complex structures, such as priority queues and heaps, can benefit your code
- Implement searching, sorting, and selection algorithms on number and string sequences
- Become confident with key string-matching algorithms
- Understand algorithmic paradigms and apply dynamic programming techniques
- Use asymptotic notation to analyze algorithm performance with regard to time and space complexities
- Write powerful, robust code using the latest features of Python

Packt is searching for authors like you

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

Share your thoughts

Once you've read *Functional Python Programming, Third Edition*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback.

`https://packt.link/r/1803232579`

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

Index

Symbols

@cache decorator 343
  for memoization 345–348
@functoolz.memoize decorator 386
@lru_cache decorator 343
  for memoization 345–348
@singledispatch decorator 366–368
@total_ordering decorator 343
  classes, defining with 348–351
@wraps decorator 400

A

Accretion design pattern 88
accumulate() function 280
  running totals 284, 285
aggregate function 86
all() function
  using, as reduction 108–110
Anscombe's quartet 66
any() function
  using, as reduction 108–110
Apache httpd 494, 505
API key
  reference link 522
  using 520, 521
Application Program Interface (API) 41
assignment operator
  using, in recursions 187, 188
assignment statement 3
asyncio module 482

B

Bird-Meertens Formalism 188
bisect module
  used, for mapping 76, 77
build_duplicates() function 251

C

callables 167
  higher-order functions, building with 167, 168
Cartesian product
  enumerating 310, 311
  reducing 311–314
chain() function, itertools 280
  iterators, combining with 286, 287
chunking 163
class statement 3

Clojure 73
collection functions 86
filter 87
mapping 86
reduction 86
collections
len() and sum(), using 110, 111
processing, through recursion 184, 185
tail-call optimization 185–187
combinations
generating 329–333
combinations() function, itertools 310
combinations_with_replacement() function, itertools 310
using 333, 334
combinatorial optimization problems 326
Common Log Format (CLF) 460
Communicating Sequential Processes (CSP) paradigm 458
complex decorators
implementing 414, 415
complicated object initialization
implementing 237–240
composite function 397, 405–407
bad data, preprocessing 407–410
compress() function, itertools 280
filtering with 291–293
concurrency 455
benefits 457, 458
boundary conditions 455, 456
resources, sharing with process or threads 456, 457
consumer 479
cookies 495
coprime 35
count() function
counting with 269
count() function, itertools 269
counting with 270
float arguments, counting with 271–274
cross-cutting concerns 404
Cross-Site Request Forgeries (CSRF) 498
CSV serializer 519
CSV-based parser 211–214
curl 493
curried higher-order functions
using 431, 432
currying 42, 140, 351, 428–431
cycle() function, itertools 269
cycle, re-iterating with 275, 276
for data sampling 276–278

D

data
collecting, with NamedTuple 227
collecting, with pyrsistent 240–245
collecting, with tuples 224–226

data access tier, web service 513–515
data, serializing with JSON or CSV formats 518, 519
data, serializing with XML and HTML 519, 520
filter, applying 515, 516
results, serializing 516–518
data sets
reducing, with reduce() function 353–355
De Morgan's Law 109
decorators 397
as higher-order functions 398–403
complicated design considerations 415–420
parameter, adding 410–414
def statement 3
deques
used, for tail-call optimization 190–193
dicttoolz functions 385, 386
distances
computing 314–318
divide and conquer strategy 53, 181
Django 506
double factorial 376
dropwhile() function, itertools 281
stateful filtering 295–297

E

eager evaluation 33, 34
embarrassingly parallel design 455
enumerate() function 280
numbers, assigning with 281–284
using 127, 128
Erlang 9
Euclidean distance 315
exploratory data analysis (EDA) 2, 15
data exploration 16
data modeling and machine learning 16
data preparation 15
evaluation 16
stages 16

F

Fibonacci numbers 182
file parsers
CSV files, parsing 211–214
plain text files, parsing with headers 214–217
writing 207–211
filter() function
filtering with 297, 298
for identifying outliers 149
for passing or rejecting data 146–148
filterfalse() function 281
filtering with 297, 298
finite iterators 280
first normal form 209

first-in-first-out (FIFO) 192
Flask 506
Flask application
processing 508–512
flat sequences
structuring 120–123
structuring, alternative approach 124, 125
free variable 48
frozen dataclasses
using for data collection 233–236
function evaluation 3
functional composition 3, 428
performing, with PyMonad 432–434
functional hybrid
example 8, 9
functional paradigm
classical example 10–15
example 5–7
functional style
versus procedural style 4
functional type systems 39, 40
functions
as first-class objects 26, 27, 51–54
collection functions 86
higher-order functions 29
pure functions 27
scalar functions 86
functools module 343
functoolz functions 386
functoolz.memoize decorator 386
functoolz.compose() 387
functoolz.compose() function 389
functoolz.compose_left() function 389
functoolz.curry() function 387
functoolz.pipe() function 389
functor 434
example 435, 436

G

generator expressions
applying to scalar functions 104–108
applying, to built-in collections 69
combining 65, 66
for dicts 69–73
for lists 69–73
for sets 69–73
limitations 63–65
using 58–63
generator functions 59, 63
raw data, cleaning with 66–68
Global Interpreter Lock (GIL) 454
global statement 3
group-by reduction
data partitioning, by key values 198–202
file parsers, writing 207–211
from many items to fewer

193–195
mapping, building by sorting 197, 198
mapping, building with Counter 195, 196
writing 203, 204
groupby() function, itertools 280
iterators, partitioning with 287–289

H

Haskell 9, 39
higher-level parser 216
higher-order filters
data structuring 164–166
writing 154, 155
higher-order functions 29, 133
building, with callables 167, 168
collection, materializing 172
collection, reducing 172
design patterns, reviewing 171, 172
filter() function 146
functional design, assuring 168–170
generator, acting as 171
generator, returning 171
iter() function 150
map() function 141
max() function 134
min() function 134
scalar 172
sorted() function 151
using 30
writing 153
higher-order mappings
additional data, unwrapping 159–161
data unwrapping 156, 157, 159
data, flattening 162, 163
writing 153
higher-order reduction
writing 205–207
HTML serialization 520
HTTP request-response model 492–494
functional view 497
server with functional design, considering 496
services, nesting 498, 499
state, injecting through cookies 495

I

imap(function, iterable) method, Pool object 476
imap_unordered(function, iterable) method, Pool object 477
immutable objects 30
imperative statement 3
import statement 3
infinite iterators
count() function, counting with 269, 270

counting, with float arguments 271–274
cycle() function, for data sampling 276–278
cycle, re-iterating with cycle() function 275, 276
value, repeating with repeat() function 278–280
working with 269
islice() function, itertools 281
subsets, picking with 293–295
iter() function
with sentinel value 150, 151
iterables
file, parsing at higher level 92–95
items, pairing from sequence 95–98
iter() function, using 98–100
working with 87–89
XML file, parsing 89–92
iteration
extending 100–104
iterator functions 267
iterators
cloning, with tee() function 301
combining, with chain() function 286, 287
finite iterators 280
infinite iterators 269
partitioning, with groupby() function 287–289
itertools module 269
recipes 302
itertools module, for combinatorics 310
itertoolz functions 379, 381

J

JavaScript Object Notation (JSON) 92
JSON serializer 518

K

k-Nearest Neighbors (k-NN) algorithm 314

L

lambda calculus 140
lambda forms
map() function, working with 143
using 138–140
last-in-first-out (LIFO) 192
lazy evaluation 33, 34
len() function
using on collections 110, 111
list comprehension 69
list display 69
ListMonad() monad
using 436–440
literal matching
example 39, 40
low-level parser 215
LRU (Least Recently Used) 345

M

Manhattan distance 316
map() function
  applying 299, 301
    for applying function to collection 141
    lambda forms, working with 143
    using, with multiple sequences 144, 145
map(function, iterable) method, Pool object 476
map-reduce operation
  for sanitizing raw data 360, 361
mapping
  building, by sorting 197, 198
    building, with Counter 195, 196
    creating, with bisect module 76
mappings 86
Markov chain 441
match statement 39
max() function
  using 134–138
memoization
  with @cache and @lru_cache 345–348
middleware 505
Miller-Rabin primality tests 35
min() function
  using 134–138
monad bind() function 440
monads 42, 441
  simulation, implementing with 441, 443, 444, 446
Monte Carlo simulation 441
mr-proper tool 27
multiprocessing application
  access details, analyzing 472, 473
    access details, filtering 471, 472
    additional fields of Access object, parsing 467–470
    analysis process 473, 474
    large files, processing 460–462
    log files, parsing 462, 463
    log lines, parsing into named tuples 463–467
    rows, gathering 462, 463
multiprocessing package
  Pool object 459
multiprocessing pool, for concurrent processing 474–478
    apply(), for making single request 478
    async functions, using 482, 483
    complex architectures 479
    concurrent processing, designing 483–486
    concurrent.futures module, using 479, 480
    concurrent.futures thread pools, using 480, 481
    queue modules, using 481
    threading modules, using 481
Multipurpose Internet Mail Extension

(MIME) 497
mypy tool 3, 40, 72, 112, 193, 200, 234, 257, 500

N

named tuples
using 56–58
using for data collection 227–232
Newton-Raphson algorithm 10
NGINX 494, 505
non-strict evaluation 32, 33
nonlocal statement 3
normalization 111
numerical recursions 178, 179
assignment operator, using 187, 188
collections, processing through 184, 185
divide and conquer strategy 181, 182
manual tail-call optimization, implementing 180, 181
tail-call optimization, handling 182–184

O

Object-Oriented Programming (OOP) 31, 168, 170, 223, 470
OCaml 9
operator module functions
reducing with 377–379

P

pairwise() function, itertools
pairs, creating with 290
parameterized decorator 410–414
partial application 351
partial() function, functools 343
partial arguments, applying with 351–353
Peano axioms 178
performance improvements 320–323
transformations, combining 324–326
permutations() function, itertools 310
used, for collection of values 326–328
pixels and colors
enumerating, with product() function 318, 319
polymorphism 257–261
Pool object 459
predicate function 146
prime number
definition 34
procedural style
versus functional style 4
producer 479
product() function, itertools 310
pixels and colors, enumerating with 318, 319
pure function
writing 27, 28

pylint tool **446**
PyMonad library **42**
PyMonad package **428**
  features **447**
    functional composition, performing with 432, 434
    installing 428
pyrsistent
  using, for data collection **240–245**
pytest tool **193**
Python Imaging Library (PIL) **313**
Python Package Index (PyPI) **428**

Q

quartile **284**
queue module **481**

R

railroad diagram **464**
raw data
  cleaning with generator functions **66–68**
recursion **34–38**
    collection folding 188
    examples 5–7
    simple numerical 178
reduce() function
  combining, with map() function **356–358**
    data sets, reducing with 353–355
    limitation 360
    problems avoiding 366
    using, with groupby() function 362–365
    using, with partial() function 358, 359
reduce() function, functools **344**
reductions **86**
  collection, folding from many to one **188–190**
    group-by reduction 193–195
    tail-call optimization, deques used 190–193
referential transparency **42**
registry **397**
repeat() function, itertools
  single value, repeating with **278–280**
Representational State Transfer (REST) **491**
RESTful web service **495**
  data access tier **513–515**
    defining, as functions 506–508
    Flask application processing 508–512
    usage, tracking 520–522
reversed() function
  using **125–127**

S

Scala **39**
scalar functions **86, 105**
secrets module **521**

Secure Socket Layer (SSL) protocols 496
sequences
  flattening 118–120
serialization 516
session 495
short-circuit operators 32
Sieve of Eratosthenes algorithm 148
simulation
  implementing, with monads 441, 443–446
singledispatch() function, functools 344
sorted() function
  used, for data sorting 151, 152
  using 125–127
Spearman rank correlation 246
  computing 252–256
stacks, abstractions 9, 10
starmap() function, itertools 281, 374
  applying 298, 299, 301
starmap(function, iterable) method, Pool object 477
state 3
stateful mappings
  using 73–76
stateful sets
  using 78
statistics
  sums and counts, using 111–116
Strategy design pattern 52–54
strict evaluation 32
strings 54–56
subsets
  picking, with islice() function 293–295
sum() function
  using on collections 110, 111
sum_to() function 34

T

tail recursion
  example 38
tail-call optimization
  deques, using 190–193
  for collections 185–187
  handling 182–184
  manual implementation 180, 181
takewhile() function, itertools 281
  stateful filtering 295–297
tee() function, itertools
  iterators, cloning with 301
threading module 481
toolz package 373
  dicttoolz functions 385, 386
  functoolz functions 386, 387, 389, 390
  itertoolz functions 379–384
  using 379
tox tool 193
Trial Division algorithm 35
tuples
  used, for avoiding stateful classes 246–252
  using 56–58

using for data collection 224–226
two-tier parser 217
type pattern matching 257–261

U

unwrapping-while-mapping design pattern 157
update_wrapper() function, functools 403
user experience (UX) design 493

W

walrus operator
  using 187, 188
Web Server Gateway Interface (WSGI) standard 499–503
  exceptions, raising 503–505
  pragmatic web applications 505, 506
werkzeug.wsgi module 500
wget 493
wrap-process-unwrap pattern
  using 30, 31
wrap-unwrap design pattern 246

X

XML Schema Definition (XSD) 520
XML serialization 519

Z

zip() function 116
  example 116, 117
zip_longest() function, itertools 280
  iterables, merging with 290
zipped sequence
  unzipping 117, 118

Download a free PDF copy of this book

Thanks for purchasing this book!

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

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

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

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

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/9781803232577`

2. Submit your proof of purchase

3. That's it! We'll send your free PDF and other benefits to your email directly